Source Listing for C+−

Ulrich Mutze
www.ulrichmutze.de

October 25, 2008

Contents

1 Introduction 12

2 Credit 13

3 Legal Matters 14

4 chebyshev.cpp 15

5 cpmactionprinciple.h 28

6 cpmactionprinciple.cpp 29

7 cpmalgorithms.h 36

8 cpmalgorithms.cpp 57

9 cpmangle.h 80

10 cpmangle.cpp 85

11 cpmapplication.h 93

12 cpmapplication.cpp 98

13 cpmbas.h 107

14 cpmbas.cpp 108

15 cpmbasicinterfaces.h 109

16 cpmbasictypes.h 111

17 cpmbody.h 113

18 cpmbody.cpp 115

19 cpmbodycpp.h 116

20 cpmbodyx.h 135

21 cpmc.h 157

22 cpmc.cpp 164

23 cpmcamera.h 171

24 cpmcamera.cpp 172

25 cpmcamera2.h 173

26 cpmcamera2cpp.h 178

27 cpmcamera3.h 184

28 cpmcamera3cpp.h 190

29 cpmcamerax.h 202

30 cpmcameraxcpp.h 209

31 cpmclinalg.h 219

32 cpmclinalg.cpp 233

33 cpmcompar.h 245

34 cpmcompar.cpp 247

35 cpmcomparcpp.h 249

36 cpmcomparx.h 304

37 cpmconstphys.h 340

38 cpmconstphys.cpp 343
39 cpmcvector.h 344
40 cpmcvector.cpp 349
41 cpmdefinitions.h 352
42 cpmdigraph.h 354
43 cpmdigraph.cpp 367
44 cpmdim.h 368
45 cpmdim.cpp 370
46 cpmdim2.h 371
47 cpmdim2cpp.h 389
48 cpmdim3.h 400
49 cpmdim3cpp.h 424
50 cpmdimdef.h 443
51 cpmdimx.h 444
52 cpmdimxcpp.h 470
53 cpmdiscretespace.h 518
54 cpmdistribution.h 521
55 cpmdistribution.cpp 526
56 cpmdynaux.h 531
57 cpmdynaux.cpp 534
58 cpmextractor.h 536
59 cpmextractorx.h 538
60 cpmf.h 548

61 cpmfa.h 558

62 cpmfft.h 567

63 cpmfft.cpp 571

64 cpmfftmore.h 578

65 cpmfftmore.cpp 583

66 cpmfield.h 590

67 cpmfieldoperator.h 595

68 cpmfieldoperator.cpp 598

69 cpmfieldstate.h 602

70 cpmfieldstate.cpp 612

71 cpmfile.h 631

72 cpmfl.h 634

73 cpmfo.h 646

74 cpmfontgeneration.h 650

75 cpmfontgeneration.cpp 656

76 cpmforces.h 672

77 cpmforces.cpp 674

78 cpmforcescpp.h 675

79 cpmforcesx.h 705

80 cpmfr.h 734

81 cpmframe.h 741

82 cpmframe.cpp 759

83 cpmframes.h 777

84 **cpmframes.cpp** 779

85 **cpmgraph.h** 782

86 **cpmgraph.cpp** 801

87 **cpmgreg.h** 842

88 **cpmgreg.cpp** 848

89 **cpmgroupgraph.h** 856

90 **cpmh.h** 858

91 **cpmh.cpp** 862

92 **cpmhistogr.h** 865

93 **cpmhistogr.cpp** 868

94 **cpmhomunitvecxcpp.h** 873

95 **cpmimage.h** 876

96 **cpmimagingtools.h** 909

97 **cpmimagingtools.cpp** 921

98 **cpmimg24.h** 936

99 **cpmimg24.cpp** 949

100 **cpminifilebasapp.h** 970

101 **cpminifilebasapp.cpp** 982

102 **cpminterfaces.h** 998

103 **cpminterval.h** 1007

104 **cpminterval.cpp** 1015

105 **cpmlinalg.h** 1024

106 **cpmlinalg.cpp** 1040

107 cpmlineart.h **1057**

108 cpmlineart.cpp **1058**

109 cpmlineartcpp.h **1059**

110 cpmlineartx.h **1119**

111 cpmlogic1.h **1127**

112 cpmlogic1.cpp **1139**

113 cpmm.h **1141**

114 cpmmacros.h **1151**

115 cpmmathfound.h **1153**

116 cpmmathfound.cpp **1155**

117 cpmmaxw1d.h **1157**

118 cpmmaxw1d.cpp **1165**

119 cpmmini.h **1168**

120 cpmmini.cpp **1172**

121 cpmmpi.h **1181**

122 cpmmpi.cpp **1188**

123 cpmmultiimage.h **1193**

124 cpmnr.h **1231**

125 cpmnr.cpp **1234**

126 cpmnumbers.h **1240**

127 cpmnumbers.cpp **1266**

128 cpmobservable.h **1274**

129 cpmobservable.cpp **1286**

130 cpmoperator.h **1318**

131 cpmoperator.cpp **1321**

132 cpmp.h **1326**

133 cpmpardynsys.h **1333**

134 cpmpardynsys.cpp **1386**

135 cpmparens.h **1387**

136 cpmparensx.h **1389**

137 cpmparofspc.h **1395**

138 cpmparofspcx.h **1397**

139 cpmparticleviewer.h **1435**

140 cpmperm.h **1442**

141 cpmperm.cpp **1444**

142 cpmproperties.h **1447**

143 cpmproperties.cpp **1456**

144 cpmpsi.h **1473**

145 cpmpsi.cpp **1477**

146 cpmq.h **1483**

147 cpmq.cpp **1485**

148 cpmqm.h **1490**

149 cpmqm.cpp **1526**

150 cpmqmtemplates.h **1608**

151 cpmr2func.h **1617**

152 cpmr2func.cpp **1621**

153 cpmrecord.h 1630

154 cpmrecord.cpp 1644

155 cpmrecordhandler.h 1669

156 cpmrecordhandler.cpp 1674

157 cpmrfunc.h 1683

158 cpmrfunc.cpp 1695

159 cpmrfuncrec.h 1726

160 cpmrfuncrec.cpp 1729

161 cpmrigbodydyn.h 1732

162 cpmrigbodydyn.cpp 1738

163 cpmrmateig.h 1753

164 cpmrmateig.cpp 1756

165 cpmrmatrix.h 1764

166 cpmrmatrix.cpp 1773

167 cpmrnfunc.h 1789

168 cpmroot.h 1798

169 cpmroot.cpp 1801

170 cpmrvector.h 1805

171 cpmrvector.cpp 1814

172 cpms.h 1823

173 cpmset.h 1838

174 cpmset.cpp 1848

175 cpmshapes.h 1860

176 cpmshapes.cpp **1861**

177 cpmshapesx.h **1862**

178 cpmshapesxcpp.h **1868**

179 cpmsi.h **1875**

180 cpmsi.cpp **1881**

181 cpmsmoothlocalinterpol.h **1891**

182 cpmsmoothlocalinterpol.cpp **1894**

183 cpmsr.h **1908**

184 cpmstate.h **1915**

185 cpmstate.cpp **1930**

186 cpmstate2.h **1952**

187 cpmstate2.cpp **1960**

188 cpmsys2.h **1976**

189 cpmsys2.cpp **1978**

190 cpmsys2aps.h **1979**

191 cpmsys2aps.cpp **1981**

192 cpmsys2apscpp.h **1982**

193 cpmsys2cpp.h **1989**

194 cpmsys2dat.h **2022**

195 cpmsys2dat.cpp **2023**

196 cpmsys2datcpp.h **2024**

197 cpmsys2datx.h **2027**

198 cpmsys2x.h **2029**

199 cpmsystem.h 2036

200 cpmsystem.cpp 2048

201 cpmsystemdependencies.h 2061

202 cpmsysviewer.h 2063

203 cpmsysviewer.cpp 2064

204 cpmsysviewercpp.h 2068

205 cpmtestr.h 2070

206 cpmtrj.h 2098

207 cpmtrj.cpp 2113

208 cpmtypes.h 2133

209 cpmtypes.cpp 2143

210 cpmuc.h 2151

211 cpmuc.cpp 2154

212 cpmunits.h 2155

213 cpmv.h 2157

214 cpmva.h 2178

215 cpmvc.h 2190

216 cpmvc.cpp 2193

217 cpmvectorgraphics.h 2196

218 cpmvectorgraphics.cpp 2201

219 cpmvectormore.h 2204

220 cpmvectormore.cpp 2214

221 cpmviewport.h 2235

222 cpmviewport.cpp 2245

223 cpmvl.h 2257

224 cpmvl.cpp 2287

225 cpmvo.h 2289

226 cpmvr.h 2302

227 cpmwaves.h 2305

228 cpmwaves.cpp 2309

229 cpmword.h 2314

230 cpmword.cpp 2326

231 cpmx.h 2337

232 cpmzinterval.h 2352

233 cpmzinterval.cpp 2356

234 pala.cpp 2359

235 smallexperiments.cpp 2362

236 smallexperiments2.cpp 2377

237 test1a.cpp 2390

238 test1b.cpp 2394

239 testcpm0.cpp 2401

240 vqm2.cpp 2404

241 vqm2app1.cpp 2406

242 vqm2app2.cpp 2490

243 vqm2app3.cpp 2513

244 survey.txt 2529

245 headerdependencies.txt 2559

1 Introduction

This is a listing of all C+− source files. See file *cpmProject.pdf* for motivation and details concerning the C+− project.

The files with names beginning with `cpm` constitute a library (i.e. a set of programming tools) whereas the other files are taken from my C+− - based applications. These are included to provide examples for usage of the tool classes. The search capability of the *Adobe Reader* program allows to find places where specified classes are employed. These search capabilities allow to extract from a verbatim code listing all informations that one may obtain from the graphically more appealing documentations generated by the *doxygen* program. As an example for this, assume that we like to know what the cryptic files `cpmparens*.*` are about. File `cpmparens.h` refers to file `cpmparensx.h` and here we learn that the name stands for `particle ensemble` and that the files define a class template `ParEns<>`. To see what the role of this class is we do a case sensitive search for for `ParEns<` and find that in file `cpmrigbodydyn.cpp` the function

```
void CpmPala1_3::App1RigBodyDyn::doTheWork()
```

makes a call

```
CpmRigidBody3::ParEns<ParT> pe(gp,box,facVol,nLim);
```

which creates an instance `pe` of class `ParEns<ParT>` for given design parameters `gp,box,facVol,nLim`. This object delivers the desired output (an array of particles) via call of

```
ParEns<ParT>::operator()(void)
```

in the statement

```
V<ParT> vcp=pe();
```

It is easily seen that no other usage of this class template occurs in the file system. It may look strange that a class has been defined in order to provide a method for placing a set of particles in a given box. There is, however, a good reason for proceeding this way. The main functionality is removed from the final application code into the tool code, where it is available for similar services in similar situations. A further convenience is that for class code there are natural places for most of the necessary comments, so that most of the readers will find the explanations they are looking for. This finally is the reason why code organized in classes can much better serve as its own documentation than any other type of code.

In the following part of the document, there is one section (in the sense of LaTex) for each file and the files are in alphabetic order with the exception that header files (*.h) precede implementation files (*.cpp) of the same primary name. Since there is table of content, that shows these sections, it is not difficult to navigate within this system of more than 200 files. Some of the implementation code is intended to become included in *.cpp files to define different functions. Such code is in files *cpp.h, so that the nature of a file in the dichotomy declaration vs. definition is indicated by appearing of 'cpp' as a file name constituent, not necessarily by the file extension. The main occurrence of this included implementation is in common treatment of 2D and 3D versions of geometry and physics related classes.

In trying to get an overview of the content of this file collection one should also consult the final files survey.txt and headerdependencies.txt. As is mentioned in the header of these files they are auto-generated by means of *Ruby* programs. I would not have been able to carry out all the testing and documentation of C+− files without my rich tool box of Ruby programs. In filling this tool box piece by piece, I became a Ruby enthusiast, wheres in the beginning it was more or less by chance that I turned to just this language. I was happy enough that, when the need of extended scripting occurred to me, Ruby was available. Those of my colleagues for whom this need arose earlier, had their personal toolbox yet full of *Perl* scripts and for me — an enthusiast of C++ and an admirer of Stroustrup's writing — it was hard to understand their enthusiasm for Perl. So I tried Ruby and found it to be in some (restricted) sense 'a better C++'. For some of C+− programming style decisions, Ruby was the model, and for others, for which agreement with Ruby strategies became clear after their introduction to C+− this backed up my confidence to be on a viable track.

2 Credit

Many of the more complex algoritms contained in the present code are modified from sources in the wonderful book *Numerical Recipies in C* by William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, of which I own and use a copy of the second edition. In all these uses I made sustantial modifications by introducing the C+− data and array types, promoting C-functions to member functions of suitable C++ classes, and by iserting diagnostic message generators. Searching for '*Press*' in the present document shows all instances of such uses. For the programming language related matters my sources of advice and inspiration are indicated in file *cpmProject.pdf*.

3 Legal Matters

Copyright (c) 2007-2008 Ulrich Mutze, Bad Ditzenbach, Germany
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE.

4 chebyshev.cpp

```
//? chebyshev.cpp
//? Status of work 2008-10-25.
//?
/************************************************************************

This program demonstrates the C+- programming style,
particularly the usage of functions as values.
That most functions in the demonstration are Chebyshev polynomials
is not important for this purpose.
The title of the program could be misleading in this respect.

Since the present program is intended as a vehicle not only
for demonstration but also for explanation,
it contains much more comments than would be appropriate for a
typical program.

The program is controled by the text file chebyshev.ini.

Here is a copy of the intoductory part of this file which gives
more information on the topics the program deals with:

//////////////////////////////////////////////////
   about
//////////////////////////////////////////////////
W file=chebyshev.ini
Ws line1=Graphical representation of Chebyshev polynomials.
Ws line2=Testing orthogonality by Romberg integration.
Ws line3=Finding the proper parameters of the orthogonalizing
Ws line4=weight function by minimization.
Ws line5=Visualization of scalar products.

These are the program specific data that go into the creation
of the start-up screen of the project. Other elements of
this start-up screen are common to all C+- programs which
use class CpmApplication::IniFileBasedApplication
(see file cpminifilebasapp.h) as an application framework.
The definition of function IniFileBasedApplication::announceTheWork()
in file cpminifilebasapp.cpp defines these additional elements.

Notice also the following section, which any ini-file is expected to
have and which allows to influence the program run and the
auto-generated documentation in many ways. Although
ini-files are not seen by the compiler, they read like
C++ code by having type indicators for their data entries.
Also comments are treated as in C++.
Valid type indicators are B, Z, R, W for bool, integer, real, and
character string and the corresponding lists Bs, Zs, Rs, Ws.
```

There is a further valid type indicator F, which introduces the name of ini-file to be included. So ini-files nest to an arbitrary level. This feature was developed for an industrial project where all data defining the kinematics of a complex robot were originally held in a single ini-file.
With the number of data growing to several thousands and with people from other groups becoming involved in adjusting the data, it became mandatory to split the data to separatly editable units and nevertheless ensure integrity of production runs by having a single ini-file at work.
Several of my C+- programs make use of this facility for similar reasons.

Text lines in which there is no data type indicator are section headings. Sections behave very much like namespaces: Althoug section 'about' (see above) has an entry 'file' a section with different name may also have an entry named 'file'. Sections don't nest, so that several F-included files may add to the same section, which turned out to be very convenient in the framework for which I developed the scheme.

```
//////////////////////////////////////////////////////////////////////
   selection
//////////////////////////////////////////////////////////////////////

R tWaitAnnounceScreen=0

Z sel=1
// a selector for the 'depth' of auto-generated documentation files.
// Control of auto-generated documentation files is based on the
// two entries sel, runName. These are the rules:
// 1. for sel<0 no documentation files will be created.
// 2. for sel==0 a light documentation file is created that
//   covers all the input so that the program run can be repeated.
// 3. for sel>0, in addition to the light documentation file
//    a more detailed one will be created that also can made to
//    list computational results.
// 4. All documentation files have names which contain runName
//    appended to the name of the calling program

Z cpmverbose=1
// controls the degree of detail with which program run
// information will be written to log file cpmcerr.txt

Z cpmdbg=2
// controls the reaction of the cpmassert macro:
// 1: warning on failing assertion
// 2: error stop on failing assertion

W runName=070830a
```

```
// identifier for the program run that will be appended to
// the names of documentation files.
// If the value of this quantity is "auto" this value will
// be replaced by an autogenerated name.

*********************************************************************/

#include <cpmv.h>
#include <cpmgraph.h>
#include <cpmframes.h>
#include <cpmfa.h>
#include <cpminifilebasapp.h>
#include <cpmmini.h>

using namespace CpmRoot;
using namespace CpmFunctions;
using namespace CpmSystem;
using namespace CpmArrays;
using namespace CpmGraphics;
using namespace CpmImaging;
using namespace CpmApplication;
using namespace std;

class ChebyApp: public IniFileBasedApplication
{
   public:
      ChebyApp();
      void doTheWork();
};

ChebyApp::ChebyApp(void):IniFileBasedApplication("chebyshev"){}
// This line implies that the program relies on a file named
// chebyshev.ini for getting the data which determine a
// program run. So we are in a position to tell the program
// in advance what it should do. If it is running, our only
// way to influence it is to cancel it.
// The location of this ini file can be given as a second argument to the
// constructor such as
// ChebyApp(void):IniFileBasedApplication("chebyshev","./control"){}
// Or, more flexibly by an entry to the ini-file cpmconfig.ini


namespace{ // Material needed in ChebyApp::doTheWork().
   // What goes on here, is easily understood taking into
   // account the usage there.
   // Functions don't nest in C++, so we have to keep
   // them outside of other functions.
   // C+- has the tools to finally use such outside function code
   // from any place in a very convenient manner. Given these tools,
```

```
// it looks like a happy idea in C++ to prevent functions
// from nesting. One would need more formal machinery
// to use nested functions in the same manner as
// we use top-level functions here.

R rwsp(R const& x, R const& eps, R const& p)
   // regularized weight (in) scalar product
{
   R y=cpmabs((1-x)*(1+x))+eps*eps;
   return ::pow(y,p); // y>=0
}

R wsp(R x){ return rwsp(x,0,-0.5);}
   // exact orthogonalizing weight (in) scalar product

R_Matrix sclPrd( V<R_Func> const& vf, R_Func const& w, Iv const& iv,
   RomCon const& rc=RomCon())
   // building a matrix of scalar products of the functions from list
   // vf. The scalar products employ a weight function w an
   // integration interval iv and depend on the method parameters rc
   // which control the integration method, which is Romberg
   // integration in our case.
{
   Z done=0;
   Z i,j,n=vf.dim();
   R_Matrix sp(n,n);
   R doneInv=2./(n*(n+1));
   for (i=1;i<=n;++i){
      for (j=i;j<=n;++j){
         R_Func fij=vf[i]*vf[j]*w;
         R rij=fij.romInt(iv[1],iv[2],rc);
         sp[i][j]=rij;
         done++;
         if (i!=j) sp[j][i]=rij;
       }
   }
   return sp;
}

R fAux( V<R_Func> const& vf, R_Func const& w, Iv const& iv,
   RomCon const& rc=RomCon())
// Instead of the matrix of scalar products we return the
// deviation of this matrix from the unit matrix measured
// by a real number.
{
   R_Matrix sp=sclPrd(vf,w,iv,rc);
   Z n=sp.dim();
   R_Vector dia(n,1.);
   R_Matrix diag(dia);
   R_Matrix diff=sp-diag;
```

```
      return diff.absVal()/(n*n);
   }

   R f2Aux(R_Vector const& p, V<R_Func> const& vf, RomCon const& rc)
      // Just as the previous function, but now function w is
      // parametrized by parameters p[1],p[2] and function code rwsp as
      // given as the first piece in the present namespace. In this form
      // the function will be used in the minmization part of the program.
   {
      R eps=p[1];
      R exponent=p[2];
      Iv iv(-1,1);
      R_Func w=F2<R,R,R,R>(eps,exponent)(rwsp);
      R res=fAux(vf,w,iv,rc);
      cpmvalue("eps",eps,2);
      cpmvalue("exp",exponent,3);
      cpmvalue("res",res,4);
         // writing to the status bar
      return res;
   }
}

void ChebyApp::doTheWork()
// does what the name says
{
   Word loc("ChebyApp::doTheWork()");
   Z mL=1;
   CPM_MA
// data input
// It is instructive to inspect file chebyshev.ini from where the
// data come.
bool writeOnShow=(cpmverbose>1);
#if defined(CPM_NOGRAPHICS)
   writeOnShow=true;
#endif
   Word sec("Show curves");
      // We read data for the first section of the program
      // from the first program specific section of the ini-file
   Z nL,nU,graphRes,bgcR,bgcG,bgcB;
   R xL,xR,tWaitGrp,gamma=-1;
   bool noGridText, noGrid, autoScale, antiAliasing,
      curvesBlack;
      // Here we declare the quantities which we will 'read in from the
      // ini-file'. To add a new quantity here, and a new corresponding
      // read statement, and the corresponding data entry in the ini-file,
      // is a simple and structured activity which is not likely to
      // produce errors. If it does nevertheless, the program will halt
      // with a clear statement of the problem on the log file
      // cpmcerr.txt.
   cpmrh(nL);
```

```
      // Reading from the RecordHandler instance rch, which is the
      // most important data element of class IniFileBasedApplication.
      // rch gets initialized from the ini-file  by means of the
      // constructor of IniFileBasedApplication.
      // RecordHandler is a container for data of various types and
      // has the capability of distributing his data to various processes
      // in MPI-based parallel processing. See file cpmrecordhandler.h
      // for details. For single process programming it is sufficient
      // to think of these read statements as 'reading from the ini-file'
      // and ignoring the fact that here the data actually come out of
      // the container rch.
      // cpmrh(x) stopps as error, if x cannot be found. Either it is not
      // is not there by omission, or the ini-file syntax is violated in
      // in the intended definition of x.
      // cpmrhf(x) (f for 'floppy') gives only a warning on the log file
      // in such a case.
   cpmrh(nU);
   cpmrh(graphRes);
   cpmrh(xL);
   cpmrh(xR);
   cpmrh(tWaitGrp);
   cpmrh(bgcR);
   cpmrh(bgcG);
   cpmrh(bgcB);
   cpmrh(curvesBlack);
   cpmrh(noGridText);
   cpmrh(noGrid);
   cpmrh(autoScale);
   cpmrh(antiAliasing);
   cpmrhf(gamma); // see previous explanation of cpmrhf

// work
   Z n=nU-nL+1;
   V< Fa<R,R> > ch(n);
   Fa<R,R> ACos(acos);
   Fa<R,R> Cos(cos);
      // bringing C -functios cos and acos into the F<R,R>
      // framework
   R ni=nL;
   Z i;
   for (i=ch.b();i<=ch.e();++i)
   {
      ch[i]=(ACos*ni)&Cos;
         // building a list of Chebyshev functions. This corresponds to
         // T_n(x)=cos(n*arccos(x)). Notice that the concatenation
         // '&' of functions is the kind which in mathematics is mostly
         // denoted by ';' Mathematical standard concatenation is
         // circ() in C+-, so one could equally well write
         // ch[i]=Cos.circ(ACos*ni);
         // Notice (ACos*n)(x) == ACos(x)*n   and  n o t
```

```
      // (ACos*n)(x )== ACos(x*n)
   ni+=1;
}
R facN=1/sqrt(Pi/2);
R facN0=1/sqrt(Pi);

V< R_Func > chNor(n); // Nor: normalized
   // to be used in the study of scalar products, the
   // original (not normalized) version will be
   // used for graphical representation.
for (i=ch.b();i<=ch.e();++i)
{
   chNor[i]= (i==1&& nL==0) ? ch[i]*facN0 : ch[i]*facN;
}
R_Func Wsp(wsp); // use outside function wsp
Color bgc(bgcR,bgcG,bgcB);
Graph gr(bgc);
   // If backgroud color does not matter, Graph gr;
   // is sufficient to have a graphical window of maximum size
   // and a wealth of capabilities.
const Z twp=2; // selects the pane (on status bar) on which the
   // remaining waiting time (if there is one) will be displayed
if (tWaitGrp>0){
   // it is convenient to switch off a program step by setting the
   // time scheduled for inspection of its result as <=0.
   cpmmessage("Graphics",4); // there ate 4 status panes and we
      // indicate on the last one what we presently are doing
   gr.setX(Iv(xL,xR));
      // setting the x-range to which the horizontal extension of
      // the graphics window corresponds.
      // It is interesting to set e.g. xL=0.999, xR=1
      // nL=50 nU=150 to see the behavior of the curves at the
      // upper end of their natural domain.
   gr.setY(Iv(-1,1));
      // Since all Chebyshev polynomials vary in [-1,1] this is
      // a natural setting for the y-range. Can be overridden
      // by asking for auto scaling.
   gr.setNoGridText(noGridText);
      // grid text gives the actual x,y-range associated with the
      // window. Whether it should be shown can be selected
   gr.setNoGrid(noGrid);
      // showing a grid can be enabled or disabled
   gr.setAutoScale(autoScale);
      // controls auto scaling
   gr.setAntiAliasLines(antiAliasing,gamma);
      // controls drawing of the curves
   gr.setWriteOnShow(writeOnShow);
   if (curvesBlack){ // normally a rainbow color code is used
      // for showing families of curces
      V<Color> cl(5,BLACK);
```

```
         gr.show(ch,graphRes,cl);
            // ch is a family of curves. Here, these are
            // the Chebyshev polynomials as selected by the
            // integers nL and nU
      }
      else gr.show(ch,graphRes);
         // graphRes sets the number of x-Values to be used
         // in converting curves to polygons for visualization
      cpmwait(tWaitGrp,twp); // causes the program to wait
         // for as many seconds as the first argument indicates. The
         // remaining time will be displayed on status pane twp
   }

//////////////////////////////////////////////////////////////////////

   sec="Integration";
     // We read data for the second section of the program
     // from the second program specific section of the ini-file
   Z iMethod,iMax,ip,nChebyshev;
   R epsRel,epsAbs,tWaitInt1,tWaitInt2;

   cpmrh(tWaitInt1);
   cpmrh(tWaitInt2);
   cpmrh(epsRel);
   cpmrh(epsAbs);
   cpmrh(iMethod);
   cpmrh(iMax);
   cpmrh(ip);
   cpmrh(nChebyshev);

   RomCon rc(epsRel,epsAbs);
   rc.setMet_(iMethod);
   rc.setCntMax_(iMax);
   rc.setLen_(ip);
   if (tWaitInt1>0){
      cpmmessage("Integration 1",4);
      R norF=nChebyshev==0 ? facN0 : facN;
      R nC=nChebyshev;
      Fa<R,R> fn=(ACos*nC)&Cos;
      fn*=norF;
      R_Func g=Wsp*fn*fn;
      R sL=-1, sU=1;
      R vExact=1;
      R val=g.romInt(sL,sU,rc);
      R err=vExact-val;
      cpmvalue("val",err,3);
      V<Word> lines(0);
      lines<<"Computing the norm of the normalized \
Chebyshev function of index "+cpm(nChebyshev);
         // to avoid long lines in the present source file, not in the
```

```
         // screen output
      lines<<"Method is Romberg integration with method parameters";
      lines<<"as given in the section Integration of the ini-file.";
      lines<<"Deviation from the mathematical result is "+cpm(err);
      gr.clr();
      gr.setText(lines);
      gr.display(writeOnShow);
      cpmwait(tWaitInt1,twp);
   }
   if (tWaitInt2>0){
      cpmmessage("Integration 2",4);
      R_Func g=R_Func(sin);
      R sL=0, sU=Pi;
      R vExact=2; // math knows
      R val=g.romInt(sL,sU,rc);
         // integral of g from sL to sU using Romberg with
         // method parameters rc
      R err=vExact-val;
      cpmvalue("val",err,3);

      V<Word> lines(0);
      lines<<"Integration of function sin from 0 to pi.";
      lines<<"Method is Romberg integration with method parameters";
      lines<<"as given in the section Integration of the ini-file.";
      lines<<"Deviation from the mathematical result is "+cpm(err);
      gr.clr();
      gr.setText(lines);
      gr.display(writeOnShow);
      cpmwait(tWaitInt2,twp);
   }

//////////////////////////////////////////////////////////////////////
   sec="Minimization";
     // We read data for the third section of the program
     // from the third program specific section of the ini-file
   R tWaitMin,epsStart,expStart,epsMin;

   cpmrh(tWaitMin);
   cpmrh(epsStart);
   cpmrh(expStart);
   cpmrh(epsMin);

   if (tWaitMin>0){
      cpmmessage("Minimization",4);
      gr.clr(); // clear
      V<Word> lines; // preparing the making of screen text
      lines<<"Minimizing the deviation from ortho-normalization";
      lines<<"by adjusting two parameters in the weight function.";
      lines<<"";
      lines<<
```

```
        "Observe the values shown on the 4 panes of the status bar.";
// string continuation used since for supporting my present code listing
// procedure, there should not be more than 75 characters in a line.
// There are many ways to achieve this.
      lines<<"pane 1: general messages and warnings";
      lines<<"pane 2: present values of regularizing epsilon, 0 is the\
 exact value";
      lines<<
        "pane 3: present values of the exponent, -0.5 is the exact value"\
        ;
      lines<<"pane 4: present function value, exact minimum value of\
 which is 0";
      lines<<"";
      lines<<"Minimization is running .... ";

      gr.setText(lines);
      gr.display(writeOnShow); // showing the screen text

      F<R_Vector,R> fMin= // this is the function to be minimized
         F2<R_Vector,V<R_Func>,RomCon,R>(chNor,rc)(f2Aux);
         // Notice that F<R_Vector,R> is the type expected by class
         // CpmAlgorithms::Mini. All pertinent quantities that are not
         // represented by the R_Vector-typed function argument have
         // to be initialized by quantities which the program has
         // already defined. These are the objects chNor and rc
         // and the piece of function code f2Aux. The syntax that
         // brings these entities together to form a F<R_Vector,R>
         // is simple and clear: f2Aux represents a mapping
         // f2Aux : R_Vector x V<R_Func> x RomCon --> R
         // (consider its declaration abstracted to:
         //       R f2Aux(R_Vector,V<R_Func>,RomCon)
         // )
         // The two parameters (to which '2' in 'F2<...>' corresponds)
         // chNor, rc of types V<R_Func> and RomCon bind the
         // last two factors of the Cartesian product to fixed values
         // leaving us with a function R_Vector --> R for which we
         // introduce the name fMin. It is fun to employ this scheme even
         // with more parameters (using F3<>, F4<>, F5<>, ... ).
         // If in the heat of the moment a mismatch of types and arguments
         // happened, the error will be caught by the compiler and the
         // diagnosis will be easy to understand. Notice that fMin, as
         // defined here, shows strictly the behavior of a F<R_Vector,R>.
         // The values of the parameters that were used in the definition
         // of fMin are completely hidden --- just as the many polynomial
         // coefficients on which the actual implementation of the
         // C-function sin() is built. Nevertheless, these parameters
         // must be 'somewhere' for the program to use them in evaluating
         // fMin. The definition of the class templates F2<>,..., F6<> in
         // files cpmfl.h and cpmf.h shows how this works --- the template
         // instantiation mechanism does the trick.
```

```
      CpmAlgorithms::Mini mini(fMin,epsMin);
         // instance of a minimizer class
      R_Vector pStart("",epsStart,expStart);
      R tStart=cpmtime();
      X2<R_Vector,R> res=mini(pStart);
         // makes the minimizer run, needs a proposal for a starting
         // value. The return value is a pair of values.
      R tEnd=cpmtime();
      Z nEval=mini.nEval();
      R tComp=tEnd-tStart;
      R tPerEval=tComp/nEval;
      R_Vector p=res.c1();
      R yMin=res.c2();
      R eps=p[1];
      R exponent=p[2];
// communicating the result
      cpmvalue("eps",eps,1);
      cpmvalue("exp",exponent,2);
      cpmvalue("yMin",yMin,3);
      gr.clr();
      lines=V<Word>();
      lines<<"Minimization came to an end";
      lines<<"Minimum function value was found as "+cpm(yMin)+
         ", exact value is 0";
      lines<<"eps was found as "+cpm(eps)+", exact value is 0";
      lines<<"exponent was found as "+cpm(exponent)+
         ", exact value is -0.5";
      lines<<"Computation time was "+cpm(tComp)+" seconds";
      lines<<"Number of function evaluations was "+cpm(nEval);
      lines<<"Thus the time per function evaluation was "+cpm(tPerEval)+
         " seconds";
      gr.setText(lines);
      gr.display(writeOnShow);
      cpmwait(tWaitMin,twp);
   }

   sec="Show orthogonality";
     // We read data for the fourth section of the program
     // from the fourth program specific section of the ini-file
   R tWaitShowOrt,eps=1e-6,exponent=-0.5;
   bool interpolate, showDiff;

   cpmrh(tWaitShowOrt);
   cpmrh(showDiff);
   cpmrh(gamma);
   cpmrh(interpolate);
   cpmrh(eps);
   cpmrh(exponent);
   if (tWaitShowOrt>0){
```

```
Viewport::clrPan();
cpmmessage("Show orthogonality",4);
Frame fr;
Frames frs(fr,1,2); // creates a 1x2 matrix of subframes
   // thus 2 frames in a line
Graph gr1(frs[1][1]);
Graph gr2(frs[1][2]);
V<Word> lines;
Z nF=chNor.dim();
lines<<"Showing the mutual scalar products of the "+cpm(nF)+
   " normalized";
lines<<"Chebyshev polynomials under consideration as a matrix.";
if (showDiff){
   lines<<"Actually, the unit matrix is subtracted from the";
   lines<<"matrix of scalar products.";
}
lines<<"Value of gamma is "+cpm(gamma);
lines<<"Value of epsilon is "+cpm(eps);
lines<<"Value of exponent is "+cpm(exponent);
lines<<"For eps=0 and exponent = -0.5 deviations of the matrix of";
lines<<"scalar products from the unit matrix are due to numerical";
lines<<"errors in Romberg integration with method parameters";
lines<<"chosen as follows:";
lines<<rc.show();
lines<<"";
lines<<"Computing graphics ...";
gr1.clr();
gr1.setText(lines);
gr1.display(writeOnShow);

R_Func weight=F2<R,R,R,R>(eps,exponent)(rwsp);
R_Matrix ort=sclPrd(chNor,weight,Iv(-1,1),rc);
if (showDiff){
   R_Vector dia(n,1.);
   R_Matrix diag(dia);
   ort-=diag;
}
R absMax=ort.supAbs();
Vl< Vl<C> > vs(n, Vl<C>(n));
for (i=0;i<n;++i) for (Z j=0;j<n;++j) vs[i][j]=C(ort[i+1][j+1]);
Color::setGamma(gamma);
gr2.clr();
gr2.mark(vs,Iv(),interpolate);
gr2.setText("Maximum of absolute values is "&cpm(absMax));
gr2.display(writeOnShow);

lines.last()="Computation done";
gr1.clr();
gr1.setText(lines);
gr1.display(writeOnShow);
```

```
      cpmwait(tWaitShowOrt,twp);
   }
   CPM_MZ
}

Z CpmApplication::main_(void)
// In this form the C++ typic function main()(defined in file
// cpmapplication.cpp) expects to see the basic functionality of
// the program.
// Notice that main() in cpmapplication.cpp provides the program
// with its graphical window and a status bar.
// The fact that class ChebyApp is derived from class
// IniFileBasedApplication lets the following short code, although it
// looks like abstract nonsense, do very specific things.
{
   Word loc("Z CpmApplication::main_(void)");
   Z mL=1;
   CPM_MA
   title_=Word("chebyshev");
   ChebyApp app;
      // creates the application class by reading and deploying
      // chebyshev.ini
   app.run();
      // brings the application class to live and action
   CPM_MZ
   return 0;
}
```

5 cpmactionprinciple.h

```
//? cpmactionprinciple.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_ACTION_PRINCIPLE_H_
#define CPM_ACTION_PRINCIPLE_H_
/*
   Studies the mid state variation of the action integral
   by numerical methods. This is intended to complement
   my symbolic studies based on Mathematica
   See mid-state-variation3j.nb of 2006-06-22
*/
#include <cpminifilebasapp.h>

namespace CpmPalaAction{ // devoted to the study of the action principle
//////////////////// class ActionPrinciple ////////////////////////////////
   using CpmApplication::IniFileBasedApplication;
   using CpmRoot::Word;

class ActionAppl: public IniFileBasedApplication { // action application
   // This will deal only with the 3D case and thus allows a simple
   // file structure. All relevant code is in cpmactionprinciple.cpp

public:
   ActionAppl(Word const& w);
   void doTheWork();
};

} // CpmPalaAction

#endif
```

6 cpmactionprinciple.cpp

```
//? cpmactionprinciple.cpp
//? Status of work 2008-10-25.
//?
#include <cpmactionprinciple.h>
#include <cpmlinalg.h>
#include <cpmrfunc.h>
#include <cpmgraph.h>
#include <cpmimage.h>

using namespace CpmRoot;
using namespace CpmArrays;
using namespace CpmGraphics;
using namespace CpmFunctions;
using CpmApplication::IniFileBasedApplication;

using CpmRootX::RecordHandler;

//////////////////// class Action ////////////////////////////////////
class Action{
// independent data
   R3 I_;
     // eigen values of the tensor of inertia
   R3 x0_;
   R3 v0_;
   R3 a0_;
      // initial values of the configuration
      // relative to which all variations will be done

   R3 v_;
      // initial angular velocity in a varied path
   R3 a_;
      // initial angular acceleration in a varied path
   R1 tau_;
   R1 eps_;
      // eps_*tau_ will be used as the integration step
      // for computing the action integral from 0 to
      // 2*tau_
   R3 Iinv_;
      // inverse eigen values of the tensor of inertia
// auxiliary functions
   B mod1_,mod2_;
   R lag1(R3 v)const
   {
      return (I_.x1*v.x1*v.x1+I_.x2*v.x2*v.x2+I_.x3*v.x3*v.x3)*0.5;
   }
   R3 J(R3 x, R3 v)const;
   R3 Jinv(R3 x, R3 v)const;
```

```
   R lag2(R3 x, R3 v)const{ return lag1((-x).ast(v));}
      // faster version
   //R lag2(R3 x, R3 v)const{ return (v|J(x,v))*0.5;}
      // also correct, but a bit slower.

   void ini_();
      // sets a0_ and a_ according of equation (52) of my Granular Matter
      // article. Also sets v_ equal to v0_.
      // To be used in the constructor only.
public:
   R tau()const{ return tau_;}
   R3 x0()const{ return x0_;}
   R3 v0()const{ return v0_;}
   R3 a0()const{ return a0_;}
   void setMod_(bool mod1, bool mod2){mod1_=mod1; mod2_=mod2;}
   Action(RecordHandler const& rch);
   Action(R3 I, R3 x, R3 v, R tau, R eps=1e-2):
      I_(I),x0_(x),v0_(v),tau_(tau),eps_(eps),
      Iinv_(cpminv(I.x1),cpminv(I.x2),cpminv(I.x3)){ini_();}

   X2<R3,R3> pp(R h)const;
      // returns the angular position and angular velocity a time h
      // along the 'parabolic path' given by x0_,v_,a_
   R lag(R h)const
   { X2<R3,R3> pph=pp(h); return lag2(pph.c1(),pph.c2());}
   R S()const;
      // Integral of lag(h) from h=0 to h=2*tau_
   void var_(R3 da);
      // changes a_ to a0_+da and also v_ to v0_-tau_*da.
      // This combined change of a_ and v_ lets pp(2*tau_) remain fixed.
      // As a mutating function, it has a name ending in '_'
};

X2<R3,R3> Action::pp(R h)const
{
   static Word loc("Action::pp(R)");
   R h4=h*0.25;
   R3 sh=v_*2+a_*h;
   R3 shh=sh*h4;
   R fac1=1+(shh|shh);
   cpmassert(fac1!=0,loc);
   R3 v=(v_+(a_+v_.crsPrd(a_)*h4)*h)*(1./fac1);

   R fac2=1-(shh|x0_);
   cpmassert(fac2!=0,loc);
   R3 x=(x0_+shh+shh.crsPrd(x0_))*(1./fac2);
   return X2<R3,R3>(x,v);
}

void Action::var_(R3 da)
```

```
{
   a_=a0_+da;
   v_=v0_-da*tau_;
}

R3 Action::J(R3 x, R3 v)const
{
   R3 e1=x.ast(R3(1,0,0));
   R3 e2=x.ast(R3(0,1,0));
   R3 e3=e1.crsPrd(e2); // saves one evaluation of ast
   return (I_.x1*(e1|v))*e1+(I_.x2*(e2|v))*e2+(I_.x3*(e3|v))*e3;
}

R3 Action::Jinv(R3 x, R3 v)const
{
   R3 e1=x.ast(R3(1,0,0));
   R3 e2=x.ast(R3(0,1,0));
   R3 e3=e1.crsPrd(e2); // saves one evaluation of ast
   return (Iinv_.x1*(e1|v))*e1+(Iinv_.x2*(e2|v))*e2+(Iinv_.x3*(e3|v))*e3;
}

void Action::ini_()
{
   R3 x1= mod1_ ? x0_ :(v0_*tau_).exp().circ(x0_);
   R3 x2= mod2_ ? (v0_*tau_).exp().circ(x0_) : x0_;
   R3 r1=J(x1,v0_);
   R3 r2=r1.crsPrd(v0_);
   a0_=Jinv(x2,r2);
   a_=a0_;
   v_=v0_;

   cpmdebug(x0_);
   cpmdebug(x1);
   cpmdebug(x2);
   cpmdebug(v0_);
   cpmdebug(a0_);
   cpmdebug(tau_);
}

namespace{
R fS(R const& h, Action const& act){ return act.lag(h);}
}// namespace

R Action::S()const
{
   R_Func f(F1<R,Action,R>(*this)(fS));
   return f.antiDer(0,tau_*eps_)(2*tau_);
      // notice that function integral returns the
      // anti-derivative of f (as a function, of course)
}
```

```
Action::Action(RecordHandler const& rch)
{
   Z mL=1;
   Word loc("Action::Action(RecordHandler)");
   CPM_MA
   R3 I,x,v,a;
   R tau,eps;

   Word sec="action data";
   V<R> eigenvalues;
   cpmrh(eigenvalues);
   I.x1=eigenvalues[1];
   I.x2=eigenvalues[2];
   I.x3=eigenvalues[3];

   V<R> pos;
   cpmrh(pos);
   x.x1=pos[1];
   x.x2=pos[2];
   x.x3=pos[3];

   V<R> vel;
   cpmrh(vel);
   v.x1=vel[1];
   v.x2=vel[2];
   v.x3=vel[3];

   cpmrh(tau);
   cpmrh(eps);

   *this=Action(I,x,v,tau,eps);
   CPM_MZ
}

CpmPalaAction::ActionAppl::ActionAppl(Word const& w):
   IniFileBasedApplication(w)
{}

void CpmPalaAction::ActionAppl::doTheWork()
// RecordHandler rch is available for reading data from it by
// means of the cpmrh-macro
{
   Word sec="topic under consideration";
   Word topic;
   cpmrh(topic);
   if (topic=="action_principle"){
      Action act(rch);
      Word sec="run control 1";
      Z i,nTest;
```

```
   cpmrh(nTest);
   V<R> limits;
   cpmrh(limits);
   R3 da(limits[1],limits[2],limits[3]);
   R aScaleRel;
   cpmrh(aScaleRel);
   R aScale=aScaleRel*act.a0().absVal();
   da.normalize_(aScale);
   R_Matrix valS(2,nTest);
   R_Matrix valPos(4,nTest);
   R nInv=1./(nTest-1);
   for (i=1;i<=nTest;++i){
      R frac=2*(i-1)*nInv-1;
      R3 dai=da*frac;
      act.var_(dai);
      R si=act.S();
      R3 posi=act.pp(2*act.tau()).c1();
      valS[1][i]=frac;
      valPos[1][i]=frac;
      valS[2][i]=si;
      valPos[2][i]=posi.x1;
      valPos[3][i]=posi.x2;
      valPos[4][i]=posi.x3;
   }
   Graph g;
   g.setWithOrigin(false);
   g.show(valS);
   cpmwait(10);
 //  act.run();
}
else if (topic=="action_image"){
   Action act(rch);
   Word sec="run control 2";
   using namespace CpmImaging;
   bool mod1,mod2,makeMovie;
   Word movieName;
   Z i,j,k,nx,ny,nz;
   R dsRel, aScaleRel,tWait,stretchFactor=1.1,fracH=1;
   cpmrh(mod1);
   cpmrh(mod2);
   cpmrh(makeMovie);
   cpmrh(movieName);
   cpmrh(fracH);
   act.setMod_(mod1,mod2);
   cpmrh(nx);
   cpmrh(ny);
   cpmrh(nz);
   cpmrh(dsRel);
   cpmrh(aScaleRel);
   cpmrh(stretchFactor);
```

```
      cpmrh(tWait);
      R aScale=aScaleRel*act.a0().absVal();
      R ds=dsRel*act.a0().absVal();
      Z m=ny,n=nx; // matrix style indexing
      Image<R> imr(m,n);
// geometry of scanning
      R3 e3=act.v0();
      e3.normalize_(1.);
      R3 iDir=act.x0().ast(R3(0,0,1));
         // this is the instantaneous location of the inertia axis
         // with the largest momentum
// now we form e2 as the component of iDir which is perpendicular to e3
      R3 e2=iDir-e3*(iDir|e3);
      e2.normalize_(1.);
      R3 e1=e2.crsPrd(e3);
      R3 pk;
      R mr=1./(m-1), nr=1./(n-1), zr=1./(nz-1);
// Generating the image 'best in focus' for setting the brightness range
      for (i=1;i<=m;++i){ // Image<> requires matrix indexing
         R fraci=2*(i-1)*mr-1;
         fraci*=aScale;
         R3 pi=pk+e2*fraci;
         for (j=1;j<=n;++j){
            R fracj=2*(j-1)*nr-1;
            fracj*=aScale;
            R3 pij=pi+e1*fracj;
            act.var_(pij);
            R as=act.S();
            imr.putPix(i,j,as);
         }
      }
      R2 mm=imr.minMax();
      cpmdebug(mm);
      R diff=mm[2]-mm[1];
      cpmdebug(diff);
      Iv iv1(mm[1],mm[2]);
      iv1=iv1.stretch(stretchFactor);
      Iv iv2(0,255);
      Response res(iv1,iv2,1,6);
      Frame frm0;
      Color cTopic(Z(200),Z(100),Z(80));
      frm0.paint(cTopic);
      frm0.display();
      Frame frm;
      R dy=fracH;
      R yC=(1-dy)*0.5;
      R aspRat=(1.*nx)/ny;
      frm.setxy(yC,yC,dy,dy);
      frm.setAspRatHoldHeight(aspRat);
      imr.mark(frm,res,1);
```

```
      frm.setText("cental frame which sets the color code",
         0.1,0.9,Color(Z(255),Z(255),Z(255)));
      frm.display();
      cpmwait(2*tWait+2);
      for (k=1; k<=nz;++k){
         R frack=2*(k-1)*zr-1;
         frack*=ds;
         pk=e3*frack;
// now we have an orthogonal tripod with e3 having the direction of the
// angular velocity.
         for (i=1;i<=m;++i){ // Image<> requires matrix indexing
            R fraci=2*(i-1)*mr-1;
            fraci*=aScale;
            R3 pi=pk+e2*fraci;
            for (j=1;j<=n;++j){
               R fracj=2*(j-1)*nr-1;
               fracj*=aScale;
               R3 pij=pi+e1*fracj;
               act.var_(pij);
               R as=act.S();
               imr.putPix(i,j,as);
            }
         }
         mm=imr.minMax();
         cpmdebug(mm);
         diff=mm[2]-mm[1];
         cpmdebug(diff);

         imr.mark(frm,res,1);
         frm.setText("frame "&cpm(k));
         frm.display();
         if (makeMovie) frm.write(movieName&"_"&cpm(k,2));
         cpmwait(tWait);
      }
   }
   else{ // may be extended
      cpmmessage("nothing to do");
   }
}
```

7 cpmalgorithms.h

```
//? cpmalgorithms.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_ALGORITHMS_H
#define CPM_ALGORITHMS_H
/*
   Description: This namespace will be populated by functions which
    express algorithms of general interest, which don't fit well into
    one of the classes already introduced.
*/
#include <cpmx.h>
#include <cpmvo.h>
#include <cpminterval.h>
#include <cpmf.h>

namespace CpmAlgorithms{

   using namespace CpmStd;

   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmRoot::N;
   using CpmRoot::Word;
   using CpmRootX::Z1;
   using CpmRootX::B;
   using CpmArrays::V;
   using CpmArrays::VV;
   using CpmArrays::Vo;
   using CpmArrays::X2;
   using CpmArrays::X3;
   using CpmFunctions::FncObj;
   using CpmFunctions::F;
   using CpmFunctions::F2;
   using CpmFunctions::F3;
   using CpmGeo::Iv;
   using CpmLinAlg::Z2;

//////////// functions gcd, lcm, lesDiv, prmFac, isPrm /////////////////
// Division properties of natural numbers
   N gcd(N const& a, N const& b, bool verify=false);
      //! greatest common divisor
      // No assumption concerning the arguments. If a or b is 0,
      // we return 0, which never occurs for meaningful arguments.
      // If verify==true, it is tested whether the result actually
      // divides a and b; if not an error is issued.
   N lcm(N const& a, N const& b);
      //! least common multiple
```

```
      // Implementation based on a*b=gcd(a,b)*lcm(a,b) and
      // ise careful in avoiding temporaries that are larger
      // than the result. If any of the arguments is 0, we return 0.
   N lesDiv(N const& n);
      //: least divisor
      // returns the least (true, i.e. !=1) divisor of n.
   V<N> prmFac(N const& n, bool verify=false);
      //: prime factors
      // Returns the prime factors of n as an array in
      // increasing order. If verify==true, it is tested whether the
      // components of the result actually multiply to n;
      // if not an error is issued.
   bool isPrm(N const& n);
      //: is prime
      // returns true iff n is prime. Although the algoritm is trivial
      // it takes less than 10 s to write a list of primes from 2
      // to 99991 (the range covered in Abramowitz, Stegun pp. 870-873)
      // to file.

/////////// functions ZfromBits, bitsFromZ, next_1 ///////////////////////

Z ZfromBits(const V<bool>& vb);
   // for n:=vb.dim() we return
   // sum(i=1,...n) vb[i]*2^(n-i)
   // i.e. the positive integer the binary expansion is given by
   // vb where vb[1] is the most significant bit
   // and vb[n] the least significant one

V<bool> bitsFromZ(Z n, Z p);
   // we assume and test 0<=n<=(2^p)-1 and return vb such that
   // ZfromBits(vb)==n; 2002-10-02: tested for (n,p)=(673,10),(972,10),
   // (0,12),(4095,12) always OK

bool next_1(std::vector<bool>& v);
   //: next
   // Notice that the trailing '_1' says that the argument
   // gets changed by a function call.
   // After the call, v is the lexicographic successor of
   // the input v (order is false < true, and as successor
   // only lists are considered which have the same number
   // of components as v. If in v already all components
   // have the value true, there is no successsor and
   // the return value is false (i.e. no success in finding
   // a successor). Otherwise, the return value is true.
   // If v.size()==0 nothing is done on v and 'false' is returned.
   // I use std::vector here since I need it in a context were
   // high efficiency is desirable. Since the template parameter
   // <bool> is a very light one, the reference counting advantage
   // of Vl or V does not exist here.
```

```
///////////////// class LatticeEnumerator ////////////////////////////

class LatticeEnumerator{ // enumerating the points in cubic lattices
   // consider the lattice {1,2,...b[1]}x{1,2,...b[2]}x...x{1,2,...b[n]}.
   // the class provides a way to enumerate the points and to
   // construct a (i[1],...i[n]) from the enumerating integer.

// indpendent data
   V<Z> b;
// dependent data
   Z n; // n=b.dim()
   V<Z> c;
      // c.dim()=n
      // example n=4
      // c[1]=b[2]*b[3]*b[4]
      // c[2]=     b[3]*b[4]
      // c[3]=          b[4]
      // c[4]= 1

   Z nLattice; // c[1]*b[1]=b[1]*b[2]*...*b[n]
   void update();
      // initializes dependent data
   typedef LatticeEnumerator Type;
public:
   CPM_IO
   LatticeEnumerator(const V<Z>& bounds):b(bounds){update();}
      // obvious
   LatticeEnumerator(){}
   Z operator()(const V<Z>& x)const;
      // We assume (and check) x.dim()==n
      // 1<=x[i]<=b[i] for all 1<=i<=n.
      // for given bounds, there are b[1]*...*b[n]=nLattice
      // points x which satisfy these conditions. The function
      // provides a suitable enumeration of these points by
      // assigning the enumeration number to x as a function value.
   V<Z> operator()(Z k)const;
      // inverse mapping of the previous one
};

///////////////// class LatticeBox ////////////////////////////////////

class LatticeBox{
 // putting points into cells defined by a cartesian product of intervals

   V<Iv> viv;
   V<Z> vz;
   Z n;
   Z np;
   V<R> val;
   V< V<R> > centers;
```

```
   LatticeEnumerator le;
public:
   LatticeBox(const V<Iv>& vivIn, const V<Z>& vzIn);
   V< V<R> > points()const{ return centers;}
   V<R> values()const{ return val;}
   void set(const V<R>& pos, R valIn);
   Z dim()const{ return n;}
   Z nPoints()const{ return np;}
   VV<R> matrix()const;
      // values of the 1-2-plane in matrix-indexed arrangement
};

///////////// bracketing a root ////////////////////////////////////////

X3<R,R,Z> brack(const F<R,R>& f, R xMin, R xMax, R dxMin, R dxMax);
   // let f(x) be defined for xMin<=x<=xMax. We try to find x1 and x2
   // in [xMin,xMax] such that
   // (i) x1<x2 and sign(f(x1))!=sign(f(x2)) (then a root x0, f(x0)==0,
   //    is said to be bracketed, x1<=x0<=x2, see Press et al. p. 350)
   // and
   // (ii) x1 and x2-x1 as small as possible
   // The return values x1:=brack.c1(), x2:=brack.c2()  are guaranteed to
   // have property (i) if found>0 (found=brack.c3()) after call.
   // For found==0 the method found no indication for a sign change
   // of the function in the range [xMin,xMax].
   // For found==1, we got directly a root which is returned both as
   // x1 and x2.
   // For found=2, we found two bracketing points (normal behavior).
   //
   // dxMax is the step size of a first scan. The method ends with
   // found==0 if in this scan (testing a eqidistant chain of points)
   // no sign change and no  trend change (sign change of first
   // differences) was found. In case of detecting a trend change, we
   // performe a refined local scan of the interval which showed the
   // trend change. For scan length < dxMin no further refinement will
   // be done and a mere trend change will not cause further action.
   // Only sign changes will be considered.
   // Notice: We assume xMin<xMan, dxMin<dxMax, otherwise the behavior
   // is undefined.

X2<R,bool> root(const F<R,R>& f, R xMin, R xMax, R dxMin, R dxMax,
      bool silent=false);
   // Let the return value be written as (x,b).
   // For b=true, x is a root: f(x) close to 0.
   // For b=false, no root was found and x is the last value of x that
   // was treated. The arguments xMin,...dxMax are used as arguments in
   // an initial call to function brak. The final localization of the
   // root is done by bisection.

X2<R,bool> rootFast(const F<R,R>& f, R xMin, R xMax, R dx, R eps);
```

```
   // An experimental method to find a root of f in the interval
   // [xMin,xMax]. One starts investigating f at xMin and tries to
   // find the root closest to xMin (just as in ssqe). One does
   // successive parabolic interpolation. In order no avoid
   // instabilities we go not completely to the extrapolated position.
   // In the intended application to visualization of extended
   // bodies it did not result in a speed improvement. For details
   // see the code.

//////////// special solution of quadratic equation ////////////////////

R ssqe(R a, R b, R c, R xMin, R xMax);
// special solution of quadratic equation: returns the smallest
// solution x of the equation a*x*x+b*x+c=0 that satisfies xMin <= x <=
// xMax.
// If no such solution exists in this range, -1 is returned
// We assume xMin < xMax (otherwise behavior is not specified).
// This is a rather optimized algorithm which (for instance) helps to
// find effectively the point where a light ray hits a sufficiently
// regular body.

//////////////// solution of Kepler's equation (for 0<=eps<1) ////////////
R solKepEqu(R M, R e, R acc=1e-8);
//: Solution (of) Kepler's  equation.
// We return the solution E of the famous transcendental equation
//    E = M + eps * sin E
// In astronomical nomenclature M is the 'mean anamoly'
// E the 'excentric anomaly, and eps the numerical excentricity.

////////////////////  bisection ////////////////////////////////////////

   X2<R,R> bisection(const F<R,bool>& f, R xF, R xT,
      R xacc=1e-3, Z iterMax=40 );
      // Finding a boundary of a 1-dimensional domain characterized by an
      // 'indicator function'.
      // If f(xF)=false, f(xT)=true, xacc>0,  we return a pair (xf,xt)
      // such that
      // f(xf)= false and f(xt)=true where xt-xf = (xT-xF)/(2^n)
      // for some natural number n and |xt-xf|<xacc.
      // (Notice: if xF<xT then xt<xf; if xF>xT then xf>xt)
      // An unbiased guess for the boundary itself then is 0.5*(xf+xt).
      // The number of iterations is limited to iterMax, if the limit is
      // reached a warning is issued and the values reached so far are
      // returned (they, then, do not satisfy |xt-xf|<xacc).

   R rootByBisection(const F<R,R>& f, R x1, R x2, R xacc, Z iterMax=40 );
      // Finding the a root of f by bisection, assuming that
      // f(x1) and f(x2) differ in sign. The control of accuracy is just
      // as in the previous function. The return value is 0.5*(x1b+x2b);
      // where (x1b,x2b) resulted from the last bisection step.
```

```
//////////////////////  fixedPoint ////////////////////////////////////////

   template <class X>
   X fixedPoint(const F<X,X>& f, const X& x0,
      R eps=1e-3, Z iterMax=40 );
      // Finding a solution x of x=f(x). Start point for the iterative
      // solution is x0, and the condition for terminating the iteration
      // x_0, x_1, x_2, ... is (x_(n+1)).disVal(x_n)<=eps.
      // We thus assume that class X defines a function X.disVal(X)
      // such that x1.disVal(x2) is small for x1 and x2 being close
      // together. The number of iterations is limited to iterMax, if
      // the limit is reached a warning is issued and the value reached
      // so far is returned. A simple and effective provision against
      // oscillatory behavior is implemented.

// notice that the MS compiler (linker?) expects all template definitions
// to be placed in header files (deviating from the practice suggested by
// Stroustrup)

   template <class X>
   X fixedPoint(const F<X,X>& f, const X& x0,
      R eps, Z iterMax)
   {
      X xOld=x0;
      X xNew=f(xOld);
      Z count=0;
      while(xOld.disVal(xNew)>eps && count++ < iterMax ){
         xOld=xNew;
         X xAux=f(xOld);
         xNew=(f(xAux)+xAux)*0.5;
      }
      if (count>=iterMax)
         cpmwarning("CpmAlgorithms::fixedPoint() exhausted iterMax");
      return xNew;
   }

   R fixedPoint(const F<R,R>& f, const R& x0,
      R eps, Z iterMax);
      // Specialization of the template version; needed since R can't
      // provide member functions (and the legal work-arround with a
      // non-member disVal(X,X) does not work always with the MS
      // compiler.

//////////////////////////  Place ////////////////////////////////////////

   template <class X>
   class Place{//creating X-valued lists, satisfying specified conditions
      F<Z,X> create; // generator for X's
      F<X,bool> condition1; // condition which each x should satisfy
```

```
        // in order to be acceptable for a result list to be created
        // via opertor ()
     F< X2<X,X>, bool> condition2;
        // condition which each pair x[i], x[j]  should satisfy
        // in order to make x[i] and x[j] both be acceptable as
        // members of the result list to be created via opertor ()
     bool useCondition2;
        // If this is false, the loop for testing condition2
        // will be omitted (added 2006-02-21)
    public:
    Word nameOf()const
    { return Word("Place< "&CpmRoot::Name<X>()(X())&" >"); }

    Place(const F<Z,X>& cr,
        const F<X,bool>& cond1,
        const F< X2<X,X>, bool>& cond2):
        create(cr),condition1(cond1),
        condition2(cond2),useCondition2(true){}
        // bringing the condition functions in place, that will be
        // utilized in a call to operator()

     Place(const F<Z,X>& cr,
        const F<X,bool>& cond1):
        create(cr),condition1(cond1),
        condition2(),useCondition2(false){}
        // Faster form of case in which condition2 is the 'always true
        // function'. Also more convenient since one has not to
        // give a definition of the 'always true function'.

    V<X> operator()(Z n, Z trialMax=1000 )const;
     // From the X-valued sequence create(1), create(2), create(3), ....
     // we select a finite sublist x[1],....x[n] such that
     // condition1(x[i])==true for all 1 <= i <= n
     // condition2(x[i],x[j])==true for all 1 <= i,j <= n, i!=j
     // If we would need higher sequence members than create(n+trialMax)
     // the function stops working and returns the array it has obtained
     // so far (this than has less than n components). The strategy is
     // simple: create a new X (via create()) test the conditions, and
     // take a new one if any of the conditions fails.
   };

template <class X>
V<X> CpmAlgorithms::Place<X>::operator()(Z n, Z trialMax )const
{
   Z mL=2;
   Word loc=nameOf()&"::operator()(Z,Z)";
   CPM_MA
   Z i=0,nValid=0;
   V<X> res(n);
   X xi;
```

```
   cpmassert(n>0,loc);
   R nInv=1./n;
   Z iLimit=n+trialMax;
   while (nValid<n){
LABEL:
      if (cpmverbose>mL) cpmprogress("Place-loop",nValid*nInv);
      xi=create(++i);
      if (i>iLimit) break;
      if (condition1(xi)==false) goto LABEL;
      if (useCondition2){
         for (Z j=1;j<=nValid;j++){
            if (condition2(X2<X,X>(res[j],xi))==false) goto LABEL;
         }
      }
      res[++nValid]=xi;
   }
   if (nValid<n){
      cpmwarning(loc&": could create only "&
         cpm(nValid)&" items of "&cpm(n));
   }
   CPM_MZ
   return res.resize(nValid);
}

//////////////////  RandomIntegration ////////////////////////////////

   template <class X>
   class RandomIntegration{
      // calculating expectation values based on a X-valued list
      // A X-valued list is the algorithmic substitute for the notion of
      // a probability measure on X:
      // Let S be a subset of X and let x[i]:=create(i) be the sequence
      // of X's resulting from the creation mechanism of the present
      // class. Then
      //    card{i | 1<=i<=n, x[i] element of S}/n
      // is a guess for (measure of S)/(measure of X).
      // If create() generates a random sequence, this will become
      // independent of n for large n 'in the sense of probabilistic
      // convergence'.
      // All random aspects ly in the function create; the algorithms
      // implemented here are strictly deterministic (and repeatable).
         F<Z,X> create; // generator for X's
         F<X,bool> chi; // this is a indicator function of a subset in X
            // over which all integrations will take place effectively.
            // chi(x)==true means, that x belongs to the 'effective
            // domain of integration'.
         Z norMet_; // normalization method
            // if 1, the identity function on X gets the integral 1
            // if 0, the identity function on X gets the the integral
            // (measure of {x \in X | chi(x)==true})/(measure of X)
```

```
      public:
        RandomIntegration(const F<Z,X>& cr,
           const F<X,bool>& ch, Z norMet=1):
           create(cr),chi(ch),norMet_(norMet){}
            // bringing the functions in place, that will be utilized
            // in a call to operator()

        V<R> operator()( V<F<X,R> > const&, Z n, Z trialMax=1000 )const;
      // For an array of functions the expectation value is calculated.
      // Actually, each of the functions is set to zero outside the set
      // defined by function chi. It is assured that n points falling
      // into this subset are used for calculation. If this set is so
      // small that more than trialMax points from function create() had
      // to be tested, an error is issued. It is essential for efficiency
      // to let one call to this function evaluate the expectation values
      // of a full list of functions: doing these functions in succession
      // would ask for creating the random sequence many times.
   };

template <class X>
V<R> RandomIntegration<X>::operator()
( V<F<X,R> > const& f, Z n, Z trialMax)const
{
   Word loc("V<R> RandomIntegration<X>::operator()");
   Z d=f.dim();
   V<R> res(d,0.);
   Z i=1;
   Z nDone=0;
   Z j;
   while (nDone<n){
      if (i>trialMax)
         cpmerror("Too many trials in RandomIntegration<X>::operator()");
      X x=create(++i);
      bool isValid=chi(x);
      if (norMet_ || isValid){
         nDone++;
      }
      if (isValid){
         for (j=1;j<=d;j++){
            res[j]+=(f[j])(x);
         }
      }
   }
   cpmassert(nDone>0,loc);
   R nInv=1./nDone;
   for (j=1;j<=d;j++) res[j]*=nInv;
   return res;
}
```

```
//////////////////// class AveragingInterpolator ////////////////////

template <class X, class Y>
// we assume R X::distSquare(X) , Y*=R, Y(), Y+=Y
// General interpolation method for vector-valued functions on
// metric spaces. Seems to work well.
// Method was invented (by me) in 1997 as a way to define
// a constructor for Fr<> from discrete data.
class AveragingInterpolator: public FncObj<X,Y>{
// interpolation method for vector-valued functions on metric spaces
   // AveragingInterpolator is a non-mutable class
   const V<X> vx;
   const V<Y> vy;
   const Z ns;
      // if this is different from 0, the ns nearest
      // neighbors are used for forming a weighted mean.
      // This helps to keep the scheme local and to have
      // only to compute a linear combination of ns terms.
      // (normally not a concern). The price is an vector ordering
      // operation (to be done for each evaluation of f)
   static Y avrIpl(const X& x, const V<X>& vx, const V<Y>& vy,
      const Z& nn);
   // algorithmic content of the class as a static function.
   // This seems to be the most direct way of injecting complicated
   // algorithms into my F<X,Y> class
public:
   AveragingInterpolator(const V<X>& x, const V<Y>& y, Z n=0);
      // default ns==0 guarantees a continuous interpolation
      // function. Be cautious when using a different value
   Y operator()(const X& x)const{ return avrIpl(x,vx,vy,ns);}
};

template <class X, class Y>
Y AveragingInterpolator<X,Y>::avrIpl(const X& x,
   const V<X>& vx, const V<Y>& vy, const Z& nn)
{
  // const R p=-1.75;
    const R p=-1.75;
   // p==-2 and smaller (sic, not in absolute value) gives something
   // close to step function interpolation (flat terrasses connected
   // by steep ridges). Has to be negative in order to give near points
   // heavy weight

   Z mL=4;
   Word loc("AveragingInterpolator<>::avrIpl(...)");
   CPM_MA
   Z i,n=vx.dim(),ny=vy.dim();
   cpmassert(n==ny,loc);
   if (n==0) return Y();
```

```
   R val,sum=0.;
   Y res=Y();
   if (nn==0 || nn>n){ // weighted mean over all points
      for (i=1;i<=n;i++){
         R r2=x.distSquare(vx[i]); // distSquare more efficient than
         // distance, could be important for large n
         if (r2==0)  return vy[i];
         val=::pow(r2,p); // val ~ 1/(r^2p) for application to points
            // in 3-space the power of r^2/r^2p should be <1 for
            // so that integral from 1 to infinity of r^2/r^2p is
            // finite
         sum+=val;
         res+=(vy[i]*val);
      }
   }
   else{ // if the vy[i] are 'heavy' objects for which
         // computing linear combinations is expensive,
         // this method which works with short linear combinations at
         // the price of having to order an vector might be better
      cpmassert(nn>0,loc);
      Vo<R> vr2(n);
      for (i=1;i<=n;i++) vr2[i]=x.distSquare(vx[i]);
      V<Z> per=vr2.permutationForIncreasingOrder();
         // finding the nearest neighbors
      for (i=1;i<=nn;i++){  // short sum over nearest neighbors
         Z j=per[i];
         R r2=vr2[j];
         if (r2==0) return vy[j];
         val=::pow(r2,p);
         sum+=val;
         res+=(vy[j]*val);
      }
   }
   cpmassert(sum>0.,loc);
   R sumInv=1./sum;
   res*=sumInv;
   CPM_MZ
   return res;
}

template <class X, class Y>
AveragingInterpolator<X,Y>::AveragingInterpolator
(const V<X>& x, const V<Y>& y, Z n):vx(x),vy(y),ns(n)
{
   Z mL=3;
   Word loc("AveragingInterpolator(...)");
   CPM_MA
   cpmassert(vx.dim()<=vy.dim(),loc);
   CPM_MZ
}
```

```
//////////////////////// Quantizer ////////////////////////////////////

   // Putting real numbers into boxes defined by subintervals of an
   // interval.

   // The interval [xL,xU] is divided into n equally sized disjoint
   // boxes.
   // These are numbered from 1 to n in the order of increasing x.
   // If x belongs to one of the boxes, we return the number of it. If x
   // belongs to the interval [xL,xU] it always belongs to exactly one
   // box by the definition that an x falling exactly on 'an internal
   // separation line' between boxes, belongs to the box with the lower
   // index. For x<xL we return 0 and for x>xU we return n+1.
   // The degenerate case is xL=xU. Here we return 1 for x=xL ( and,
   // again, n+1 for x>xU). The previous explanation describes the first
   // version of Quantizer considered. Meanwhile a second constructor,
   // taking as an argument a const V<R>& v, provides more flexible
   // quantisation services. Consider the set X of R's provided by the
   // components of v. (this may be void since v.dim() is allowed
   // to be 0. Notice card(X)<=v.dim().
   // Set card(X)=:n+1. We order the elements of X and get n ordered
   // boxes. Quantizing x means to find the index of the box to which it
   // belongs. Again we return 0 for x below every box and n+1(=card(X))
   // above every box.
   // For card(X)=0, it is thus consistent to return always 0, indicating
   // that every x is below every box and above every box.

   // This functionality of quantizing is here put into a class
   // (instead of a function) since this allows to separate the pre-
   // computations (which get done after constructor call) and the
   // actual quantization (which get done after call of operator()).
   // This could increase speed in a situation in which many numbers
   // have to be quantized with respect to one system of values xL,xU,n.

class Quantizer: public F<R,Z>{ // Putting real numbers into boxes
      // this derivation defines Z operator()(const R&)const and makes
      // available all member functions of F<R,Z> and all functions that
      // take F<R,Z>& as arguments. Notice that the only data member is
      // an address of a memory area that stores an instance of a class
      // that is known to be derived from FncObj<R,Z>. It thus defines Z
      // operator()(const R&)const; the actual definition of this virtual
      // operator may utilize data of types that vary from one value of
      // the adress to the next. So, the class itself does not determine
      // the information layout of its internal state. Very advanced
      // construct, indeed. Notice that this is an immutable class that
      // satisfies the value interface.

   Quantizer(const F<R,Z>& f):F<R,Z>(f){}
      // private tool for defining public constructors
```

```
public:
   Quantizer();
      // trivial round-Off, needs no data!
   Quantizer(R xL, R xU, Z n);
      // setting the limits of the interval and the number of boxes.
      // If n is <1, the actual number of boxes will be 1.
      // xL, xU needs not to be ordered on input.
   Quantizer(Iv, Z n);

   Quantizer(const V<R>& x, bool ordered=false);
      // here the limits of the boxes are obtained from an ordered and
      // unified version (multiple appearance of a component in x is
      // changed into single appearance) of the array x. Here the boxes
      // need not be of equal size. If there are no boxes (x.dim()==0),
      // operator() will always return 0, which is by far the most
      // logical prescription (see above). If the second argument is
      // true,
      // we assume that x is already ordered and unified.
   F<R,Z> toBase()const{ return *this;}
};

//////////////////////// IntQuant ////////////////////////////////////

   // Putting integer numbers into boxes defined by subintervals of an
   // integer interval.

   // The integer interval [xL,xU] i.e. the set {xL,xL+1,...xU}
   // is divided into n approximately equally sized disjoint
   // subintervals (equally sized they can be only if xU-xL+1
   // is a multiple of n)
   // These are numbered from 1 to n in the order of increasing x.
   // If x belongs to one of the boxes, we return the number of it. If x
   // belongs to the interval [xL,xU] it always belongs to exactly one
   // box by the definition.
   // For x<xL we return 1 and for x>xU we return n.
   // The degenerate case is xL=xU. Here we return 1 always.
   // Here the class has its own data and a function among them.
   // This seems to be the most powerful type of implementation.
   // The main point is here that in defining the constructor
   // one may pre-process the constructor arguments before the
   // data member f has to be initialized. Initializing f is
   // simply by assignment. Notice that it would be contraproductive
   // to derive InQuant from FuObj<Z,Z> to get operator()(Z).
   // This would make the result an immutable class wheras the
   // present definition gives a class that implements the strict
   // value interface.

class IntQuant{ // Putting integer numbers into boxes

   Z zL;
```

```
   Z zU;
   Z nP;
   R spanInv;
   F<Z,Z> f;

public:
   IntQuant(Z xL=1, Z xU=1, Z n=1);
      // setting the limits of the interval and the number of boxes.
      // If n is <1, the actual number of boxes will be 1.
      // xL, xU needs not to be ordered on input.
      // For xL==xU, operator()(Z) will always return 1
   Z operator()(Z i)const{ return f(i);}
   F<Z,Z> toF()const{ return f;}
};

//////////////////// Percentilizer ////////////////////////////////////
// Consider an array a[1],...., a[n] of real numbers. Here we consider
// this as a distribution of the a's over the real axis.
// An object Percentilizer(a) defines characteristic quantities
// of this distribution.
// We have the minimum, the median and the maximum as
// the most obvious landmarkes, and the quartiles (hinges) as similarly
// built constructs ( see Weisstein: CRC Concise Encyclopedia of
// Mathematics p. 844 for hinge).
// All these are special cases of percentiles, i.e. just the numbers
// which the 'Percentilizer' will return via operator(). Consider:
// V<R> a=...;
// Percentilizer per(a);
// R x=0.25;
// R y=per(x);
// y is a number such that 25% of all components of a have
// values below y. An important application of Percentilizer is to
// exclude from a set of numbers a 'few extreme values' that may be
// considered as irregular.

class Percentilizer: public F<R,R>{ // handling quartiles (hinges)

      // Notice that this is an immutable class that does not exactly
      // satisfies the value interface since it does not define a default
      // constructor. Such a constructor could not be of any value.

   Vo<R> arr;
      // ordered list of numbers that define a statistical distribution;
      // In contrast to class Quantizer the basic data elements are not
      // held by the base class. So it is readily available for
      // efficient implementation of further methods needing it.

 public:
   Percentilizer(const V<R>& );
      // setting the list of real values (no ordering implied) that
```

```
      // define the distribution, which in turns defines the quantiles

   F<R,R> toBase()const{ return *this;}
      // evident

   R mean(const Iv& iv)const;
      // iv is a subinterval of Iv(0,1) (or Iv(0,1) itself).
      // This selects the components of arr which lie
      // between (*this)(iv[1]) and (*this)(iv[2]). The mean value
      // of these selected components is returned.

   R modulation( const Iv& ivL, const Iv& ivU)const
      // a flexible descriptor of the variability (deviation from
      // constancy) of the values in arr. Allows to ignore
      // extreme (potentially irregular) values in arr.
      // Details clear from code:
      { return (mean(ivU)-mean(ivL))*cpminv(mean(ivU|ivL));}

   V<R> select(const Iv& iv)const;
      // returns the sub-list of arr over which function mean
      // performs the average
};

R gaussRandomValue(const R& mean, const R& sigma);
// Returns normally distributed random numbers with mean and sigma
// given by the respective arguments.
// Modified from gasdev(long*) of Press et al. by introducing another
// basic random generator and by not specializing the arguments.
// Notice that the involved method of generation will not easily enable
// to define a random array (i.e. introduction of the argument j as in
// ranVal(const R& x, Nd j) such that consequtive components are
// independent.

//////////////// some counter-like data types ////////////////////////

////////////////////// PeriodicAlarm ////////////////////////////////

// Let per be an instance of class PeriodicAlarm. Then for some t of type
// R we may ask for the value of per(t) which is either true or false.
// This boolean return value does not only depend on t but also on the
// values of the t's for which per was called earlier. The situation for
// which PeriodicAlarm is designed and in which its functionality follows
// an easy to describe pattern is that calls to per(t) come in order of
// increasing t. This means that a code fragment
//
//    PeriodicAlarm per(...);
//    R t1=...;
//    bool b1=per(t1);
//       ...
//    R t2=...;
```

```
//    bool b2=per(t2);
//       ...
// should ensure that t2>=t1.

// Let the tasks be 'write the state variables of a dynamical system to
// disk' be scheduled for instants of time which form an equidistant
// lattice of width period (period>0) and let t be the time variable of
// the dynamical system that changes from one computational step to the
// next by varying amounts (adaptive time stepping) (normally small
// compared to the fixed step 'period' from task to task).
// Then per(t)==true says that we should call writeState().

class PeriodicAlarm{ // periodic counter
   bool active;
      // if false, the counter is deactivated so that calling operator()
      // gives always 'false' and no further action is taken
   R period;
      // distance between points of alarm
   R lastAlarm;
      // t walue for which (*this)(t) was found 'true' the last time
   Z counter;
      // counts the alarms
   R nextLatticePoint;
      // the time points for alarm form a equidistant lattice (chain)
      // of points on the t-axis. Thus the name
   R tol;
      // tolerance as fraction of period, so that not
      // the condition t>=nextLatticePoint triggers
      // operator()(t)==true but already
      // t>=nextLatticePoint-tol. We always assume period>=0.
      // In the frequently occurring case that t grows in steps of
      // period between calls of operator() this makes the trigger
      // condition independent of numerical chance events.
public:
   PeriodicAlarm(R period=0, R tIni=0, R phase=0.);
      // time for the first alarm is tIni+phase*period
   bool operator()(R t);
      // gives the alarm status for time t. Is 'true' for
      // (t+tol) >= nextLatticePoint
      // updates nextLatticePoint by adding the smallest integer multiple
      // of period such that t<nextLatticePoint
   Z count()const{ return counter;}
      // returns the number of calls to operator() with a true result.
      // More precisely, a call can become counted with a value larger
      // than one if t-tLast is larger than period.
   R getLastAlarm()const{ return lastAlarm;}
   void setCounter(Z c){counter=c;}
   void switchOff(){ active=false;}
      // if this is the case, always false==(*this)(t)
   void switchOn(){ active=true;}
```

```
   R per()const{ return period;}
};

//////////////////////////// SingleAlarm ////////////////////////////////////

// alarm is issued only once

class SingleAlarm{ // 1-step counter
   bool done;
   R alarmPoint;

public:
   SingleAlarm(R tAlarm=0):alarmPoint(tAlarm),done(false){}
   bool operator()(R t);
      // gives the alarm status for time t. Is 'true' for t>=alarmPoint
      // sets done=true
};

//////////////////////////// CyclicAlarm ////////////////////////////////////

// version of a perodic alarm with a discrete periodic counter. Each
// call of the status (operator()) updates the counter

class CyclicAlarm{ // periodic counter

// notice that all attributes are class typed and thus initialized

   B dormant;
      // if this is true, the object does not react on calls
      // to operator(), especially the value of 'alarm'
      // remains unchanged

   B done;
      // if 'true' the one has alarm==false
      // and operator() always returns false

   Z1 cog; // 'Zahn','Zahnrad' in German
      // each call to operator() cyclically increases cog
      // cyclic structure given by period.
      // One should have available residue class Z/n at this place

   Z1 period;
      // after period increments of cog we set alarm==true

   Z1 counter;
      // each setting alarm==true increases counter

   Z1 counterMax;
      // after counter has reached the value counterMax
      // we have done==true
```

```
      // since counter>=0 always, a negative valur for
      // counterMax let it never be done (which is considered
      // normal behavior: 'work is never done'.

   B alarm;
      // if this is true, alarm is set

public:
   typedef CyclicAlarm Type;
   virtual Type* clone()const{ return new CyclicAlarm(*this);}
   CyclicAlarm(Z per=1, Z cogIni=0, Z maxAlarms=-1);
   // per=0 gives return value false for all calls of
   // operator().
   // Effectively cogIni<period (since enforced).
   // Negative values of cogIni create an initial latency
   // of -cogIni calls to step or operator().
   virtual void step(void);
      // moves *this to the next state
   bool operator()(void)
   {
      if (period<1) return false;
      else{bool alNow=alarm; step(); return alNow;}
   }
      // most convenient usage of the class e.g. in the implementation
      // of function 'record'. Returns the status of alarm at present
      // state and moves *this to the next state
   Z count()const{ return counter;}
   void setAlarm(bool b){ alarm=B(b);}
   bool getAlarm()const{ return alarm;}
   void switchOff(){ dormant=true;}
   void switchOn(){ dormant=false;}
   bool ready()const{ return done;}
   void restart(){ done=false; cog=0, counter=0; alarm=true;}
   void restartWithNewPeriod(Z per){ period=per; done=false;
      cog=0, counter=0; alarm=true;}
   V<B> record(Z n);
   Z per()const{ return period;}
      // returns the sequence resulting from n-fold application of
      // operator(). Means for testing.
};

//////////////////// NonCyclicAlarm ////////////////////////////////

// cog-based state machine that gives alarm iff
// cog==points_[counter]. Each alarm increases not only
// cog but also counter.
//
// The behavior is thus just the following:
// V<Z> vz=...;
// NonCyclicAlarm nca(vz);
```

```
// bool b;
// b=nca();
//    ...
// b=nca();
// ...
// b=nca(); // n-th call of operator()() after constructor call
// results in b==true iff there if a k \in {1,2,..., vz.dim()} such that
// n==vz[k]

class NonCyclicAlarm{ // cog-based state machine

// notice that all attributes are class typed and thus initialized

   B done;
      // if 'true' the one has alarm==false
      // and operator() always returns false

   Z1 cog; // 'Zahn','Zahnrad' in German
      // each call to operator() increases cog by 1

   Z1 counter;
      // each setting alarm==true increases counter

   V<Z> points_;
      // list of values such that cog==points_[counter]
      // is the point to trigger

   Z1 counterMax;
      // after counter has reached the value counterMax
      // we have done==true

   B alarm;
      // if this is true, alarm is set

public:
   typedef NonCyclicAlarm Type;
   virtual Type* clone()const{ return new NonCyclicAlarm(*this);}
   NonCyclicAlarm(const V<Z>& points=V<Z>(0));
      // condition for alarm==true is cog==points[counter]
      // alarm==true implies counter->counter+1
      // Der Bedeutung von 'points' entsprechend sollte dieses
      // eine strict monoton wachsende Liste sein. points_
      // wird aus points durch umordnen und Weglassen von
      // Doubletten erhalten. Komponenten von points_, die
      // kleiner als 1 sind, werden nicht eliminiert, führen
      // aber dazu daß immer alarm==false gilt.
   virtual void step(void);
      // moves *this to the next state: cog->cog+1
   bool operator()(void){ bool alNow=alarm; step(); return alNow;}
      // most convenient usage of the class.
```

```
      // Returns the status of alarm at present
      // state and moves *this to the next state
   Z count()const{ return counter;}
   bool getAlarm()const{ return alarm;}
   bool ready()const{ return done;}
   void restart(){ done=false; cog=0, counter=1; alarm=false;}
   Z last()const{ return points_.last();}
};

//////////////////////////// BurstAlarm ////////////////////////////////////

// Similar to Cyclic alarm but opertor() gives true for if the
// call number belongs to to a doubly periodic lattice
// 2004-05-11: Triggers in some cases also if per=0. This is not
// intended. Not completely understood!. Better don't use it!

class BurstAlarm: public CyclicAlarm{ // double periodic counter

  CyclicAlarm burst;

public:
   typedef BurstAlarm Type;
   typedef CyclicAlarm Base;
   virtual CyclicAlarm* clone()const{ return new BurstAlarm(*this);}

   BurstAlarm(Z per=1, Z burstPer=0, Z burstLen=0);
      // We exemplify the meaning of the data by considering the
      // sequence created by function record (1=true, 0=false)
      // e.g. for
      //       per=10,  burstPer=2,    burstLen=3:
      // | 1 0 | 1 0 | 1 0 | 0 0 0 0 | 1 0 | 1 0 | 1 0 | 0 0 0 0
      // | 1 0 | 1 0 | 1 0 | 0 0 0 0 ...
      // one Burst is | 1 0 | 1 0 | 1 0 |. Period from burst beginning
      // to burst beginning is 10
   void step(void);
      // redefined
};

////////////////////////////////// enum Switch ////////////////////////////

enum Switch{ //  to be used as an argument type in cases where the values
      // 'true' and 'false' of a boolean argument would not suggest the
      // meaning with sufficient clarity
   ON,
   OFF
};

//////////////////////////////// getNerFrc_1(...) ////////////////////////////

Z getNerFrc_1(R& x, R xL);
```

```
   //: get nearest fraction
   // Notice that '_1' indicates that the first argument may be changed by
   // by a function call. Assumed is x<=xL and x!=0.
   // Consider  S(xL):={ y: there is n \in { 1,2,...} such that y*n=xL}
   // The function outputs the n and changes x into
   // x0 \in S(xL) such that |x0-x|=inf{ |s-x| : s \in S(xL)}.

//////////////////////////////// genSqrRoot(Z) ////////////////////////////

Z2 genSqrRoot(Z n);
//: generalized square root
// The square root in R solves the problem to  write a number x as a
// product of two factors which are equal.
// If we consider only natural numbers, a factorization with just
// this properties is possible only for the square numbers
// (1,4,9,16,...). It remains a meaningful problem to find two
// factors which are 'differ as little as possible' and represent
// x as a product 'as accurate as possible'. It is this analogy
// to the square root to which the name refers, the generalization
// thus also comprises a specialization --- the specialization to
// integer or natural numbers.
// Let (z1,z2) be the return value of the function. Then z1*z2>=n, z2>=z1.
// Among the infinite number of solutions of these inequalities
// we choose one for which d1:=z1*z2-n and d2:=z2-z1 are both as small as
// they together can be. For instance, if n is a square number n=m*m, then
// z1=z2=m are oviously what we want since then d1=d2=0.
// For n=7 we have for z1=1, z2=7 the desirable property d1=0, but
// d2=7-1=6 is not small. Here we could consider the pairs  (2,4), (3,3).
// In the first case we have d1=8-7=1, d2=4-2=2, and
// in the second case d1=9-7=2, d2=3-3=0.
// d1+d2 is smaller in the second case and so (3,3) will be selected.
// See the implementation in cpmalgorithms.cpp for details.
// Defining this function was suggested by a graphical problem. Given
// n scalable figures; how to layout them on screen if only
// matrix subdivisions of the screen are under consideration.

} // namespace

#endif
```

8 cpmalgorithms.cpp

```
//? cpmalgorithms.cpp
//? Status of work 2008-10-25.
//?
// notice that some algorithms are templates and are defined not here
// but in cpmalgorithms.h !!!!!!!!!!!!!!!!!!!!

#include <cpmva.h>
#include <cpmalgorithms.h>
#include <cpmmacros.h>

   // the names from CpmRootX will be used via using declarations in
   // function blocks.

using namespace std;

using CpmRoot::R;
using CpmRoot::Z;
using CpmRoot::N;
using CpmRoot::Word;
using CpmRootX::B;
using CpmArrays::V;
using CpmArrays::VV;
using CpmArrays::Vo;
using CpmArrays::Va;
using CpmArrays::X2;
using CpmArrays::X3;
using CpmFunctions::F;
using CpmFunctions::F1;
using CpmFunctions::F2;
using CpmFunctions::F3;
using CpmFunctions::F4;
using CpmFunctions::FncObj;
using CpmGeo::Iv;
using CpmAlgorithms::LatticeBox;
using CpmLinAlg::Z2;
   // these using declarations are active till the end of this file, also
   // within the anonymus namespace to be defined later.
   // The directive #include <cpmalgorithms.h> brings also some using
   // declarations in action. These are only valid within the namespace
   // CpmAlgorithms and  n o t  in the anonymus namespace mentioned
   // above.
   // They are also not valid as return values of CpmAlgorithms member
   // functions (only in argument lists and definition blocks)


N CpmAlgorithms::lesDiv(N const& n)
{
```

```
   if (n<=2) return n;
   N res=2;
   while (n%res) res++;
   return res;
}

V<N> CpmAlgorithms::prmFac(N const& n, bool verify)
{
   V<N> res;
   if (n<=2) res=V<N>(1,n);
   else{
      N na=n;
      N f=CpmAlgorithms::lesDiv(na);
      res<<f;
      while(f<na){
         na/=f;
         f=CpmAlgorithms::lesDiv(na);
         res<<f;
      }
   }
   if (verify){
      N nt=1;
      for (Z i=res.b();i<=res.e();++i) nt*=res[i];
      if (nt!=n) cpmerror("CpmAlgorithms::prmFac("&cpm(n)&") failed");
   }
   return res;
}

N CpmAlgorithms::gcd(N const& n1, N const& n2, bool verify)
{
   N a,b;
   if (n1>=n2){ a=n1; b=n2; }
   else{ a=n2; b=n1;}
   if (b==0) return 0;
   N r=a%b;
   while(r){
      a=b;
      b=r;
      r=a%b;
   }
   if (verify){
      if (n1%b || n2%b){
         cpmerror("CpmAlgorithms::gcd("&cpm(n1)&","&cpm(n2)&") failed");
      }
   }
   return b;
}

N CpmAlgorithms::lcm(N const& a, N const& b)
{
```

```
   N g=CpmAlgorithms::gcd(a,b);
   return g==0 ? 0 : ((a/g)*(b/g))*g;
   // avoids forming temporaries that are larger than the result
}

bool CpmAlgorithms::isPrm(N const& n)
{
   if (n<2) return false;
   return CpmAlgorithms::lesDiv(n)==n;
}

namespace{
   Z pow2(Z n)
   {
      if (n<0) cpmerror("pow2(Z): argument <0");
      Z res=1;
      for (Z i=1;i<=n;i++) res*=2;
      return res;
   }
}

Z CpmAlgorithms::ZfromBits(const V<bool>& vb)
   // res=sum i=1,...vb.dim vb[i]*2^(i-1)
{
   Z mL=3;
   Word loc("CpmAlgorithms::ZfromBits(V<bool>)");
   cpmmessage(mL,loc&" started");
   Z n=vb.dim();
   // we need 2^(n-1) as the first sepatator
   cpmassert(n>0,loc);
   Z d=pow2(n-1);
   Z res=0;
   for (Z i=1;i<=n;i++){
      if (vb[i]) res+=d;
      d/=2;
   }
   cpmmessage(mL,loc&" done");
   return res;
}

V<bool> CpmAlgorithms::bitsFromZ(Z n, Z p)
   // we assume and test 0<=n<=2^(p-1) and return vb such that
   // ZfromBits(vb)==n
{
   Z mL=3;
   Word loc("CpmAlgorithms::bitsFromZ(Z n, Z p)");
   cpmmessage(mL,loc&" started");

   cpmassert(n>=0,loc);
   cpmassert(p>0,loc);
```

```
   cpmassert(p<32,loc);

   Z d=pow2(p);
      // e.g. p=8, d=256
   Z limit=d-1; // 255
   cpmassert(n<=limit,loc);
   V<bool> res(p,false);
   Z separator=d/2; // 128
   d=separator; // 128
   for (Z i=1;i<=p;i++){
      d/=2; // 64
      if (n<separator){ //x<=127
         separator-=d;
      }
      else{
         separator+=d;
         res[i]=true;
      }
   }
   cpmmessage(mL,loc&" done");
   return res;
}

bool CpmAlgorithms::next_1(std::vector<bool>& v)
// should be very efficient
{
   using std::vector;
   size_t n=v.size();
   if (n==0) return false;
   size_t i=n-1;
   while (v[i]==true){
      --i;
      if (i>n || i<0 ) return false; // then all entries were true
      // notice that size_t may not have negative values
   }
   // now v[i]==false
   v[i]=true; // so a change was made and return value
      // is rightfully 'true'
   for (size_t j=i+1; j<n; ++j) v[j]=false;
      // componets right to the point of action (if there are
      // such objects) get set to false.
   return true;
}

//////////////////// class LatticeEnumerator ////////////////////////

void CpmAlgorithms::LatticeEnumerator::update()
{
   Z mL=3;
   Word loc("CpmAlgorithms::LatticeEnumerator::update()");
```

```
   cpmmessage(mL,loc&" started");
   n=b.dim();
   cpmassert(n>0,loc);
   Z i,errorSum=0;
   for (i=1;i<=n;i++) errorSum+=(b[i]<1 ? 1 : 0);
   cpmassert(errorSum==0,loc);
   c=V<Z>(n,1);
   for (i=1;i<n;i++) c[n-i]=c[n-i+1]*b[n-i+1];
   nLattice=c[1]*b[1];
   cpmmessage(mL,loc&" done");
}

Z CpmAlgorithms::LatticeEnumerator::operator()(const V<Z>& x)const
{
   Z mL=3;
   Word loc("CpmAlgorithms::LatticeEnumerator::eval(V<Z>)");
   cpmmessage(mL,loc&" started");
   Z i;
   cpmassert(x.dim()==n,loc);
   Z errorSum=0;
   for (i=1;i<=n;i++) errorSum+=(x[i]<1 || x[i]>b[i] ? 1 : 0);
   cpmassert(errorSum==0,loc);
   Z res=1;
   for (i=1;i<=n;i++) res+=c[i]*(x[i]-1);
   cpmmessage(mL,loc&" done");
   return res;
}

V<Z> CpmAlgorithms::LatticeEnumerator::operator()(Z k)const
{
   Z mL=3;
   Word loc("CpmAlgorithms::LatticeEnumerator::eval(Z)");
   cpmmessage(mL,loc&" started");
   cpmassert(k>0 && k<=nLattice,loc);
   k--;
   V<Z> res(n,0);
   Z i=1;
   while (i<=n){
      Z ri=k/c[i];
      k-=ri*c[i];
      res[i]=ri+1;
      i++;
   }
   cpmmessage(mL,loc&" done");
   return res;
}

bool CpmAlgorithms::LatticeEnumerator::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("CpmAlgorithms::LatticeEnumerator",str);
```

```
   V<Z> b_(b);
   cpmp(b_);
   return true;
}

bool CpmAlgorithms::LatticeEnumerator::scanFrom(istream& str)
{
   V<Z> b_;
   cpms(b_);
   b=b_;
   update();
   return true;
}


//////////////////// class LatticeBox ////////////////////////////////

LatticeBox::LatticeBox(const V<Iv>& vivIn, const V<Z>& vzIn):
viv(vivIn),vz(vzIn)
{
   Z mL=3;
   Word loc("LatticeBox(V<Iv>,V<Z>)");
   CPM_MA
   using CpmAlgorithms::LatticeEnumerator;
   n=viv.dim();
   cpmassert(n==vz.dim(),loc);
   Z i,j;
   np=1;
   for (i=1;i<=n;i++) np*=vz[i];
   val=V<R>(np,0.);
   centers=V< V<R> >(np,V<R>(n,0));
   le=LatticeEnumerator(vz);
   for (i=1;i<=np;i++){
      V<Z> lei=le(i);
      V<R> vali(n);
      for (j=1;j<=n;j++) vali[j]=viv[j].centers(vz[j])[lei[j]];
      centers[i]=vali;
   }
   CPM_MZ
}

void LatticeBox::set(const V<R>& pos, R valIn)
{
   V<Z> zdim(n);
   Z i;
   for (i=1;i<=n;i++){
      zdim[i]=viv[i].quant(pos[i],vz[i]);
   }
   Z k=le(zdim);
   val[k]+=valIn;
```

```
}

VV<R> LatticeBox::matrix()const
{
   Z mL=3;
   Word loc("LatticeBox::matrix()");
   CPM_MA
   cpmassert(n>=2,loc);
   Z mres=vz[2];
   Z nres=vz[1]; // sic

   VV<R> res(mres,nres);
   Z i,j,k;
   for (k=1;k<=np;k++){
      V<Z> vij=le(k);
      Z ix=vij[1];
      Z iy=vij[2];
      i=1+mres-iy;
      j=ix;
      res[i][j]=val[k];
   }
   CPM_MZ
   return res;
}

namespace{ // anonymous namespace for tools

   R xLimit;
      // a number for communication between class xf and function brack

   struct xf{ // for encapsulating some functionality
      R x1,x2,x3;
      R f1,f2,f3;
      xf(R x, R f):x1(x),x2(x),x3(x),f1(f),f2(f),f3(f){}
      void shift(R x, R f){x1=x2;f1=f2;x2=x3;f2=f3;x3=x,f3=f;}
      Z eval()const;
         // the return value encodes the meaning of the data of *this
         // with respect to the aims of brack()
   };

   inline bool signChange(R x, R y)
   {
      return  (x>=0 && y>=0)||(x<=0 && y<=0) ? false : true;
   }

   Z xf::eval()const
   {
      const R fac=2;
      using CpmRoot::absVal;
      using CpmRootX::inf;
```

```
      if (x3>xLimit) return 1;
      if (f3==0) return 2;
      if (signChange(f2,f3)) return 3;
      R d1=f2-f1;
      R d2=f3-f2;
      if (signChange(d1,d2)){
         R var=absVal(d1)+absVal(d2);
         R mini=inf(absVal(f1),absVal(f2),absVal(f3));
         if (mini>fac*var) return 0; else return 4;
      }
      return 0; // no special event, no action triggered
   }

} // anonymous namespace

X3<R,R,Z> CpmAlgorithms::brack(const F<R,R>& f, R xMin,
                               R xMax, R dxMin, R dxMax)
{
   const Z mL=4;

   Z found;
   R x1=xMin;
   R length=xMax-xMin;
   R nh=floor(length/dxMax+0.5);
   R hStart=length/nh; // after doing some steps hStart one should
      // come very close to xMax
   R h=hStart;
   xLimit=xMax; // x1,h,xLimit are the iter ini data
   bool local=false;
iter:
   R f1=f(x1); // x1 needed
   if (cpmverbose>mL) cpmdata<<endl<<"f1="<<f1;
   xf state(x1,f1);
   Z res=0;
   while (res==0){
      R xn=state.x3+h; // h needed
      R fn=f(xn);
      state.shift(xn,fn);
      res=state.eval(); // xLimit needed (hidden)
   }
   // now res!=0
   if (res==1 ){
      if ((!local) || state.x3>=xMax){
         found=0;
         return X3<R,R,Z>(state.x2,state.x3,found);
      }
      else{
         x1=state.x3; // start where you ended, progress
         h=hStart;
         xLimit=xMax;
```

```
         local=false;
         goto iter;
      }
   }
   else if(res==2){
      found=1;
      return X3<R,R,Z>(state.x3,state.x3,found);

   }
   else if(res==3){
      found=2;
      return X3<R,R,Z>(state.x2,state.x3,found);
   }
   else{ // then res==4, we found a change in trend
      // which can be an indication that a local scan may find a
      // root between state.x1 and state.x3
      if (local){
         cpmmessage(mL,
            "CpmAlgorithms::brack(): trend change and no root");
         x1=state.x3; // start where you ended, progress
         h=hStart;
         xLimit=xMax;
         local=false;
      }
      else{
         cpmmessage(mL,
            "CpmAlgorithms::brack(): refinement on trend change");
         x1=state.x1; // start again at the same point, no progress
         xLimit=state.x3;
         h=dxMin;  // have to take a closer look to a limited range at
            // higher resolution
         local=true;
      }
       // now we have new values for the iter ini data
      goto iter;
   }
}

X2<R,bool> CpmAlgorithms::root(const F<R,R>& f, R xMin,
                               R xMax, R dxMin, R dxMax, bool silent)
{
   Z mL=3;
   Word loc("CpmAlgorithms::root(...)");
   cpmmessage(mL,loc&" started");
   const Z iterMax=40;
   const R redFac=0.05;
   X3<R,R,Z> prr=brack(f,xMin,xMax,dxMin,dxMax);
   R x0=prr.c2();
   bool k=true;
   Z found=prr.c3();
```

```
   if (found==0){
      if (!silent) cpmwarning(loc&" could not bracket the root");
      k=false;
   }
   else if (found==1){
      ; // x0 and k are OK as already defined
   }
   else if (found==2){ // bisection
      R x1=prr.c1();
      R x2=prr.c2();
      x0=rootByBisection(f,x1,x2,dxMin*redFac,iterMax);
   }
   else{
      cpmerror("CpmAlgorithms::root() unvalid exit, found="
         &cpmwrite(found));
   }
   return X2<R,bool>(x0,k);
}

R CpmAlgorithms::solKepEqu(R M, R eps, R acc)
{
   R xLast=M+1000, xNew=M;
   while ( cpmabs(xLast-xNew) > acc ){
      xLast=xNew;
      R x1=M+eps*sin(xNew);
      R x2=M+eps*sin(x1);
      xNew=(x1+x2)*0.5;   // My standard provision against oscillations.
         // Works extremely well. Never seen this elsewhere.
   }
   return xNew;
}

R CpmAlgorithms::ssqe(R a, R b, R c, R xMin, R xMax)
{

   R x1,x2,w1,w2,n,xmin,xmax;
   if (a==0){   // linear equation
      if (b==0) return -1; else x1=-c/b;
      if (x1>=xMin && x1<=xMax) return x1; else return -1;
   }
   else{   // quadratic equation

      w2=b*b-4*a*c;

      if (w2>0){   // two real solutions
         n=0.5/a;
         w1=sqrt(w2);
         x1=( b<0. ? (-b+w1)*n : (-b-w1)*n );   // this is the
         // numerically
            // uncritical root (never a difference of large quantities)
```

```
                // also never 0

         x2= c/(a*x1);   // second root from Vieta, c=a*x1*x2
         if (x1<x2) { xmin=x1; xmax=x2;} else { xmin=x2; xmax=x1;}
         if (xmin>=xMin && xmin<=xMax){
            return xmin;
         }
         else if (xmax>=xMin && xmax<=xMax){
            return xmax;
         }
         else{
            return -1;
         }
      }
      else if (w2<0){    // two complex solutions
         return -1;
      }
      else{    // w2==0, this gives the case of coinciding solutions
               // of a quadratic equation
         x1=-b/(2*a);    // notice a!=0
         if (x1>=xMin && x1<=xMax) return x1; else -1;
        }
   }

   cpmerror("psdqe(): never reach this point");

   return x1; // never reached, to avoid compiler warning
}

X2<R,bool> CpmAlgorithms::rootFast(F<R,R> const& f, R xMin, R xMax,
                                   R dx, R eps)
{
   const R red=0.667;
   const R redComp=1-red;
   const Z iterMax=20;

   const Z mL=3;

   R range=xMax-xMin;
   if (range<=0.) cpmerror("rootFast(...): xMax<=xMin");

   R h=CpmRootX::inf(dx,range);
   R x1=xMin;
   R f1=f(x1);
   R err=CpmRoot::absVal(f1);
   if (err<=eps){
      if (cpmverbose>=mL){
         Word mes="rootFast(...): root found immediately, err=";
         mes&=cpmwrite(err);
         cpmmessage(mes);
```

```
      }
      return X2<R,bool>(x1,true);
   }
   Z count=0;

Iter:
   count++;
   R x2=x1+h;
   R x3=x2+h;
   R f2=f(x2);
   R f3=f(x3);
   R hInv=1./h;

   R a=0.5*(f1-2*f2+f3)*hInv*hInv;
   R b=-0.5*(3*f1-4*f2+f3)*hInv;
   R t=ssqe(a,b,f1,0.,xMax-x1);

   if (t==-1){ //
      cpmmessage(mL,"rootFast(...): root called");
      return root(f,xMin,xMax,0.1*dx,10*dx);
   }

   R xRoot=x1+t;
   R fRoot=f(xRoot);
   err=CpmRoot::absVal(fRoot);
   if (err<=eps){
      if (cpmverbose>=mL)
         cpmmessage("rootFast(...): reg. return, err"&cpmwrite(err));
      return X2<R,bool>(xRoot,true);
   }
   if (count==iterMax){
      if (cpmverbose>=mL)
         cpmmessage("rootFast(...): iterMax exhausted, count="&
            cpmwrite(count));
      return X2<R,bool>(xRoot,false);
   }
   x1+=red*t;
   f1=f(x1);
   h=CpmRootX::inf(h,redComp*t);
   goto Iter;
}

X2<R,R> CpmAlgorithms::bisection(const F<R,bool>& f, R tF, R tT,
   R tacc, Z iterMax)
{
   using CpmRoot::absVal;
   if (tacc<=0) cpmerror("CpmAlgorithms::bisection(...): tacc<=0");
   Z count=0;
   R tM;
   while( absVal(tT-tF)>tacc && count++<iterMax ){
```

```
      tM=0.5*(tF+tT);
      if ( f(tM) ) tT=tM; else tF=tM;
   }
   if (count>=iterMax)
      cpmwarning("CpmAlgorithms::bisection() exhausted iterMax");
   return X2<R,R>(tF,tT);
}

R CpmAlgorithms::rootByBisection(const F<R,R>& f, R x1, R x2,
                                 R xacc, Z iterMax)
{
   Z mL=3;
   Word loc("CpmAlgorithms::rootByBisection(...)");
   CPM_MA
   using CpmRoot::absVal;
   R f1=f(x1);
   R f2=f(x2);
   R xM,fM;
   if (f1==0){
      cpmwarning(loc&" x1 found to be a root");
      CPM_MZ
      return x1;
   }
   if (f2==0){
      cpmwarning(loc&" x2 found to be a root");
      CPM_MZ
      return x2;
   }
   if (f1*f2>0){
      cpmerror(loc&": no sign change for initial values");
      return 0; // never get here
   }
   Z count=1;
   while( absVal(x1-x2)>xacc && count<iterMax ){
      xM=0.5*(x1+x2);
      fM=f(xM);
      if (fM==0) return xM;
      if ( f1*fM<=0) { // notice that sign change is what we need,
            // so f2 may be replaced by fM
         f2=fM;
         x2=xM;
      }
      else{ // then f1 and fM have the same sign , and this differs from
         // that of f2, so f2 and fM are a team, f1 gets replaced
         f1=fM;
         x1=xM;
      }
      count++;
   }
   if (count>=iterMax){
```

```
      cpmwarning(loc&": exhausted iterMax="&cpm(iterMax));
      cpmdebug(x1);
      cpmdebug(x2);
   }
   CPM_MZ
   return 0.5*(x1+x2);
}


R CpmAlgorithms::fixedPoint(const F<R,R>& f, const R& x0,
   R eps, Z iterMax)
{
   using CpmRoot::disVal;
   R xOld=x0;
   R xNew=f(xOld);
   Z count=0;
   while(disVal(xOld,xNew)>eps && count++ < iterMax ){
      xOld=xNew;
      R xAux=f(xOld);
      xNew=(f(xAux)+xAux)*0.5;
   }
   if (count>=iterMax)
      cpmwarning("CpmAlgorithms::fixedPoint() exhausted iterMax");
   return xNew;
}

//////////////////////// Quantizer ////////////////////////////////////
// auxiliaries:

namespace{

Z qant0(const R& x){ return (Z)(x+0.5);}

Z quantV(const R& x, const Vo<R>& v)
{
   return v.locate(x);
}

Z quantIv(const R& x, const Iv& iv, const Z& n)
{
   return iv.stair(x,n);
}

} // namespace

// constructors

CpmAlgorithms::Quantizer::Quantizer()
{
   *this=F<R,Z>(qant0);
```

```
}

CpmAlgorithms::Quantizer::Quantizer(R xL, R xU, Z n)
{
   Word loc("Quantizer::Quantizer(R xL, R xU, Z n)");
   cpmassert(n>0,loc);
   *this=F2<R,Iv,Z,Z>(Iv(xL,xU),n)(quantIv);
}

CpmAlgorithms::Quantizer::Quantizer(Iv iv, Z n)
{
   Word loc("Quantizer::Quantizer(Iv iv, Z n)");
   cpmassert(n>0,loc);
   *this=F2<R,Iv,Z,Z>(iv,n)(quantIv);
      // this stores the data iv, n that characterize the
      // instant of Quantizer constructed by the given constructor.
      // We could define another constructor of the class which stores a
      // completely different data structure; for example some complex
      // numbers. This shows that a class needs not to have a single data
      // member'layout' if sufficiently flexible data structures are
      // available.
}

CpmAlgorithms::Quantizer::Quantizer(const V<R>& x, bool ordered)
{
   Z n=x.dim();
   if (n<=0){
      Z constVal=0;
      *this=F<R,Z>(constVal);
   }
   else{
      Vo<R> y(x);
      if (!ordered) y=y.strictlyIncreasingVersion();
      *this=F1<R,Vo<R>,Z>(y)(quantV);
   }
}

//////////////////////////// IntQuant ////////////////////////////////

namespace{

   Z intQuant(const Z& i, const Z& zL, const Z& nP, const R& spanInv)
   {
      R p=(i-zL)*spanInv;
      bool restrict=true;
      return Iv(0,1).quant(p,nP,restrict);
   }
}

CpmAlgorithms::IntQuant::IntQuant(Z xL, Z xU, Z n)
```

```
{
   if (xL<=xU){
      zL=xL;
      zU=xU;
   }
   else{
      zL=xU;
      zU=xL;
   }
   nP=(n<1 ? 1 : n);
   spanInv=( zU==zL ? 0 : 1./(zU-zL) );
   f=F3<Z,Z,Z,R,Z>(xL,nP,spanInv)(intQuant);
}

//////////////////////// Percentilizer ////////////////////////////////////

namespace { // auxiliaries in anonymous namespace
   using CpmArrays::Vo;
   using CpmArrays::V;

class quantil: public FncObj<R,R>{
   Vo<R> b;
public:
explicit quantil(const V<R>& a):b(a){}
   R operator()(const R&)const;
};

R quantil::operator()(const R& p)const
// notice: function quantil will only be called for b in ascendend order
// quantil(0.5,b)=median(b);
// quantil(0.25,b)=Hinge_1(b);
// quantil(0,b)=minimum(b);
// quantil(0.75,b)=Hinge_2(b);
// quantil(1,b)=maximum(b);
// see Weisstein: CRC Concise Encyclopedia of Mathematics p. 844 for
// Hinge.
// we use an obvious method to define this in cases where n is not of the
// form
// 4*m+5
{
   R pp=p;
   if (pp<0) pp=0;
   if (pp>1) pp=1;
   Z n=b.dim();
   if (n==0) return 0;
   R ir=1+(n-1)*pp;
   if (ir==1.) return b[1];
   if (ir==n) return b[n];
   Z i=(Z)floor(ir);
   R frac=ir-i; // notice 0<=frac<1
```

```
   return frac*b[i]+(1.-frac)*b[i+1];
}

} // anonymus namespace for hiding this construct

// constructors

CpmAlgorithms::Percentilizer::Percentilizer(const V<R>& a):arr(a)
{
   if (arr.dim()>0) arr.order();
   p=new quantil(arr); // no ordering done here !!!
   // p i protected member base class P< FncObj<R,R> >
   // notice that the initialization of u by the default constructor is
   // just right.
   // this structure of a constructor seems to be the most flexible one
}

V<R> CpmAlgorithms::Percentilizer::select(const Iv& iv)const
{
   Z mL=4;
   static Word loc("CpmAlgorithms::Percentilizer::select(Iv)");
   cpmmessage(mL,loc&" started");
   R iv1=iv[1];
   cpmassert(iv1>=0,loc);
   R iv2=iv[2];
   cpmassert(iv2<=1,loc);
   Z i,n=arr.dim();
   R x1=(*this)(iv1);
   R x2=(*this)(iv2);
   Iv ivOk(x1,x2);
   V<bool> sel(n);
   for (i=1;i<=n;i++) sel[i]=ivOk.contains(arr[i]);
   cpmmessage(mL,loc&" done");
   return arr.select(sel);
}

R CpmAlgorithms::Percentilizer::mean(const Iv& iv)const
{
   Va<R> arrOk=select(iv);
   Z nOk=arrOk.dim();
   return nOk==0 ? 0 : arrOk.sum()/nOk;
}

R CpmAlgorithms::gaussRandomValue(const R& mean, const R& sigma)
// Returns normally distributed random numbers with mean and sigma
// given by the respective arguments.
// Modified from gasdev(long*) of Press et al. by introducing another
// basic random generator and by not specializing the arguments.
// Notice that the involved method of generation will not easily enable
// to define a random array (i.e. introduction of the argument j as in
```

```
// ranVal(const R& x, Nd j) such that consequtive components are
// independent.
{
    static R randRef=1.;
    static Z iset=0;
    static R gset;
    using CpmRoot::ranVal;
    if (sigma==0.) return mean;
    R fac,rsq,v1,v2;
    if (iset==0){
        do{
            v1=ranVal(randRef);
            v2=ranVal(randRef);
            rsq=v1*v1+v2*v2;
        }
        while(rsq>=1.0 || rsq==0.0);
            // these are the results we don't want
        fac=sigma*sqrt(-2.*log(rsq)/rsq);
        gset=mean+v1*fac;
        iset=1;
        return mean+v2*fac;
    }
    else{
        iset=0;
        return gset;
    }
}

//////////////////// class PeriodicAlarm ////////////////////////////

using CpmAlgorithms::PeriodicAlarm;

PeriodicAlarm::PeriodicAlarm(R per, R tIni, R phase):
active(true),
period(per),
counter(0),
lastAlarm(tIni)
{
   Z mL=3;
   static Word loc="PeriodicAlarm(R,R,R)";
   static R relTol=1e-3;
   CPM_MA
   if (period<=0) active=false;
   tol=relTol*period;
   nextLatticePoint=tIni+period*phase;
}

bool PeriodicAlarm::operator()(R t)
{
   if (!active){
```

```
      return false;
   }
   R tDiff=t-nextLatticePoint;
   if (tDiff+tol>=0){
      Z fr=(Z)(tDiff/period); // starts with 0
      lastAlarm=t;
      fr+=1;
         // fr should start with 1
      nextLatticePoint+=fr*period;
      counter+=fr;
      return true;
   }
   else{
      return false;
   }
}

//////////////////////// class SingleAlarm ////////////////////////////

using CpmAlgorithms::SingleAlarm;

bool SingleAlarm::operator()(R t)
{
   if (done) return false;
   if (t>=alarmPoint){
      done=true;
      return true;
   }
   else{
      return false;
   }
}

//////////////////////// class CyclicAlarm ////////////////////////////

using CpmAlgorithms::CyclicAlarm;

CyclicAlarm::CyclicAlarm(Z per, Z cogIni, Z maxAlarms):
period(per),cog(cogIni),counterMax(maxAlarms)
{
   Z mL=3;
   static Word loc("CyclicAlarm(Z,Z,Z)");
   CPM_MA
   if (period<1){
      dormant=true; // alarm=false by initialization
      done=true;
   }
   else{
      while (cog>=period) cog-=period;
      alarm=true;
```

```
   }
   CPM_MZ
}

void CyclicAlarm::step(void)
{
   if (dormant){
      return;
      // don't change state
   }
   else if (done){
     alarm=false;
   }
   else{
     cog+=1;
     if (cog==period || cog==0){
         cog=0;
         alarm=true;
         counter+=1; // now this the number of alarms given
         if (counter==counterMax){
            done=true;
            alarm=false;
         }
      }
      else{
         alarm=false;
      }
   }
}

V<B> CyclicAlarm::record(Z n)
{
   V<B> res(n);
   for (Z i=1;i<=n;i++) res[i]=(*this)();
   return res;
}

//////////////////////// class NonCyclicAlarm ////////////////////////////

using CpmAlgorithms::NonCyclicAlarm;

NonCyclicAlarm::NonCyclicAlarm(const V<Z>& points):points_(points),
counter(1),cog(1)
{
   Z mL=3;
   static Word loc("NonCyclicAlarm()");
   CPM_MA
   Vo<Z> p_=points_;
   points_=p_.strictlyIncreasingVersion();
   counterMax=points_.dim();
```

```
   if (counterMax<=0){
      alarm=false;
      done=true;
   }
   else{
      alarm=(cog==points_[counter]);
      if (alarm) counter+=1;
      done=(counter>counterMax);
   }
   CPM_MZ
}

void NonCyclicAlarm::step(void)
{
   if (done){
     alarm=false;
   }
   else{
     cog+=1;
     if (cog==points_[counter]){
         alarm=true;
         counter+=1; // now this the number of alarms given
         if (counter>counterMax){
            // never call points_[counterMax+1]
            done=true;
         }
      }
      else{
         alarm=false;
      }
   }
}

//////////////////////// class BurstAlarm ////////////////////////////

using CpmAlgorithms::BurstAlarm;

BurstAlarm::BurstAlarm(Z per, Z burPer, Z burLen):
Base(per,0,-1),burst(burPer,0,burLen)
{
   Z mL=3;
   static Word loc("BurstAlarm(Z,Z,Z)");
   CPM_MA
   CPM_MZ
}

void BurstAlarm::step(void)
{
   if (getAlarm() && burst.ready()) burst.restart();
```

```
   Base::step();
   burst.step();
   setAlarm(getAlarm()||burst.getAlarm());
}

//////////////////////////////////////////////////////////////////////
Z CpmAlgorithms::getNerFrc_1(R& x, R xL)
{
  Z mL=4;
  Word loc("CpmAlgorithms::getNerFrc_1(R&,R)");
  CPM_MA
  Z res=0;
  cpmassert(x<xL,loc);
  cpmassert(x!=0,loc);
  res=cpmround(xL/x);
  cpmassert(res!=0,loc);
  if (res!=0) x=xL/res;
  CPM_MZ
  return res;
}

namespace{
   Z penalty(Z n1, Z n2, Z n)
   {
      Z z1=n1, z2=n2;
      CpmRootX::order<Z>(z1,z2);
      Z p=z1*z2;
      Z a1=z2-z1;
      Z a2=p-n;
      if (a2<0) return 1+n*n;
      return a1+a2;
   }
}

Z2 CpmAlgorithms::genSqrRoot(Z n)
{
   Word loc("CpmAlgorithms::genSqrRoot(Z)");
   Z z1=cpmround(::sqrt((R)n));
   Z z2=n/z1;
   Z n0=z1*z2;
   if (n0==n) return Z2(z1,z2);
   CpmRootX::order<Z>(z1,z2);
   if (n0>n){ // in principle OK, but try an improvement
      Z p1=penalty(z1,z2,n);
      Z p2=penalty(z1+1,z2-1,n);
      if (p2<p1){
         z1++; z2--;
      }
      return Z2(z1,z2);
   }
```

```
   while (n0<n){
      Z p1=penalty(z1+1,z2,n);
      Z p2=penalty(z1,z2+1,n);
      if (p1<=p2) z1++;
      else z2++;
      n0=z1*z2;
   }
   CpmRootX::order<Z>(z1,z2);
   return Z2(z1,z2);
}
```

9 cpmangle.h

```
//? cpmangle.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_ANGLE_H_
#define CPM_ANGLE_H_
//////////////////////////////////////////////////////////////////////////
// Description: A class describing angles and the 1-dimensional
//   torus
//////////////////////////////////////////////////////////////////////////
#include <cpmc.h>
#include <cpmtypes.h>
#include <cpmv.h>

namespace CpmGeo{

   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmRoot::Word;
   using CpmRoot::C;
   using CpmRoot::Pi;
   using CpmArrays::V;

enum AngleUnit { CYCLE, HOUR, DEG, RAD};
   // cycle (full angle), hour, degree, radian
enum AngleRange { POS, NEG};
   // always positive, negativ allowed
enum AngleHex {DEC, MIN, SEC};
   // full decimal, minutes, minutes and seconds

const R degree=Pi/180.;

class Angle{ // angles
   // an instance of Angle is an angle, the manifold of all angles is
   // a 1-dimensional torus. There are two interpretations.
   // (i) Angle as a coordinate on a circle (as a curve) and
   // (ii) angle as a descriptor of a rotation arround a given axis.
   // Both are closely related due to the fact, that the rotations act
   // transitively and effectively on a circle. For defining a proper set
   // of methods we have not to make a choice in favor of one of these
   // interpretations.
   // Interpretation (i) endows Angle most directly with the structure of
   // an oriented metric space, and (ii) with the structure of an
   // Abelian group

   static void normalizeAngle(R& a0);
      // changes a0 into a value in the interval (-Pi,Pi] by adding
      // a multiples of 2*Pi
```

```
   R a_; // is read as alpha, is the measure of the angle as arc length
      // on the unit circle (for effective usage of built-in
      // trigonometric functions. The range is by definition (-Pi,Pi].

   void normalize(void){ normalizeAngle(a_);}

public:
   typedef Angle Type;
   typedef R ScalarType;

   CPM_IO
   CPM_ORDER

   CPM_SUM_PLAIN
   CPM_DIFFERENCE_PLAIN
      // arithmetics among angles (= arithmetics of R/(2*Pi), residue
      // classes 'modulo 2*Pi')

   CPM_SCALAR_M
   // The meaning of multiplication with real numbers derives by
   // continuity from the meaning of multiplication by rational
   // numbers. Let us consider first multiplication by a natural
   // number n. The proper definition follows from the group
   // structure: alpha*n := alpha + ... + alpha (n terms).
   // Multiplication by 1/m for natural number:
   // alpha*(1/m) := smallest angle beta for which beta*m=Angle(),
   // where Angle() is the zero angle which is the unit element of
   // the group. Smallness, obviously, refers to the distance to
   // the unit element. Obviously, one may add (2*Pi)/m to beta
   // (let the result be called beta') and one also has
   // beta'*m=Angle().

   CPM_CONJUGATION
      // a idempotent conjugation operation. If the angle is interpreted
      // as the phase angle of a complex number, this corresponds to
      // the complex conjugation of the number. This is a natural
      // geometric operations for angles, resulting from the distinction
      // of one angle as the zero-angle (origin on torus)

   Word nameOf()const{ return Word("Angle");}

   R operator /(const Angle&)const;
      // alpha/beta is the number r for which beta*r=alpha.
      // Notice the comment to ScalarType on multiplying
      // Angles by numbers. This is not defined in residue
      // classes modulo 2*Pi.
      // Products of angles don't give angles; they are
      // omitted here.
```

```
// constructors

   Angle(void):a_(0.){}
      // angle zero

   explicit Angle(R angle, bool deg=true);
      // Constructor for Angles out of real numbers. The first argument
      // is always interpreted as an angle in degrees, unless deg==0
      // Never enable automatic conversion. Would result in ambiguous
      // arithmetics. Input needs not to be normalized

   Angle(R vAngle, AngleUnit au);
      // creates an angle which is vAngle*au

   explicit Angle(C const& z):a_(z.arg()){}
      // Constructs the phase angle (polar angle) of z

   C expi()const{ return C(1.,a_,"polar");}
      // returns the complex number exp(i*a). Thus Angle now has a
      // bi-directional efficient interface to C

// automatic conversion in one direction Angle-->R is OK

   operator R(void)const{return a_;}
   // cast to R for making trigonometric functions defined for angles

   R toR()const{ return a_;}
   // explicit conversion in a standard manner

   R cut( R c)const;
   //: cut
   // returns a value in the range (c*degree-2*Pi,c*degree] which from
   // a is derived by adding or subtracting a multiple of 2*Pi.
   // To have such a facility turned out to be very useful in
   // dealing with angular motion of stepper motors (project
   // CO-former)

   void qun_(Z n);
      //: quantize
      // Changes angle *this into the nearest of n equi-spaced
      // angles which are equi-distributed over the whole range.
      // For n<1, no action.

   R toDeg(Z i=0)const;
      //: to degrees
      // returns the angle *this in degrees. For the default
      // value 0 of the argument, the range is the standard range
      // (-180,180], and for 1 the range is [0,360)

   Word toWord(AngleUnit u, AngleRange r, AngleHex h, Z digits=1)const;
```

```
      // returns a string which gives the angle in the units
      // given by u, in the range indicated by r, in a hexagesimal
      // subdivision characterized by h and with a number of figures
      // after the decimal point is given by digits. If digits is -1,
      // there is one figure behind the decimal point and this is rounded
      // to 0 or 5 corresponding to the accuracy with which the hour
      // circle of my C5 telescope can be read. The meaning of the
      // parameters follows from the explanation to the
      // enumerations AngleUnit, AngleRange, and AngleHex

   friend Word AngleToWord(const Angle& alpha,
      AngleUnit u, AngleRange r, AngleHex h, Z digits=1)
      { return alpha.toWord(u,r,h,digits);}

// modifying an angle
   void setArc(R alpha);
      // input is an arc (i.e. something like a, but need not be
      // normalized)

// absolute value
   R absVal(void)const{ return (a_>=0. ?  a_ : -a_);}
      // absolute value: positive arc as a number (distance from zero
      // angle)

   R disVal(const Angle& phi)const;
   bool isPos()const{ return a_>=0;}
      // returns true if the angle is in quadrant 1 or 2 (by definition,
      // 0 belongs to quadrant 1, Pi to quadrant 2 ) and false else

   void pos(){ if (a_<0) a_=-a_;}
      // makes the angle positive by conjugation

// mean values taking into account the embedding in the cpmplex numbers
   friend Angle mean(const CpmArrays::V<Angle>& phi);
      // mean of an array of angles.
      // Polar angle of exp(i*phi[1])+...+exp(i*phi[n]).
      // There seems to be no easy way to define this on R/~
      // where a1 ~ a2 <==> a1 = a2 (mod 2Pi)

   friend Angle mean(const CpmArrays::V<Angle>& angles,
      const CpmArrays::V<R>& weights);
      // weighted mean of an array of angles
      // Polar angle of
      // weights[1]*exp(i*phi[1])+...+weights[n]*exp(i*phi[n])

   friend Angle RealToAngle(R x, AngleUnit u, AngleHex h);
      // returns an angle alpha which is given by a real number
      // according the format indicated by the unit u and the
      // subdivision schema indicated by h.
```

```
   friend Angle realToAngle(R c){ return Angle(c);}
      // converts a real number, interpreted as an arc into a
      // instance of class Angle

   static V<R> smoothAngleList(const V<Angle>& va);
      // given a list va of Angles, we return a list res of corresponding
      // R's such that
      //    res[i]==va[i] mod(2*Pi)
      // where adding or subtracting of 2*Pi is done such that the
      // resulting list is as smooth as possible. More precisely we
      // assure that for ech i
      // | res[i+i]-res[i]| < |res[i+1]+2*Pi-res[i]|
      // and
      // | res[i+i]-res[i]| < |res[i+1]-2*Pi-res[i]|
      // As a result of this the res[i] may grow to much larger values
      // than 2*Pi as in a list of successive angular positions of a
      // rotating wheel

    static V<R> smoothList(const V<R>& vr);
      // similar to previous function. The argument list is not a list
      // of angles but a list of their conversions to numbers (based
      // on the RAD unit)

    static Angle deg1, deg90, deg180, deg270, deg360;
};

} // namespace

#endif
```

10 cpmangle.cpp

```
//? cpmangle.cpp
//? Status of work 2008-10-25.
//?
//Description: see cpmangle.h

#include <cpmangle.h>

using namespace CpmStd;

using namespace CpmGeo;
using CpmRoot::R;
using CpmRoot::Pi;
using CpmRoot::Z;
using CpmRoot::C;
using CpmRoot::Word;

namespace{
   const R fullAngle=CpmRoot::Pi2;
   const R iFullAngle=R(1)/fullAngle;
   const R hour=fullAngle/24;
}

Angle Angle::deg1=Angle(1,DEG);
Angle Angle::deg90=Angle(90,DEG);
Angle Angle::deg180=Angle(180,DEG);
Angle Angle::deg270=Angle(270,DEG);
Angle Angle::deg360=Angle(360,DEG);

void Angle::normalizeAngle(R& a)
{
   if (a>-Pi && a<=Pi) return;
      // then no normalization needed. Introduced 2006-03-29
      // after it was observed that normalizing the zero-angle
      // caused difficulties
   a*=iFullAngle;
   a-=floor(a);
   if (a>0.5) a-=1;
   a*=fullAngle;
}

Angle::Angle(R angle, bool deg):a_(angle)
{
   if (deg) a_*=degree;
   normalize();
}

Angle::Angle(R angle, AngleUnit au):a_(angle)
```

```
{
   if (au==RAD){
      ;
   }
   else if (au==DEG){
      a_*=degree;
   }
   else if (au==CYCLE){
      a_*=fullAngle;
   }
   else if (au==HOUR){
      a_*=hour;
   }
   else{
      cpmerror("Angle(R, AngleUnit): unvalid AngleUnit");
   }
   normalize();
}

void Angle::setArc(R alpha)
{
   a_=alpha;
   normalize();
}

namespace{
void quantize(R& x, Z n)
{
   R y=n*x;
   y=floor(y+0.5);
   x=y/n;
}

}
Angle& Angle::operator +=(const R& s)
{
   a_+=s;
   normalize();
   return *this;
}

Angle& Angle::operator *=(const R& s)
{
   a_*=s;
   normalize();
   return *this;
}

Angle Angle::operator -(void)const
{
```

```
   Angle res;
   res.a_=-a_;
   res.normalize();
   return res;
}

Angle& Angle::operator +=(const Angle& s )
{
   a_+=(s.a_);
   normalize();
   return *this;
}

Angle& Angle::operator -=(const Angle& a1 )
{
   a_-=a1.a_;
   normalize();
   return *this;
}

Angle Angle::operator +(const Angle& a1)const
{
   return Angle(a_+a1.a_,RAD);
}

Angle Angle::operator -(const Angle& a1)const
{
   return Angle(a_-a1.a_,RAD);
}

R Angle::operator /(const Angle& a1)const
{
   R a1Inv=cpminv(a1.a_);
   return a_*a1Inv;
}

Angle Angle::con(void)const
{
   if (a_==Pi) return *this;
   else return Angle(-a_,RAD);
}

R Angle::cut(R c)const
{
   R lim=c*degree;
   R ar=a_+fullAngle;
   if (ar>lim) ar-=fullAngle;
   return ar;
}
```

```
bool Angle::prnOn(ostream& out)const
{
   return CpmRoot::write(a_,out);
}

bool Angle::scanFrom(istream& in)
{
   bool res=CpmRoot::read(a_,in);
   normalize();
   return res;
}

Angle CpmGeo::mean(const CpmArrays::V<Angle>& angles)
{
   Z i,n=angles.dim();
   Word loc("Angle CpmGeo::mean(const CpmArrays::V<Angle>& angles)");
   cpmassert(n>0,loc);
   C z,sum(0,0);
   for (i=1;i<=n;i++){
      z.polar(1.,angles[i]);
      sum+=z;
   }
   R r,phi;
   sum.toPolar(r,phi);
   return Angle(phi,RAD);
}

Angle CpmGeo::mean(const CpmArrays::V<Angle>& angles,
   const CpmArrays::V<R>& weights)
{
   Word loc("Angle CpmGeo::mean(...)");
   Z i,n=angles.dim(),n1=weights.dim();
   cpmassert(n>0,loc);
   cpmassert(n1>=n,loc);
   C z,sum(0,0);
   for (i=1;i<=n;i++){
      z.polar(1.,angles[i]);
      sum+=z*weights[i];
   }
   R r,phi;
   sum.toPolar(r,phi);
   return Angle(phi,RAD);
}

Z Angle::com(const Angle& a1)const
{
   if (a_<a1.a_) return 1;
   else if (a_>a1.a_) return -1;
   else return 0;
}
```

```
R Angle::disVal(const Angle& phi)const
{
   R res=(*this-phi);
   return res*=iFullAngle;
}

R Angle::toDeg(Z i)const // 'to degrees'
   {
      static R iDegree=R(180)/Pi;
      R b=a_*iDegree;
      if (i==1){
         if (b<0) b+=360;
      }
      return b;
   }

Word Angle::toWord(AngleUnit u, AngleRange r, AngleHex h, Z digits)const
{
   const R iDegree=R(180)/Pi;
   const R ihour=R(12)/Pi;
   R y=a_;
   if (r==POS && y<0) y+=fullAngle;
    Z flag=0;
   Word res, sep=":";
   char* ff="%-#3.1f";
   char* fff="%2d";
   if (digits==0)  ff="%-#2.0f";
   if (digits==1)  ff="%-#3.1f";
   if (digits==-1){ff="%-#3.1f"; flag=1;}
   if (digits==2)  ff="%-#4.2f";
   if (digits==3)  ff="%-#5.3f";
   if (digits==4)  ff="%-#6.4f";
   if (digits==5)  ff="%-#7.5f";
   if (digits==6)  ff="%-#8.6f";

   if (y<0){
      res="-";  // this is the first part of the result
                // further parts will be appended later
      y=-y;
   }
   else{
      res=" ";
   }

   if (u==CYCLE){
      y*=iFullAngle; // positive angle in full angles (turns)
   }
    else if (u==HOUR){
      y*=ihour; // positive angle in hours
```

```
   }
   else if (u==DEG){
      y*=iDegree; // positive angle in degrees
   }

   if (h==DEC){
      if (flag) quantize(y,2);
      res=res&CpmRoot::toWord(y,ff);
      return res;
   }

   Z ip=Z(y); // integer part (degrees or hours depending on u)
   res=res&CpmRoot::toWord(ip,fff);
   y-=ip;
   y*=60;  // y=minutes (time or angle depending on u)
   if (h==MIN){
      if (flag) quantize(y,2);
      res=res&sep&CpmRoot::toWord(y,ff);
      return res;
   }

   ip=Z(y); // integer part (minutes of time or angle)
   res=res&sep&CpmRoot::toWord(ip,fff);
   y-=ip;
   y*=60; // y=seconds (time or angle depending on u)
   if (h==SEC){
      if (flag) quantize(y,2);
      res=res&sep&CpmRoot::toWord(y,ff);
      return res;
   }
   return res;  // this should not be reached, for symmetry and for
      // avoiding a warning
}

Angle CpmGeo::RealToAngle(R x, AngleUnit u, AngleHex h)
{
   const R i60=1./60.;
   Z sign;
   Angle res;
   if (h==DEC){
      if (u==CYCLE) x*=fullAngle;
      if (u==HOUR) x*=hour;
      if (u==DEG) x*=degree;
      Angle::normalizeAngle(x);
      res.a_=x;
      return res;
   }
   if (x<0.){
      sign=-1; x=-x;
   }
```

```
   else{
      sign=1;
   } // now x is positive
   R ix=floor(x);
   R y=(x-ix)*100;
   if (h==MIN){
      x=ix+y*i60;
      if (u==CYCLE) x*=fullAngle;
      if (u==HOUR) x*=hour;
      if (u==DEG) x*=degree;
        x*=sign;
      Angle::normalizeAngle(x);
      res.a_=x;
      return res;
   }

   R iy=floor(y);
   R z=(y-iy)*100;
   if (h==SEC){
      x=ix+(iy+z*i60)*i60;
      if (u==CYCLE) x*=fullAngle;
      if (u==HOUR) x*=hour;
      if (u==DEG) x*=degree;
      x*=sign;
      Angle::normalizeAngle(x);
      res.a_=x;
      return res;
   }
   return res;
}

V<R> Angle::smoothAngleList(const V<Angle>& va)
{
   Z i,n=va.dim();
   V<R> res(n);
   for (i=1;i<=n;i++) res[i]=va[i];
   for (i=2;i<=n;i++){
      R diff=res[i]-res[i-1];
      R dz=cpmabs(diff);
      if (dz<Pi) continue;
      R dp=cpmabs(diff+fullAngle);
      R dm=cpmabs(diff-fullAngle); //p,z,m stands for plus, zero, minus
      R dMin=cpminf(dp,dz,dm);
      if (dp==dMin) res[i]+=fullAngle;
      else if (dm==dMin) res[i]-=fullAngle;
      else ;
   }
   return res;
}
```

```
V<R> Angle::smoothList(const V<R>& vr)
{
   Z i,n=vr.dim();
   V<R> res=vr;
   for (i=2;i<=n;i++){
      R ri=res[i];
      R rim=res[i-1];
      R diff=ri-rim;
      R dz=cpmabs(diff);
      if (dz<Pi) continue;
      R dp=cpmabs(diff+fullAngle);
      R dm=cpmabs(diff-fullAngle); //p,z,m stands for plus, zero, minus
      R dMin=cpminf(dp,dz,dm);
      if (dp==dMin){
         res[i]+=fullAngle;
      }
      else if (dm==dMin){
         res[i]-=fullAngle;
      }
      else ;
   }
   return res;
}

void Angle::qun_(Z n)
{
   if (n<1) return; // do nothing for meaningless input
   R a=(a_+Pi)*iFullAngle*n; // ia a value between 0 and n
   a=cpmround(a);
   a*=(fullAngle/n);
   a_=a-Pi;
}
```

11 cpmapplication.h

```
//? cpmapplication.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_APPLICATION_H
#define CPM_APPLICATION_H

//////////////// Introductory section ////////////////////////////////////
//
//  Description: Provides an application framework that makes
//    convenient graphical and messaging capabilities available.
//
//  The user has to provide the definition of function
//     Z CpmApplication::main_(void).
//  The definition of the function
//     int main(int argc, char *argv[]),
//  which each C++ program has to define, is provided by having
//  cpmapplication.cpp among the source files of the C++ project.
//  The context in which main_() is called in cpmapplication.cpp
//  provides all C+- communication capabilities by having done all
//  necessary initializations.
//
//  The following code gives a simple, though realistic example.
//  It is ment as content of a source file smallexperiments.cpp. In
//  reading this code, please ignore the leading comment indicators '//'.
//  If we would switch here to the C comment style, my simple
//  documentation scripts are likely to get fooled.
//
//#include <cpmc.h>
//#include <cpmdigraph.h>
//#include <cpmfa.h>
//#include <cpmapplication.h>
//#include <cpmtestr.h>
//#include <cpmrecordhandler.h>
//#include <cpmmultiimage.h>
//   // not all needed for test3()
//
//using namespace CpmRoot;
//using namespace CpmRootX;
//using namespace CpmSystem;
//using namespace CpmFunctions;
//using namespace CpmArrays;
//using namespace CpmGraphics;
//using namespace CpmImaging;
//using namespace CpmApplication;
//using namespace CpmGeo;
//
//R tWait=0;
```

```
//   ...
//void test3()
//{
//   // test of composing several images into one
//   Word dr=Message::ext("images");
//   Word n11=glue(dr,"bussard.ppm");
//   Word n12=glue(dr,"gnom.ppm");
//   Word n21=glue(dr,"thomas_meier.ppm");
//   Word n22=glue(dr,"kohlmeisel.ppm");
//
//   VV<Img24> vv(2,2); // 2 by 2 matrix of images
//   vv[1][1]=Img24(n11);
//   vv[1][2]=Img24(n12);
//   vv[2][1]=Img24(n21);
//   vv[2][2]=Img24(n22);
//   Z m=750, n=1000; // height and width of combined
//      // image
//   ImgR ir(vv,m,n);
//      // combined image, auto-layout
//   Graph g;
//   ir.mark(g);
//   g.dis(true);
//      // shows the composed image and writes a ppm
//   cpmwait(tWait,2); // remaining wait time to be displayed
//       // on pane 2 of the status bar. There are 4 panes,
//       // the first one is for news from CpmSystem::Message.
//}
//// define  functions test1(), test2()
//   ....
//
//Z CpmApplication::main_(void)
//{
//   Z mL=1;
//   Z sel=args_[2].toZ();
//   Word loc("CpmApplication::main_(), sel="&cpm(sel));
//   CPM_MA
//   title_="smallexperiments";
//   tWait=args_[6].toR();
//   if (sel==1) test1();
//   else if (sel==2) test2();
//   else if (sel==3) test3();
//   else test4();
//   CPM_MZ
//   return 0;
//}
//////////////////////////////////////////////////////////////////////////

#include <cpmv.h>

namespace CpmSystem{
```

```
   void updFromIniFile();
   //: update from ini-file
}

namespace CpmApplication{
   // Provides a framework for defining  c o n s o l e  applications which
   // offer convenient graphical and messaging capabilities.
   //
   // Both capabilities can be eliminated at compile time by enabling or
   // disabling the lines
   //    #define CPM_NOGRAPHICS
   // and
   //    #define CPM_NOLOG
   // in file cpmdefinitions.h.
   // If CPM_NOGRAPHICS is defined, no graphics or images can be shown on
   // screen. But they still can be created in memory and written to file
   // in the portable pix-map format (*.ppm). The size of these images is
   // not limited to screen size. With this setting of CPM_NOGRAPHICS,
   // the program needs not to be linked with its screen graphics enabling
   // library, which is OpenGL and (Open)GLUT. It then also needs no
   // access to to the gl and glut header files. Of course, this is only
   // possible since all graphical primitives, including fonts (greek
   // included) are implemented within C+-.
   // Disabling message streams to the console and to log-files (as done
   // by #define CPM_NOLOG) was essential in a large industrial program
   // where my C+- program translated the intended motions of a complex
   // wire bending tool into stepper motor control files.
   // To the IT-professional, who had finally to put all these control
   // files together and to download them to the the wire-O bending robot,
   // the messages from my program were disturbing and he was happy to
   // hear that there was no problem switching them off. During
   // development and testing these messages played such a vital role that
   // I failed to anticipate that in a different mode of usage, there are
   // different priorities and needs.
   // Many of the design decisions in C+- are actually revisions of old
   // decisions which turned out to support adaption to new needs not
   // as efficiently as they should.

   // A  C + -  a p p l i c a t i o n  (not just a C++ application which
   // uses some C+- classes) has cpmapplication.cpp among its source
   // files and defines the function
   //    Z CpmApplication::main_(void)
   // in its highest-level source files. This highest level source file
   // (which typically has name that characterizes the application,
   // and for which a 'cpm' in the name would be misleading) has to
   // include the present file cpmapplication.h. The the function
   //    int main(int argc, char *arv[])
   // which a console application needs to define, is defined in
   // cpmapplication.cpp in a way which depends on CPM_NOGRAPHICS and
   // CPM_NOLOG as explained above.
```

```
   // See cpminifilebasapp.h for an extensions of the C+- application
   // framework in applications based on reading ini-files

   using CpmRoot::Word;
   using CpmRoot::Z;
   using CpmArrays::V;

// data

   extern Word title_;
      // Title of the whole program. Most conveniently to be defined
      // within function main_() as i the example of the Introductory
      // section.

   extern V<Word> args_;
      // An array that waits to get initialized by commandline arguments
      // and by additional arguments which may be provided by an ini-file.
      // A C+- application (see definition above) args_[1] always as the
      // full name (including directory location) of the program's
      // executable.
      // In calling such a program from the command line, one may enter
      // additonal parametes, which main(...) transforms into components
      // of args_. Such additional parameters do not affect any function
      // defined in cpmapplication.cpp. They can be used however in
      // programming the following function Z main_(), see the
      // introductory section of the present file for an example.
      // Function main, as implemented in cpmapplication.cpp always
      // checks for ini-files for getting additional data. From data which
      // are both given on the command line and in the ini-file, the
      // command line always wins.

      // The class IniFileBasedApplication introduces a particular form of
      // args_ as a list consisting of:
      //    selection,
      //    verbosity,
      //    initial wait time,
      //    relative path of further input data,
      //    final wait time.
      // This structure and meaning needs, however, not be followed in an
      // implementation of of main_().
      // V<Word> is the most flexible among the lower-level data types
      // and it should not, at this place, replaced by a more specific
      // one.

// functions

   Z main_(void);
      // Defines the main work of the program.
      // The function has available data args_.
```

```
      // Defining code is allowed to rely on correct initialization of
      // graphical, font-related, and directory-related matters.
      // The details of this initialization mechanism are described in
      // cpmapplications.cpp as comments to function
      //    int main(int argc, char *argv[])

} // namespace

#endif
```

12 cpmapplication.cpp

```
//? cpmapplication.cpp
//? Status of work 2008-10-25.
//?
#include <cpmapplication.h>
#include <cpmrecordhandler.h>
#include <cpmgreg.h>
#include <cpmsystemdependencies.h>

using CpmRoot::Word;
using CpmRoot::Z;
using CpmRoot::R;
using CpmRootX::RecordHandler;
using CpmRootX::extComLine;
using CpmRootX::sup;
using CpmArrays::V;
using CpmSystem::Message;
using CpmSystem::updFromIniFile;

using namespace std;

void CpmSystem::updFromIniFile()
{
   Z mL=1;
   Word loc("CpmSystem::updFromIniFile()");
   CPM_MA
   Word fn0("cpmconfig.ini");
   Word fn1=CpmSystem::glue(Message::relPos1,fn0);
   Word fn2=CpmSystem::glue(Message::relPos2,fn0);
   Word fn3=CpmSystem::glue(Message::relPos3,fn0);
   Word fn4=CpmSystem::glue(Message::relPos4,fn0);
   V<Word> vc("",fn1,fn2,fn3,fn4);
   RecordHandler rch(vc,"","call in updFromIniFile()");
   if (!rch.isVal()){
      CPM_MZ
      return;
   }
   Z ped=1; // degree of pedantary
      // all data are initialized anyway, so 1 is OK

   Word sec="input directory";
   rch.read(sec,"dir",Message::inpDir,ped);

   sec="font directory";
   rch.read(sec,"dir",Message::fontDir,ped);

   sec="size of viewport";
   rch.read(sec,"width",Message::vpw_,ped);
```

```
   rch.read(sec,"height",Message::vph_,ped);

   sec="size of frame store";
   rch.read(sec,"width",Message::fsw_,ped);
   rch.read(sec,"height",Message::fsh_,ped);

   sec="pane sizes";
   rch.read(sec,"pane1",Message::sttBarWid1,ped);
   rch.read(sec,"pane2",Message::sttBarWid2,ped);
   rch.read(sec,"pane3",Message::sttBarWid3,ped);
   rch.read(sec,"pane4",Message::sttBarWid4,ped);

   sec="write precision";
   rch.read(sec,"val",CpmRoot::wrtPrc,ped);

   sec="write title";
   rch.read(sec,"val",CpmRoot::wrtTit,ped);

   sec="append run id";
   rch.read(sec,"val",Message::appRunId,ped);

#if !defined(CPM_NOLOG)
   sec="suppress messages";
   rch.read(sec,"val",Message::silent,ped);
#endif

   CPM_MZ
}

Word CpmApplication::title_("defaulttitle");

V<Word> CpmApplication::args_=V<Word>(0);

#include <cpmframe.h>

using CpmGraphics::Frame;

using CpmApplication::main_;
using CpmApplication::title_;
using CpmApplication::args_;

namespace{
// variables common to functions main and idleFunc
   Word loc_;
   Word argv_;
}

//////////////// int main(int argc, char *argv[]) ///////////////////////

#if !defined(CPM_NOGRAPHICS)
```

```
// main for application  which uses OpenGL and glu, glut (or OpenGlut)
// for output to screen

void displayFunc(void)
// everything is done by direct graphical commands
{
}
/* (from OpenGL Programming Guide, 2nd edition, page 601)
 If exact two-dimensional raterization is desired, you must
 carefully specify both the orthographic projection and the vertices
 of primitives that are to be rasterized.  The orthographic projection
 should be specified with integer coordinates, as shown in the
 following example:
      gluOrtho2D(0, width, 0, height);
 where width and height are the dimensions of the viewport.  Given
 this projection matrix, polygon vertices and pixel image positions
 should be placed at integer coordinates to rasterize predictably.
 For example, glRecti(0,0,1,1) reliably fills the lower left pixel of
 the viewport, and glRasterPos2i(0,0) reliably positions an unzoomed
 image at the lower left of the viewport.  Point vertices, line
 vertices, and bitmap positions should be placed at half-integer
 locations, however.  For example, a line drawn from (x1,0.5) to
 (x2,0.5) will be reliably rendered along the bottom row of pixels
 into the viewport, and a point drawn at (0.5,0.5) will reliably fill
 the same pixel as glRecti(0,0,1,1).

 An optimum compromise that allows all primitives to be specified at
 integer positions, while still ensuring predictable rasterization, is
 to translate x and y by 0.375, as shown in the following code
 fragment [reshapeCB()].  Such a translation keeps polygon and pixel
 image edges safely away from the centers of pixels, while moving line
 vertices close enough to the pixel centers.
*/

void glToglut(int w, int h)
// sets the glViewport and coordinate transformations of gl in a way that
// the concept of position in the image plane coincides with glut's
// definitions. The code and comments are from an code example 'canvas.c'
// that Thomas Dera found somewhere.
// Seems to be needed only in function reshape which will be only invoked
// at program begin.
{
  glViewport(0,0,w,h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0,w,0,h);
  glScalef(1.0,-1.0,1.0);   // this flips the coordinate system so
         // that y increases going down the
         // screen (see below)
  glTranslatef(0.0,-h,0.0);
```

```
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
   // glTranslatef(0.375,0.375,0.0); so we can draw using integer
   // coordinates (see above)  has no influence on my graphics functions

   glPixelStorei(GL_UNPACK_ALIGNMENT,1);
   // in order to intoduce no alignement errors depending on pixel
   // matrices having sizes that are a multiple of 4 or 8.;
   // value 4 (instead of our 1) is default
}

/*
 Note the flip of the y coordinate.  GLUT thinks the upper left hand
 corner of the screen or window is (0,0).  OpenGL, by default, thinks
 the lower left corner is (0,0).  This presents a problem when, for
 example, you compare the coordinates of a mouse event (from GLUT) with
 the coordinates of an OpenGL object.  The y coordinate flip makes OpenGL
 coordinates correspond to GLUT coordinates.
*/

void reshapeFunc(int w, int h)   // generic 2D reshape callback
{
   Word loc("reshapeFunc(int w, int h)");
   glToglut(w, h);
}

void idleFunc( void ) // calls main_ in a suitable context
// The function Z CpmApplication::main_(void) has to be provided by the
// user. In its definition, the following data are available:
// CpmGraphics::Viewport and thus CpmGraphics::Frame and
// CpmGraphics::Graph are initialized to the size of the graphics window
// the size of which is set in cpmconfig.ini.
// Together with the graphics classes the accompanying fonts are
// available. A name of the program is available as
// CpmApplication::args_[1]. More components of args_ are available only
// if provided by a calling command line or by reading a ini-file.
// When calling the present function the task of completing the
// command-line input by data from an ini-file is already done.
{
   cpmmessage("idleFunc() started");
   using CpmApplication::args_;
   Z nx=glutGet(GLUT_WINDOW_WIDTH);
   Z ny=glutGet(GLUT_WINDOW_HEIGHT);
   Word title=args_[1];
   title&=", CpmGraphics::Viewport initialized";
   CpmGraphics::Viewport vp(nx,ny,title);
      // initializes the viewport nx and ny as read from
      // cpmconfig.ini. No restriction to actual screensize
   Z vi=CpmGraphics::Viewport::isInitialized();
   if (vi==0) cpmerror(loc_&"not able to initialize Viewport");
```

```
   nx=CpmGraphics::Viewport::pelX();
   ny=CpmGraphics::Viewport::pelY();
   Z rank=CpmMPI::Cpm_com.getRank();
   Z size=CpmMPI::Cpm_com.getRank();
   if (size>1){
      Z sh=10*rank;
      Word wr("Rank "&cpm(rank)&" controls the viewport");
      vp.addText(100,sh,wr);
      cpmwait(0.5);
   }
   Word mes=loc_&" initialized Viewport with nx="&cpm(nx)&
      ", ny="&cpm(ny);
      // nx and ny may be much larger than the computers screen and
      // graphical hardware supports
   cpmmessage(mes);
   R tStart=cpmtime();
   Z res=main_(); // this is the main thing
   R tTot=cpmtime();
   tTot-=tStart; //now the name applies
   cpmmessage(loc_&"main_() done; res was "&cpm(res));
   cpmurgent("Main computation done in "&cpm(tTot)&" seconds");
   if (rank==1){
      std::ostringstream ost;
      ost<<"idleFunc() done, return value of main_()) is "<<res<<endl;
      ost<<"Function main() done for program "<<argv_;
      ost<<"Computation time was "<<tTot<<" seconds"<<endl;
      cpmmessage(ost,-1); // negative status bar segment writes to
      // console if there is one, even if graphics is active
   }
   R someTimeWait=0.5;
   // notice that a wait time for inspecting the final graphics can be
   // set with the *.ini-file of the main application so that there is no
   // need to have a long time here
   cpmwait(someTimeWait);
   CpmMPI::finalize();
   exit(res);
}

int main(int argc, char *argv[])
// Notice the C-native argument types int and char* which are chosen in
// order to not irritate some compilers.
// This is the main function for a console application. Under Windows it
// should also be startable by (double)clicking on the exe-file.
// Then nothing but the application's name gets transmitted to the present
// function, meaning that only args_[1] is then defined.
// When starting the program from the commandline, e.g. from the cygwin
// bash, e.g.
//    > ./myexcel2.exe 2
// one can provide it with further parameters for which the present
// function, however, does nothing more than storing them into the
```

```
// variabe CpmApplications::args_ of type V<Word>. They are then in place
// for the application-specific function main_ to make use of them.
// Notice that V<Word> is a univeral data type and that more convenient
// data formats comprising a type and an identifier are too complex to be
// convenient as command line input. They also have to follow specific
// rules to be readable and it is too easy to violate such rules while
// typing to the command line.

// In the present function there is a mechanism built in, which asks for
// information in a ini-file to be used to extend the command line
// content. This ini-file is expected to have the name of the executable
// with the file extension replaced by ".ini". (For this to work the
// executable needs not really have a file extension.) Also, if no
// ini-file can be found at any of the expected positions, the command
// line simply stays unchanged.
// If for some component of args_ there is input both from an ini-file
// and the command line, the latter alway is used.
// The 'expected positions' for the ini-file are explained in the
// comment to function CpmSystem::Message::ext(Word,Word) in cpmsystem.h.
// Notice that the expected positions for cpmconfig.ini are set by
// static data in C+- code and that the expected positions for ini-files
// depend on the base directory CpmSystem::Message::inpDir read from file
// cpmconfig.ini.

// Addionally, there here is a mechanism built in to obtain data for the
// size of a graphical application window and an internal frame store
// (which by default has the same size as the application window).
// These data are also expected from a file cmconfig.ini.

// It is the responsibility of function main_() to make contact to
// additonal files and ask for more data. See also idleFunc().
// Notice that messaging goes entirely through cpmmessage so that
// cout gets involved as defined in CpmSystem::Message.
{
   using CpmApplication::args_;
   using CpmGraphics::Viewport;
   using CpmGraphics::Frame;
   CpmMPI::initialize(&argc,&argv);
   CpmMPI::Cpm_com=CpmMPI::Com();
   Z size=CpmMPI::Cpm_com.getSize();
   Z rank=CpmMPI::Cpm_com.getRank();
   Word mes("C+- says Hello World in function ");
   mes&="int main(int argc, char *argv[])";
   cpmmessage(mes);
      // initializes the message streams cpmcerr and cpmdata
   if (rank==1){
      std::ostringstream ost;
      ost<<endl<<"Function main() started for program "<<argv[0]<<endl;
      ost<<"Number of processes is "<<size<<endl;
      cpmmessage(ost);
```

```
   }
   updFromIniFile(); // now cpmconfig.ini has beed read
   args_=extComLine(argc,argv);
   // Notice that this is the actual console input (although args_[1]
   // seems to get initialized to the full executable's name (i.e.
   // path included) also when the executable gets started by (double)
   // clicking on its representation in something like Windows Explorer).
   // extended by info from a ini-file if available. The name of the
   // ini-file is given by the program's name and for which
   // search positions are influenced by cpmconfig.ini.
   std::ostringstream ost2;
   ost2<<"Extended command line transformed to "<<args_;
   cpmmessage(ost2);
   argv_=args_[1];
   loc_=Word( "main(" );
   loc_=loc_&cpm(argc)&", "&args_[1]&"): ";
   glutInit( &argc, argv );
   glutInitDisplayMode( GLUT_SINGLE | GLUT_RGB );
#if defined(MINGW)
   Word gcw="Non-interactive OPENGLUT window for "&argv_;
#else
   Word gcw="Non-interactive GLUT window for "&argv_;
#endif
   glutCreateWindow(-gcw);
   int gdb=glutGet(GLUT_WINDOW_DOUBLEBUFFER);
   if (rank==1) cout<<"GLUT_WINDOW_DOUBLEBUFFER="<<gdb<<endl;
   Z dmp=glutGet(GLUT_DISPLAY_MODE_POSSIBLE);
   if (rank==1) cout<<"GLUT_DISPLAY_MODE_POSSIBLE="<<dmp<<endl;
   Z wbs=glutGet(GLUT_WINDOW_BUFFER_SIZE);
   if (rank==1) cout<<"GLUT_WINDOW_BUFFER_SIZE="<<wbs<<endl;
   if (rank==1) cpmwait(0.5);
   Z width=Message::w();
   Z height=Message::h();
   glutReshapeWindow(width,height);
#if defined(MINGW)
   glutPositionWindow(0,0);
#endif
   // is needed - otherwise the position is nearly random and renders the
   // status bar unreadable. Under Visual C++ there seems to be no
   // difference. Under Cygwin one looses the title bar of the main
   // window. The implementations of glut thus differ considerably.
   Z hSB=Viewport::getHeightStatusBar();
   Z widthFrmStr=Message::ws();
   Z heightFrmStr=Message::hs();
   Z heightFree=height-hSB;
   if (heightFrmStr<=height && heightFrmStr>heightFree){
      // then showing the bitmaps should fill the
      // available screen window as completely as possible
      heightFrmStr=heightFree;
   }
```

```
   Frame::setStaticSize(widthFrmStr,heightFrmStr);
   // The size of the system's display screen has no influence on this
   // action. Class' Frame internal frame store is set to this size
   // and if the display screen is smaller, it displays only a
   // part of the frame store. The whole frame store can always be
   // represented as a PPM-image file on disk.
   // This call also triggers loading the Cpm fonts, which will be used
   // by placing text to frame store content.
   glutReshapeFunc(reshapeFunc);
   glutDisplayFunc( displayFunc );
   glutIdleFunc( idleFunc );
      // this is the place where main_() gets called.
   glutMainLoop();
}

#else // no glut and openGL functionality needed. Graphics
// may be generated as *.ppm bitmaps of arbitrary
// pixel count (to be set in file cpmconfig.ini)

Z mainFunc( )
// calls main_ in a suitable context
{
   cpmmessage("mainFunc() started");
   Z res=main_();
   cpmmessage(loc_&"main_() done; res was "&cpm(res));
   cpmurgent("Computation done");
   R someTimeWait=0.5;
   // notice that a wait time for inspecting the final graphics can be
   // set with the *.ini-file of the main application so that there is no
   // need to have a long time here
   cpmwait(someTimeWait);
   cpmmessage("mainFunc() done, return value of main_()) is "
      &cpm(res));

   CpmMPI::finalize();
   return res;
}

int main(int argc, char *argv[])
// notice C-native argument types in order to not irritate the compiler.
// This is the main function for a application that under Windows should
// be startable by (double)clicking on the exe-file. Then nothing but
// the application's name gets transmitted to the present function
// (only args_[1] is the defined). For providing additional
// data such as the desired size of the application window,
// we read data from file cpmconfig.ini which is expected to
// be located in the same directory as the executable
{
   using CpmApplication::args_;
   using CpmSystem::Config;
```

```
   using CpmGraphics::Viewport;
   using CpmGraphics::Frame;
   CpmMPI::initialize(&argc,&argv);
   CpmMPI::Cpm_com=CpmMPI::Com();
   Z size=CpmMPI::Cpm_com.getSize();
   Z rank=CpmMPI::Cpm_com.getRank();
   cpmmessage("C+- says Hello World");
   if (rank==1){
      std::ostringstream ost;
      ost<<endl<<"Function main() started for program "<<argv[0]<<endl;
      ost<<"Number of processes is "<<size<<endl;
      cpmmessage(ost);
      cpmwait(1.);
   }
   updFromIniFile(); // now cpmconfig.ini has beed read
   args_=extComLine(argc,argv);
   std::ostringstream ost2;
   ost2<<"Extended command line transformed to "<<args_;
   cpmmessage(ost2);
   argv_=args_[1];
   loc_=Word( "main(" );
   loc_=loc_&cpm(argc)&", "&args_[1]&"): ";
   Z height=Message::w();
   Z hSB=Viewport::getHeightStatusBar();
   Z widthFrmStr=Message::ws();
   Z heightFrmStr=Message::hs();
   Z heightFree=height-hSB;
   if (heightFrmStr<=height && heightFrmStr>heightFree){
      // then showing the bitmaps should fill the
      // available screen window as completely as possible
      heightFrmStr=heightFree;
   }
   Frame::setStaticSize(widthFrmStr,heightFrmStr);
      // the actual screen size has no influence on this action
   Z res=mainFunc();
      // this is the place where main_() gets called.
   cpmmessage("function main() done for program "&argv_);
   return res;
}

#endif // defined(CPM_NOGRAPHICS)
```

13 cpmbas.h

```
//? cpmbas.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_BAS_H_
#define CPM_BAS_H_
/*
   Description: Collects header files from cpm0/include in one file
   in order to make working with basic C+- easier
*/
   #include <cpmtypes.h>
   #include <cpmfr.h>
   #include <cpmvr.h>
   #include <cpmsr.h>
   #include <cpmm.h>
   #include <cpmp.h>
   #include <cpmc.h>
   #include <cpmangle.h>
   #include <cpmgreg.h>
#endif
```

14 cpmbas.cpp

```
//? cpmbas.cpp
//? Status of work 2008-10-25.
//?
// see cpmbas.h

#include "cpmc.cpp"
#include "cpmangle.cpp"
#include "cpmvl.cpp"
#include "cpmgreg.cpp"
#include "cpmsystem.cpp"
#include "cpmtypes.cpp"
#include "cpmuc.cpp"
#include "cpmzinterval.cpp"
#include "cpmnumbers.cpp"
#include "cpmmpi.cpp"
#include "cpmword.cpp"
#include "cpmviewport.cpp"
```

15 cpmbasicinterfaces.h

```
//? cpmbasicinterfaces.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_BASIC_INTERFACES_H_
#define CPM_BASIC_INTERFACES_H_
/*

   Description: see cpminterfaces.h

*/

// The following makes sure that no client class will use Type in a way
// that would need copy constructor and assignment.
// Should be placed in the private section

#include <cpmbasictypes.h>
   // for Z in CPM_ORDER

#define CPM_INVAR(TypeName)\
   Type& operator = (const Type&);\
   TypeName(const Type&);

// example for usage
/*
   template <class X, class Y>
   class SUMFUO: public ...{
      typedef SUMFUO<X,Y> Type;
      CPM_INVAR(SUMFUO)
   public:
      ...
   };
*/

// order related stuff

#define CPM_ORDER_PLAIN\
   bool operator == ( const Type & x)const;\
   bool operator != (const Type & x)const;\
   bool operator < (const Type & x)const;\
   bool operator > (const Type & x)const;\
   bool operator <= (const Type & x)const;\
   bool operator >= (const Type & x)const;

// consistent generation of all six order-related operators from a
// single member functions. Notice that it would be useless to make com
// virtual since any useful re-definition probably differs in the
// type of the second argument too.
```

```
// see cpmnumers.h for explanations to 'com'

#define CPM_ORDER\
   CpmRoot::Z com(const Type&)const;\
   bool equalTo(const Type& x)const{ return com(x)==0;}\
   bool priorTo(const Type& x)const{ return com(x)==1;}\
   bool operator == ( const Type & x)const\
      { return 0==com(x);}\
   bool operator != (const Type & x)const\
      { return 0!=com(x);}\
   bool operator < (const Type & x)const\
      { return 0 < com(x);}\
   bool operator > (const Type & x)const\
      { return 0 > com(x);}\
   bool operator <= (const Type & x)const\
      { return 0 <= com(x);}\
   bool operator >= (const Type & x)const\
      { return 0 >= com(x);}

#endif
```

16 cpmbasictypes.h

```
//? cpmbasictypes.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_BASIC_TYPES_H_
#define CPM_BASIC_TYPES_H_
/*

   Description: Introduces mathematics style named
      aliases for the built-in types that the C+- class

*/
#include <cpmdefinitions.h> // for CPM_LONG

namespace CpmRoot{

// These are the basic types on which most of the C+- class hierarchy
// will be built.
// Their names are
//    Z, R, (L, N, Rh)
// Types in brackets occur only in special areas.
#if defined(CPM_LONG)
   typedef long int Z;
      // integer numbers
   typedef unsigned long int N;
      // natural numbers
      // important due to cyclic definition of addition
   typedef long double R;
      // real numbers
#else
   typedef int Z;
      // integer numbers
   typedef unsigned long int N;
      // natural numbers,
      // should always be long
      // Type N will be used only in implementation code,
      // where it is important that addition is a cyclic operation
      // which never creates overflow problems. As type of function
      // arguments I don't use N. The reason is the following:
      // Function arguments for which Z or N would be approriate types
      // occur frequently. Often, in cases that N seemed to be natural
      // whe defining a function first, it has to be discovered that
      // an extension to Z could be useful. So one would be tempted
      // so switch (or oscillate!) between using Z and N. This is is to
      // be avoided!

   typedef double R;
      // real numbers
```

```
#endif
   // The dependence on CPM_LONG was introduced August 2006.

// Less important types:

   typedef float Rh;
      // Real numbers of reduced ('half') accuracy, mainly to be
      // used in floating point image processing, where reducing the
      // size of large image matrix data by a factor 2 might be
      // desirable (less important today).

   typedef unsigned char L;
      // 'L' for letter represents a byte which
      // after 'integral promotion' is a value between 0 and 255.
      // Mainly needed for storing pixel values in image matrices.
}

#endif
```

17 cpmbody.h

```
//? cpmbody.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_BODY_H_
#define CPM_BODY_H_

//Description: see cpmbodyx.h

#include <vector>

#include <cpmv.h>
#include <cpmframes.h>
#include <cpmvectormore.h>
#include <cpmcompar.h>
   // includes cpmdimdef.h

#if defined(CPM_DIMDEF2)

#define CPM_DIM CpmDim2
#define CPM_CAMERA CpmCamera2
#define CPM_RIGID_BODY CpmRigidBody2
#define CPM_PHYSICS CpmPhysics2

namespace CPM_PHYSICS{
   #include <cpmbodyx.h>
}

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_RIGID_BODY
#undef CPM_PHYSICS

#endif

//////////////////////////////////////////////////////////////////////////

#if defined(CPM_DIMDEF3)

#define CPM_DIM CpmDim3
#define CPM_CAMERA CpmCamera3
#define CPM_RIGID_BODY CpmRigidBody3
#define CPM_PHYSICS CpmPhysics3

namespace CPM_PHYSICS{
   #include <cpmbodyx.h>
}
```

```
#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_RIGID_BODY
#undef CPM_PHYSICS

#endif

#endif
```

18 cpmbody.cpp

```
//? cpmbody.cpp
//? Status of work 2008-10-25.
//?
#include <cpmbody.h>
#include <cpmalgorithms.h>
#include <cpmr2func.h>

#if defined(CPM_DIMDEF2)

#define CPM_DIM CpmDim2
#define CPM_CAMERA CpmCamera2
#define CPM_RIGID_BODY CpmRigidBody2
#define CPM_PHYSICS CpmPhysics2

namespace CPM_PHYSICS{
   #include <cpmbodycpp.h>
}

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_RIGID_BODY
#undef CPM_PHYSICS

#endif

//////////////////////////////////////////////////////////////////////////

#if defined(CPM_DIMDEF3)

#define CPM_DIM CpmDim3
#define CPM_CAMERA CpmCamera3
#define CPM_RIGID_BODY CpmRigidBody3
#define CPM_PHYSICS CpmPhysics3

namespace CPM_PHYSICS{
   #include <cpmbodycpp.h>
}

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_RIGID_BODY
#undef CPM_PHYSICS

#endif
```

19 cpmbodycpp.h

```
//? cpmbodycpp.h
//? Status of work 2008-10-25.
//?
using namespace CpmRoot;
using namespace CpmProperties;
using namespace CpmImaging;
using namespace CpmArrays;
using namespace CpmAlgorithms;
using namespace CpmPhysics; // for physical constants

using CPM_DIM::Vec;
using CPM_DIM::AxVec;
using CPM_DIM::Spc;
using CPM_DIM::Group;
using CPM_DIM::RefSys;
using CPM_DIM::Plane;
using CPM_DIM::InTens;
using CPM_DIM::Rdim;
using CPM_DIM::Rdim_;
using CPM_DIM::Skeleton;
using CPM_DIM::WireFrame;

using CPM_CAMERA::Camera;

using CPM_RIGID_BODY::ComPar;
using CPM_RIGID_BODY::PointPar;
using CPM_RIGID_BODY::RigidObject;

using CpmFunctions::F;
using CpmFunctions::F1;
using CpmFunctions::F2;
using CpmFunctions::F4;
using CpmFunctions::F5;
using CpmFunctions::R2_Func;
using CpmGeo::Angle;

////////////////////// class ExtSys ////////////////////////////////////

ForTor ExtSys::interaction(const ComPar& cl , const R& dt)
{
   if (field_only) return ForTor();
   ForTor ft1; // felt by *this
   ForTor ft2; // felt by cl
   V< X4<Force,Spc,Force,Spc> > vx4=forceSpots(cl);
   Z i,j,n=vx4.dim();
   Spc x1=get_x();
   Spc x2=cl.get_x();
```

```
for (i=1;i<=n;i++){
   X4<Force,Spc,Force,Spc> x4=vx4[i];
   Force f1=   x4.c1();
   Spc   p1=   x4.c2();
   Force f2=   x4.c3();
   Spc   p2=   x4.c4();
   Spc pc=p1.cen(p2);
   ft1.add(f1,pc,x1);
   ft2.add(f2,pc,x2);
}
if (!SysControl::noInteractionChargeToWall){
   // interaction with mirror charges
   if (SysControl::singleCharge){
      PointCharge q=cl.getPointCharge();
      Spc xq=q.get_x();
      V<PointCharge> vpc=influence(q);
      Z n2=vpc.dim();
      for (j=1;j<=n2;j++){
         PointCharge qj=vpc[j];
         Force f1j=qj.force(q);
         Spc xj=qj.get_x();
         ft1.add(f1j,xj,x1);
         ft2.add(-f1j,xq,x2);
      }
   }
   else{
      V<PointPar> vp=cl.getConstituents();
      for (i=1;i<=n;i++){
         PointPar pi=vp[i];
         Spc xi=pi.get_x();
         PointCharge qi=pi.getPointCharge();
         V<PointCharge> vpci=influence(qi);
         Z n3=vpci.dim();
         for (j=1;j<=n3;j++){
            PointCharge qij=vpci[j];
            Force f1ij=qij.force(qi);
            Spc xij=qij.get_x();
            ft1.add(f1ij,xij,x1);
            ft2.add(-f1ij,xi,x2);
         }
      }
   }
}
Vec fcl=ft1.get_force();
Vec dMom=fcl*dt;
force+=fcl;
AxVec tcl=ft1.get_torque();
torque+=tcl;
AxVec dAngMom=tcl*dt;
momentum+=dMom; // was + instead of += till
```

```
   // 2005-03-22
   angularMomentum+=dAngMom;

   return ft2;
}

Force ExtSys::interaction(const PointPar& p, const R& dt)
{
   if (field_only) return Force();
   X2<Force,Spc> fs=forceSpot(p);
   Force fa=fs.c1();// fa is the force on the system *this
   Spc xa=fs.c2();
   Vec fp=fa.get_f();
   force+=fp;
   Vec dMom=fp*dt;
   AxVec tp=(xa-get_x()).cross(fp);
   torque+=tp;
   AxVec dAngMom=tp*dt;
   momentum+=dMom;
   angularMomentum+=dAngMom;
   return -fa;
}

R ExtSys::disVal(const ExtSys& b )const
{
   R_Vector a(6);
   a[1]=x_.disVal(b.x_);
   a[2]=v_.disVal(b.v_);
   a[3]=ori_.disVal(b.ori_);
   a[4]=omega_.disVal(b.omega_);
   a[5]=momentum.disVal(b.momentum);
   a[6]=angularMomentum.disVal(b.angularMomentum);
   return a.sup();
}

bool ExtSys::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("ExtSys",str);
   if (!RigidObject::prnOn(str)) return false;
   cpmp(active);
   cpmp(momentum);
   cpmp(angularMomentum);
   return true;
}

bool ExtSys::scanFrom(istream& str)
{
   if (!RigidObject::scanFrom(str)) return false;
   cpms(active);
   cpms(momentum);
```

```
   cpms(angularMomentum);
   return true;
}

//////////////////////////////// class Body ////////////////////////////
namespace{
const R facEl=sqrt(2.)/3;
// factor appearing in Hertz' formula
const R facLarge=1e8;
} // anon namespace

X4<Spc,Spc,Vec,R> Body::clsApp(PointPar const& pa)const
// since only the functions get_x() and get_r() of the particle
// are needed this is given by the same code as for SphPar
{
   Spc px=pa.get_x();
   X2<R,Vec> prv=distAndDirection(px);
      // distAndDirection(...) defined in cpmbodyx.h
      // as shape.distAndDirection(x) which is defined in
      // cpmdimxcpp.h as
      // SubsetOfSpace::distAndDirection(const Spc& x)const
   R d=prv.first();
   R ra=pa.get_r();
   R gap=d-ra;
   Vec n=prv.second();
   Spc res1=px-d*n;
   Spc res2=px-ra*n;
   return X4<Spc,Spc,Vec,R>(res1,res2,n,gap);
}

R Body::gap(PointPar const& pa)const
{
   Spc px=pa.get_x();
   X2<R,Vec> prv=distAndDirection(px);
   R d=prv.first();
   R ra=pa.get_r();
   return d-ra;
}

R Body::gap(ComPar const& cp)const
{
   V<PointPar> vsp=cp.getConstituents();
   Z i,np=vsp.dim();
   R_Vector gaps(np);
   for (i=1;i<=np;++i) gaps[i]=gap(vsp[i]);
   return gaps.inf();
}

V< X4<Force,Spc,Force,Spc> > Body::forceSpots(const ComPar& cl)const
   // force which a particle feels from the Body and which is not
```

```
   // mediated by data member field.
{
   Word loc("Body::forceSpots(ComPar)");
   if (!active || fieldOnly()) return V< X4<Force,Spc,Force,Spc> >(0);
   V<PointPar> vsp=cl.getConstituents();
   Z i,np=vsp.dim();
   V< X4<Force,Spc,Force,Spc> > res(np);
   R rp=cl.get_r();
   if (rp<pinhole_) return res;

   ContactProperties cp1=bodyProp;
   for (i=1;i<=np;i++){
      PointPar pi=vsp[i];
      X4<Spc,Spc,Vec,R> x4=clsApp(pi);
      R h=x4.c4(); // for the gap and many following quantities
         // the dependence on i is suppressed in the notation
      ContactProperties cp2=pi.get_cp();
      R r2=pi.get_r();
      R r1=facLarge*r2;
         // the Body *this is considered flat as far as Hertz formula
         // for the elastic contribution is considered
      if (h>cl.get_r()+2*r2) return res;
      if (h>0) continue; // avoid adoo if not needed
      Spc x1=x4.c1();
      Spc x2=x4.c2();
      Vec n=x4.c3();
      Vec v1=get_v(x1);
      Vec v2=cl.get_v(x2);
      Force f=PointPar::forceFrictionAdhesionElastic(
         h,n,r1,r2,v1,v2,cp1,cp2);
      res[i]=X4<Force,Spc,Force,Spc>(f,x1,-f,x2);
   }
   return res;
}

R Body::potElastic(const ComPar& p)const
{
   Word loc("Body::pot(ComPar)");
   Z mL=3;
   CPM_MA
   R res=0;
   if (!active || SysControl::noElasticRepulsion
      || SysControl::ignoreExternalSystems || fieldOnly()) return res;
   R rp=p.get_r();
   if (rp<pinhole_) return res;
   V<PointPar> vsp=p.getConstituents();
   Z i,np=vsp.dim();
   for (i=1;i<=np;i++){
      PointPar pi=vsp[i];
      res+=Body::potElastic(pi);
```

```
   }
   CPM_MZ
   return res;
}


X2<Force,Spc> Body::forceSpot(const PointPar& p)const
{
   X2<Force,Spc> res;
   if (!active || fieldOnly()) return res;
      // added 2002-04-19 in order to be able to 'switch off'
      // the body effectively
   R rp=p.get_r();
   if (rp<pinhole_) return res;

   ContactProperties cp1=bodyProp;
   ContactProperties cp2=p.get_cp();

   X4<Spc,Spc,Vec,R> x4=clsApp(p);
   R h=x4.c4(); // for the gap and many following quantities
         // the dependence on i is suppressed in the notation
   R r2=p.get_r();
 //  const R largeFac=1e8;
   R r1=facLarge*r2;
      // the Body *this is considered flat as far as Hertz formula
      // for the elastic contribution is considered
   if (h>0) return res;
   Spc x1=x4.c1();
   Spc x2=x4.c2();
   Spc xc=x1.cen(x2);
   Vec n=x4.c3();
   Vec v1=get_v();
   Vec v2=p.get_v();
   Force f=PointPar::forceFrictionAdhesionElastic(
      h,n,r1,r2,v1,v2,cp1,cp2);
   return X2<Force,Spc>(f,xc);
}

R Body::potElastic(const PointPar& p)const
{
   R res=0;
   if (!active || fieldOnly()) return res;
      // added 2002-04-19 in order to be able to 'switch off'
      // the body effectively
   R rp=p.get_r();
   if (rp<pinhole_) return res;

   ContactProperties cp1=bodyProp;
   ContactProperties cp2=p.get_cp();
```

```
   X4<Spc,Spc,Vec,R> x4=clsApp(p);
   R h=x4.c4(); // for the gap and many following quantities
         // the dependence on i is suppressed in the notation
   R r2=p.get_r();
 //  const R largeFac=1e8;
   R r1=facLarge*r2;
      // the Body *this is considered flat as far as Hertz formula
      // for the elastic contribution is considered
   if (h>0) return res;
   Vec n=x4.c3();
   R pot=PointPar::elasticPotential(h,n,r1,r2,cp1,cp2);
   return pot;
}

bool Body::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("Body",str);
   if (!ExtSys::prnOn(str)) return false;
   cpmp(bodyProp);
   return true;
}

bool Body::scanFrom(istream& str)
{
   if (!ExtSys::scanFrom(str)) return false;
   cpms(bodyProp);
   setColor();
      // color is a dependent quantity that gets not written to file
   return true;
}

//////////////////////////// class HomFil /////////////////////////

void HomFil::update()
{
   RigidObject::update();
   FieldSpace fsUpd(hfd_.fE_,hfd_.fBH_,hfd_.fG_);
   VectorField vf(fsUpd);
   setField(vf);
   ScalarField sf(fsUpd);
   setU(sf);
   Group g=frmMasSys();
   field*=g;
   fieldU*=g;
}

bool HomFil::prnOn(ostream& str)const
{
   Z mL=2;
   Word loc("HomFil::prnOn(...)");
```

```
   CPM_MA
   CpmRoot::writeTitle("HomFil",str);
   if (!ExtSys::prnOn(str)) return false;
   cpmp(hfd_);
   CPM_MZ
   return true;
}

bool HomFil::scanFrom(istream& str)
{
   Z mL=2;
   Word loc("HomFil::scanFrom(...)");
   CPM_MA
   if (!ExtSys::scanFrom(str)) return false;
   cpms(hfd_);
   update();
   CPM_MZ
   return true;
}

//////////////////// class VibData ////////////////////////////////

bool VibData::prnOn(ostream& str)const
{
   cpmwat; // write automatic title
   cpmp(amp_);
   cpmp(omg_);
   cpmp(del_);
   return true;
}

bool VibData::scanFrom(istream& str)
{
   cpms(amp_);
   cpms(omg_);
   cpms(del_);
   return true;
}

Group VibData::vib(R t)const
{
   Vec shift;
   if (amp_[1]!=0) shift[1]=amp_[1]*sin(omg_[1]*t+del_[1]);
   if (amp_[2]!=0) shift[2]=amp_[2]*sin(omg_[2]*t+del_[2]);
   if (amp_[3]!=0 && Spc::dimension()>2)
      shift[3]=amp_[3]*sin(omg_[3]*t+del_[3]);
   return Group(shift);
}

VibData::VibData(Record const& rec, Word const& sec)
```

```
{
   Z dim=Spc::dimension();
   Z ped=0; // we do not assume vibration data to be provided
   V<R> amp(dim);
   rec.get(sec,"vibAmp",amp,ped);
   V<R> frq(dim);
   rec.get(sec,"vibFrq",frq,ped);
   V<R> phs(dim);
   rec.get(sec,"vibPhs",phs,ped);
   amp_=Rdim(amp);
   omg_=Rdim(frq);
   del_=Rdim(phs);
   omg_*=cpm2pi;
   del_*=cpm2pi;
}

//////////////////////// class Pot ////////////////////////////////////

R potFunc(Rdim const& x, PotData const& vd)
// a combinations of half-space contributions, no attempt to smoothing
// out the edges
{
   bool sw=vd.getSideWalls();
   Rdim sh=vd.s*0.5;
   R res=x[2]+sh[2];
   if (sw){
      R h1L=x[1]+sh[1];
      R h1U=-x[1]+sh[1];
      res=CpmRootX::inf<R>(h1L,h1U,res);
   }
   if (Spc::dimension()==3){
      R h3L=x[3]+sh[3];
      R h3U=-x[3]+sh[3];
      res=CpmRootX::inf<R>(h3L,h3U,res);
   }
   return res;
}

void Pot::upd()
{
   Z nCorners=(Spc::dimension()==3 ? 8 : 4);
   corners=V<Spc>(nCorners);
   Rdim s=dat.s*0.5;
   R s1=s[1];
   R s2=s[2];
   R s3=s[3];
   if (nCorners==4){
      Rdim c1(-s1,s2);
      Rdim c2(-s1,-s2);
      Rdim c3(s1,-s2);
```

```
      Rdim c4(s1,s2);
      corners[1]=Spc(c1);
      corners[2]=Spc(c2);
      corners[3]=Spc(c3);
      corners[4]=Spc(c4);
   }
   else{
      Rdim c1(-s1,s2,-s3);
      Rdim c2(-s1,-s2,-s3);
      Rdim c3(s1,-s2,-s3);
      Rdim c4(s1,s2,-s3);
      Rdim c5(-s1,s2,s3);
      Rdim c6(-s1,-s2,s3);
      Rdim c7(s1,-s2,s3);
      Rdim c8(s1,s2,s3);
      corners[1]=Spc(c1);
      corners[2]=Spc(c2);
      corners[3]=Spc(c3);
      corners[4]=Spc(c4);
      corners[5]=Spc(c5);
      corners[6]=Spc(c6);
      corners[7]=Spc(c7);
      corners[8]=Spc(c8);
   }
   RigidObject::update();
   F<Rdim,R> vsf=F1<Rdim,PotData,R>(dat)(potFunc);
   shape.setShape(vsf);
   shape.set_dr(SysControl::singRange); // factor experiment
   R shpr=0.5*dat.s.absVal();
   set_r(shpr);
   shape.set_ds(shpr*0.002);
   Group g=frmMasSys();
   Group gVib=vd.vib(get_t());
   g*=gVib;
   shape*=g;
   for (Z i=1;i<=nCorners;++i) corners[i]*=g;
   shape.set_x(x_);
   shape.set_r(r_);
}

bool Pot::prnOn(ostream& str)const
{
   Z mL=2;
   Word loc("Pot::prnOn(...)");
   CPM_MA
   cpmwat;
   if (!Body::prnOn(str)) return false;
   cpmp(dat);
   cpmp(vd);
   CPM_MZ
```

```
   return true;
}

bool Pot::scanFrom(istream& str)
{
   Z mL=2;
   Word loc("Pot::scanFrom(...)");
   CPM_MA
   if (!Body::scanFrom(str)) return false;
   cpms(dat);
   cpms(vd);
   Pot::upd();
   CPM_MZ
   return true;
}

void Pot::mark(const Camera& sc)const
{
   if (SysControl::ignoreExternalSystems) return;
   if (active ){
      sc.mark(corners[1],corners[2],color);
      sc.mark(corners[2],corners[3],color);
      sc.mark(corners[3],corners[4],color);
      if (Spc::dimension()==3){
         sc.mark(corners[4],corners[1],color);
         sc.mark(corners[5],corners[6],color);
         sc.mark(corners[6],corners[7],color);
         sc.mark(corners[7],corners[8],color);
         sc.mark(corners[8],corners[5],color);
         sc.mark(corners[1],corners[5],color);
         sc.mark(corners[4],corners[8],color);
         sc.mark(corners[2],corners[6],color);
         sc.mark(corners[3],corners[7],color);
      }
   }
}

#define CPM_SC\
   cpmassert(i>0 && i<=3,loc);\
   if (i>Spc::dimension()) return true;\
   else{\
      Spc x=cl.get_x();\
      R h=get_h();\
      Iv ivScaled(h*iv[1],h*iv[2]);\
      return ivScaled.contains(x[i]);\
   }

bool Pot::boxSel(const ComPar& cl, Iv iv, Z i)
{
   static Word loc("Pot::boxSel(ComPar,Iv,Z)");
```

```
   CPM_SC
}

bool Pot::boxSel(const PointPar& cl, Iv iv, Z i)
{
   static Word loc("Pot::boxSel(PointPar,Iv,Z)");
   CPM_SC
}

#undef CPM_SC

//////////////////////// class Lid ////////////////////////////////////
namespace{
R lidFunc(Rdim const& x, R const& h0, B const& isBottom)
{
   if (isBottom) return x[2]-h0;
   else return h0-x[2] ;
}
} // namespace

void Lid::upd()
{
   Z mL=3;
   Word loc("Lid::upd()");
   CPM_MA
   Z nCorners=2;
   corners=V<Spc>(nCorners);
   R s=get_r();
   Rdim c1(-s,dat.y0,0);
   Rdim c2(s,dat.y0,0);
   corners[1]=Spc(c1);
   corners[2]=Spc(c2);

   RigidObject::update();
   F<Rdim,R> vsf=F2<Rdim,R,B,R>(dat.y0,dat.isBottom)(lidFunc);
   shape.setShape(vsf);
   shape.set_dr(SysControl::singRange);
   R shpr=dat.iv1.radius();
   set_r(shpr);
   shape.set_ds(shpr*0.002);
   Group g=frmMasSys();
   Group gVib=vd.vib(get_t());
   g*=gVib;
   shape*=g;
   for (Z i=1;i<=nCorners;++i) corners[i]*=g;
 //  field=VectorField(FieldE(),FieldBH(),FieldG());
   shape.set_x(x_);
   shape.set_r(r_);
   CPM_MZ
}
```

```
bool Lid::prnOn(ostream& str)const
{
   Z mL=2;
   Word loc("Lid::prnOn(...)");
   CPM_MA
   cpmwt("Lid");
   if (!Body::prnOn(str)) return false;
   cpmp(dat);
   cpmp(vd);
   CPM_MZ
   return true;
}

bool Lid::scanFrom(istream& str)
{
   Z mL=2;
   Word loc("Lid::scanFrom(...)");
   CPM_MA
   if (!Body::scanFrom(str)) return false;
   cpms(dat);
   cpms(vd);
   upd();
   CPM_MZ
   return true;
}

void Lid::mark(const Camera& sc)const
{
   Z mL=3;
   Word loc("Lid::mark(Camera)");
   CPM_MA
   if (SysControl::ignoreExternalSystems) return;
   if (active ){
      sc.mark(corners[1],corners[2],color);
   }
   CPM_MZ
}

//////////////////////// class Fen ////////////////////////////////////

R fenFunc(Rdim const& x, R const& r, V<R> const& cenX)
{
   Z n=cenX.dim();
   R_Vector dis(n);
   R dy=x[2];
   R dy2=dy*dy;
   for (Z i=1;i<=n;++i){
      R dxi=x[1]-cenX[i];
      dis[i]=::sqrt(dxi*dxi+dy2)-r;
```

```
   }
   return dis.inf();
}

void Fen::upd()
{
   Z mL=2;
   Word loc("Fen::upd()");
   CPM_MA
   RigidObject::update();
   //active=;
   R r=dat.r;
   R g=dat.g;
   Iv iv=dat.iv1;
   R l=iv.absVal();
   Z nBars=cpmround(1+l/(2*r+g));
   cenX=iv.div(nBars);
   r=0.5*(l/(nBars-1)-g);
      // r gets adjusted so that the fence fits iv
      // (the endpoints of iv are centers of pickets
      // since this promisses smoother motion of particles
      // at the regions where the fence contacts the pot.
   F<Rdim,R> vsf=F2<Rdim,R,V<R>,R>(r,cenX)(fenFunc);
   shape.setShape(vsf);
   shape.set_dr(SysControl::singRange);
   R shpr=0.5*l;
   set_r(shpr);
   shape.set_ds(shpr*0.002);
   Group gr=frmMasSys();
   shape*=gr;
   field=VectorField(FieldE(),FieldBH(),FieldG());
   shape.set_x(x_);
   CPM_MZ
}

bool Fen::prnOn(ostream& str)const
{
   Z mL=2;
   Word loc("Fen::prnOn(...)");
   CPM_MA
   CpmRoot::writeTitle("Fen",str);
   if (!Body::prnOn(str)) return false;
   cpmp(dat);
   CPM_MZ
   return true;
}

bool Fen::scanFrom(istream& str)
{
   Z mL=2;
```

```
   Word loc("Fen::scanFrom(...)");
   CPM_MA
   if (!Body::scanFrom(str)) return false;
   cpms(dat);
   Fen::upd();
   CPM_MZ
   return true;
}

void Fen::mark(const Camera& sc)const
{
   if (SysControl::ignoreExternalSystems) return;
   if (active){
      Z i,n=cenX.dim();
      if (Spc::dimension()==2){
         for (i=1;i<=n;++i){
            Skeleton ski=Skeleton::circle(Spc(cenX[i],0),dat.r);
            sc.mark(ski,color);
         }
      }
      else{
         Z circPoints=10;
         R h=dat.iv3.absVal();
         R z0=dat.iv3.inf();
         Skeleton cy0=Skeleton::cylinder(dat.r,h,circPoints);
         for (i=1;i<=n;++i){
            Skeleton cyi=cy0;
            Vec shi;
            shi[1]=cenX[i];
            shi[3]=z0;
            cyi+=shi;
            sc.mark(cyi,color);
         }
      }
   }
}

//////////////////////// class SphTank ////////////////////////////////////

R sphFunc(Rdim const& x, SphTankData const& vd)
{
   return vd.r_-x.absVal();
      // value increases from outside to inside since
      // the body is ouside its spherical surface
}

void SphTank::upd()
{
   RigidObject::update();
   F<Rdim,R> vsf=F1<Rdim,SphTankData,R>(dat_)(sphFunc);
```

```
   shape.setShape(vsf);
   shape.set_dr(SysControl::singRange); // factor experiment
   R shpr=dat_.r_;
   set_r(shpr);
   shape.set_ds(shpr*0.002);
   Group g=frmMasSys();
   shape*=g;
   shape.set_x(x_);
}

bool SphTank::prnOn(ostream& str)const
{
   Z mL=2;
   Word loc("SphTank::prnOn(...)");
   CPM_MA
   cpmwt("SphTank");
   if (!Body::prnOn(str)) return false;
   cpmp(dat_);
   CPM_MZ
   return true;
}

bool SphTank::scanFrom(istream& str)
{
   Z mL=2;
   Word loc("SphTank::scanFrom(...)");
   CPM_MA
   if (!Body::scanFrom(str)) return false;
   cpms(dat_);
   SphTank::upd();
   CPM_MZ
   return true;
}

void SphTank::mark(const Camera& sc)const
{
   if (SysControl::ignoreExternalSystems) return;
   if (active ){
      Spc x=get_x();
      Spc ori;
      Vec sh=x-ori;
      R r=dat_.r_;
      WireFrame gl=WireFrame::globe(r);
      gl*=Group(sh);
      sc.mark(gl,Color(STEREO));
   }
}

//////////////////////// some local tools ////////////////////
```

```
namespace{

   R profFunc(C const& z, R1 const& k1, R1 const& k2, R1 const& del1,
      R1 const& del2, R1 const& amp)
   { return amp*cos(z[1]*k1+del1)*cos(z[2]*k2+del2);}

// form function for class Drum
   R fDrum(Rdim const& x, R1 const& r0,
      B const& hollow, R2_Func const& prof)
   {
      C z(x[1],x[2]); // experiment
      R x2=Spc::dimension()==3 ? x[3] : 0;
      R r,phi;
      z.toPolar(r,phi);
      C z1(r0*phi,x2);
      // first is a peripheral coordinate along the circum-
      // pherence of the cylinder
      R res=r-r0-prof(z1);
      if (hollow) res=-res;
      return res;
   }

} // namespace

DrumData::DrumData(RecordHandler const& rch, Word const& sec, Z ped)
{
   rch.read(sec,"amp",amp,ped);
   rch.read(sec,"pointsPerHalfwave",pointsPerHalfwave);
   rch.read(sec,"k1",k1,ped);
   rch.read(sec,"del1",del1,ped);
   rch.read(sec,"k3",k3,ped);
   rch.read(sec,"del3",del3,ped);
   rch.read(sec,"r0",r0,ped);
   rch.read(sec,"omg",omg,ped);
   rch.read(sec,"hollow",hollow,ped);
}

//////////////////////// class Drum ////////////////////////////////////

void Drum::upd()
{
   Z mL=3;
   Word loc("Drum::upd()");
   CPM_MA
   RigidObject::update();
   if (drd.r0<=0){
      active=0;
      CPM_MA
      return;
   }
```

```
   else active=1;
// First we have to make the wavelength arround the circumference
// equal to an integer faction of it.
   Z nk1=cpmround(drd.r0*drd.k1);
   drd.k1=nk1/drd.r0; // corrected value such that a integer number
      // of waves makes up the circumference of the circle of
      // radius drd.r0
   R nk3=drd.k3*drd.r0;
   R nk=cpmsup<R>(nk1,nk3);
   nP=cpmround(nk*drd.pointsPerHalfwave);
   R2_Func prf=F5<C,R1,R1,R1,R1,R1,R>(drd.k1,drd.k3,drd.del1,drd.del3,
      drd.amp)(profFunc);
   F<Rdim,R> fif=F3<Rdim,R1,B,R2_Func,R>(drd.r0,drd.hollow,prf)(fDrum);
   shape.setShape(fif);
   R drFine=1e-3*drd.r0;
   shape.set_ds(drFine);
   R drCoarse=20*drFine+0.5*drd.amp;
   shape.set_dr(drCoarse);
   Group g=frmMasSys();
   Group gVib=vd.vib(get_t());
   g*=gVib;
   shape*=g;
      // it is important that the data member rs_ of RigidObject does
      // not move as a consequence of the vibration. Otherwise the
      // vibration shift could not be defined relative to this frame.
   shape.set_x(x_);
   shape.set_r(r_);
   CPM_MZ
}

bool Drum::prnOn(ostream& str)const
{
   cpmwat;
   if (!Body::prnOn(str)) return false;
   cpmp(drd);
   cpmp(vd);
   return true;
}

bool Drum::scanFrom(istream& str)
{
   if (!Body::scanFrom(str)) return false;
   cpms(drd);
   cpms(vd);
   Drum::upd(); // sets nP
   return true;
}

void Drum::mark(const Camera& sc)const
{
```

```
   if (active && !SysControl::ignoreExternalSystems){
     AxVec ax=sc.getAxis();
     Spc p=shape.get_x();
     sc.markFan(shape,p,sc.get_v(),ax,cpm2pi,nP,color);
   }
}
```

20 cpmbodyx.h

```
//? cpmbodyx.h
//? Status of work 2008-10-25.
//?
/*
Description: External Systems, an important building block for
   Heterogeneous systems

  Design consideration: The aim is to organize the interaction of
  subsystems to form a larger system

*/

   using CpmRoot::R;
   using CpmRoot::Z;

   using CpmRootX::Z1;
   using CpmRootX::R1;
   using CpmRootX::B;
   using CpmRootX::Record;
   using CpmRootX::RecordHandler;

   using CpmGeo::Angle;
   using CpmGeo::Iv;

   using CpmFunctions::F;
   using CpmFunctions::F1;

   using CpmArrays::V;
   using CpmArrays::X4;
   using CpmArrays::R_Vector;

   using CpmLinAlg::R2;

   using CpmGraphics::Graph;
   using CpmGraphics::Frame;
   using CpmGraphics::Frames;
   using CpmGraphics::SitRec;
   using CpmGraphics::UPP;
   using CpmGraphics::LOW;
   using CpmGraphics::LEFT;
   using CpmGraphics::RIG;

   using CpmImaging::Color;

   using CpmProperties::BodyProperties;
   using CpmProperties::ParticleProperties;
```

```
   using CPM_DIM::Group;
   using CPM_DIM::Spc;
   using CPM_DIM::Vec;
   using CPM_DIM::AxVec;

   using CPM_DIM::Plane;
   using CPM_DIM::SubsetOfSpace;
   using CPM_DIM::Sphere;
   using CPM_DIM::Rdim;
   using CPM_DIM::Skeleton;

   using CPM_RIGID_BODY::RigidObject;
   using CPM_RIGID_BODY::PointPar;
   using CPM_RIGID_BODY::ComPar;

   using CPM_CAMERA::Camera;

//doc namespace CpmPhysicsx{

//////////////////////////// class ExtSys ////////////////////////////////

class ExtSys: public RigidObject{ // external systems
   // uses Spc, Vec, AxVec, Force, ForTor,
   // VectorField; depends on SphPar, Graph, Camera
   // RigidObject defined in cpmcomparx.h

// External systems for confining, shaping, transporting systems of
// carrier and toner particles should be instances of classes which are
// derived from this class. This will allow the template class
// ParDynSys<> to execute the proper dynamics of the combined system.

protected:
   // The class is made to define derived classes; so it looks
   // acceptable to have protected data.

   Z1 active;
      // If active==0, the system does not interact with particles
      // and is invisible on screen.

   Z1 field_only;
      // If fieldOnly!=0, the system interacts with particles only via
      // field and not via interaction() i.e. the function interaction
      // is defined such that it can be called without creating
      // computational burden.
      // Type was Z instead of Z1 and initialisation of field_only was
      // missing in the ExtSys default constructor till 2000-4-7.
      // Terrible consequences!!!

   Vec momentum;
      // holds the time-integral of the forces applied to the system:
```

```
      // not necessarily mass times velocity (in most cases mass is
      // treated as infinitely large)

   AxVec angularMomentum;
      // holds the time-integral of the torques relative to x applied
      // to the system

   Vec force;
      // accumulates all forces acting on *this as a result of
      // calling interaction(...). In order to sum up only those
      // interactions that correspond to a physical state, one has
      // to call clear() at the right time. See the implementation
      // of function steplet(...) in file cpmpardynsys.h for proper
      // usage
   AxVec torque;
      // see function force: accumulates all torques ....

   ScalarField fieldU;

   VectorField field;
      // This is a field which the system creates and which may influence
      // other systems

    void transformStreamableData(const Group& g)
      // transforms the streamable data from ExtSys which are not
      // invariant under group action (so we don't transform the
      // parameter active)
   {
      RigidObject::transformStreamableData(g);
      momentum*=g;
      angularMomentum*=g;
   }

// group action to transform the function objects (non-streamable data)
// This operation may not be useful in derived classes since there the
// transformation of field has to be defined from transformed field
// sources which can be described by a streamable data type

    virtual void transformNonStreamableData(const Group& g)
      {field*=g; fieldU*=g;}
      // transforms the streamable data from ExtSys which are not
      // invariant under group action (so we don't transform the
      // parameter active)

// no redefinition of update()

public:
   typedef ExtSys Type;
   CPM_IO
      // I/O for the streamable part of data
```

```
   CPM_NAM(ExtSys)

   virtual RigidObject* clone()const{ return new ExtSys(*this);}

   ExtSys(void){}
   explicit ExtSys(const RigidObject& ro,
         const VectorField& vf=VectorField()):
            RigidObject(ro),active(1),field(vf){}

   Vec get_momentum()const{ return momentum;}

   AxVec get_angularMomentum()const{ return angularMomentum;}

   Vec getMeanForce()const{ return momentum*cpminv(t_);}

// new functions to registrate the weight on bodies (2005-06-13)
   Vec getForce()const{ return force;}
      // see explanation for 'force'
   AxVec getTorque()const{ return torque;}
      // see explanation for 'torque'
   void clear(){ force=Vec(); torque=AxVec();}
      // see explanation for 'force'

   void setActivity(Z act){active=act;}

   bool fieldOnly()const{ return field_only!=0;}

   void makeFieldOnly(){ field_only=Z1(1);}

   virtual R dist(const Spc& x)const{ x;return 0;}
      // will be defined in a useful manner in derived classes

   virtual V<PointCharge> influence(const PointCharge& q)const
      // mirror charge created by q when interacting with *this
      // has to be redefined if body is metallic or dielectric
      { q; return V<PointCharge>(0);}

   virtual  V< X4<Force,Spc,Force,Spc> > forceSpots(
      const ComPar& part)const
      // List of forces which a component particles feel from the Body
      // (and which are not mediated by data member field) together with
      // the
      // 'contact points' on the side of the component particles and the
      // system *this.
      // This function defines class behavior. In order to define this
      // function, one has to define the shape of the system and the
      // mechanisms that generate the forces.
      // alse see implementation of Body::forceSpots(...)
   {  part; return V< X4<Force,Spc,Force,Spc> >(0);}
```

```
   virtual  X2<Force,Spc> forceSpot(const PointPar& part )const
      // since the two contact points (this on part and the one
      // on *this) have to be united anyway, and also the force
      // on part and *this determine each other (their sum is 0)
      // we give here not X4<Force,Spc,Force,Spc> but
      // simply X2<Force,Spc>
      // also see implementation of Body::forceSpot(...)
   { part; return X2<Force,Spc>();}

   ForTor interaction(const ComPar& cl, const R& dt);
      // returns the force and torque excerted on a particle cl
      // and adds to the data members momentum and angularMomentum of
      // *this the contribution corresponding to having the reaction
      // forces acting for a time span dt on the system.

   Force interaction(const PointPar& p, const R& dt);
      // returns the force (torque=0) excerted on a particle p
      // and adds to the data members momentum and angularMomentum of
      // *this the contribution corresponding to having the reaction
      // forces acting for a time span dt on the system.

   virtual R potElastic(PointPar const& p)const{ p; return 0;}
      // potential energy of p due to interaction with the body

   virtual R potElastic(ComPar const& p)const{p; return 0;}
      // this functions potElastic will be defined meaningfully in Body,
      // where a shape is defined

// accessing the field values, from the general to the specific

   VectorField getField(void)const{ return field;}
   ScalarField getU(void)const{ return fieldU;}

   FieldSpace getField(Spc const& x)const{ return field(x);}
   R getU(Spc const& x)const{ return fieldU(x);}

   VecE getE(Spc const& x)const{ return field(x).c1();}
   VecBH getBH(Spc const& x)const{ return field(x).c2();}
   VecG getG(Spc const& x)const{ return field(x).c3();}

   void setField(VectorField const& f){field=f;}
   void setU(ScalarField const& f){fieldU=f;}

// potential energy of a particle due to the field member

   virtual R potField(PointPar const& p)const
   { return p.pot(getField(p.get_x()));}
      // potential energy of p due to interaction with the
      // electro-magnetic-gravitational field generated by object *this.
      // The formula is exact only if the field is homogeneous
```

```
      // Since the particle has no magnetic moment, there is no
      // contribution of the magnetic field
   virtual R potField(ComPar const& p)const
   { return p.pot(getField(p.get_x()));}
      // potential energy of p due to interaction with the
      // electro-magnetic-gravitational field generated by object *this.
      // The formula is exact only if the field is homogeneous
      // Since the particle has a magnetic moment, there is a contribution
      // of the magnetic field

  // miscellaneous
   virtual R disVal(const ExtSys& )const;
      // this function can be meaningfully defined yet on this level

   virtual void mark(const Camera& sc)const{sc;}
      // visualizing the field of an idividual body is not useful
      // since the field of all external bodies has to summed up an shown
      // as one entity. The shape, that should be shown, is not yet
      // defined in this class

   bool isActive()const{ return active!=0;}

   void setAct(bool act){active = (act==true  ? Z1(1) : Z1(0));}

// list of new virtual functions that are needed to let
// derivations of this class act as external systems in
// class Station

virtual bool boxSel(const ComPar& cl, Iv iv, Z i=2)
{ cl; iv; i; return false;}
      // box selector, first valid implementation in class Pot

virtual bool boxSel(const PointPar& cl, Iv iv, Z i=2)
{ cl; iv; i; return false;}
      // box selector
};

//////////////////////////// class Body ////////////////////////////

class Body: public ExtSys{ // external bodies
   // adds the shape, and physical properties
   // as variables, from which then the interaction quantities (such as
   // forces) can be calculated.

protected:

// independent quantities

   R1 pinhole_;
      // if a particle has a radius smaller than this it will
```

```
      // not feel a force from the shape and thus may penetrate the body

   BodyProperties bodyProp;
      // The Body has physical properties

   SubsetOfSpace shape;
      // needed for defining repulsive forces which prevent particles
      // from penetrating the body

   Color color;
      // color for visualization of the contour of shape

   void transformNonStreamableData(const Group& g)
   {
      ExtSys::transformNonStreamableData(g);
      shape*=g;
   }

public:
   void setColor() // initializing color
   {
      color=Color(Vec::dimension==2 ?
         CpmImaging::LIGHTGRAY:
         CpmImaging::INVISIBLE);
   }
      // Note that in the additive coloring mode (which is the most
      // information preserving one) WHITE is not usefull since it makes
      // white independent of the color to which it gets added (just as
      // it would be the case in the color setting mode). So, we use a
      // gray instead of white.

// no redefinition of update(), transformStreamableData(const Group&)

//public:
   typedef Body Type;
   CPM_IO
   CPM_NAM(Body)
   virtual RigidObject* clone()const{ return new Body(*this);}

   Body(){}

   explicit Body(const ExtSys& es,
      const BodyProperties& bp=BodyProperties(),
      const SubsetOfSpace& sp=SubsetOfSpace()):
         ExtSys(es), bodyProp(bp),shape(sp)
   {
      shape.set_x(x_);
      shape.set_r(r_);
      setColor();
   }
```

```
   void set_bodyProp(const BodyProperties& bp){ bodyProp=bp;}

   void set_shape(const SubsetOfSpace& sh)
   {
      shape=sh;
      shape.set_x(x_);
      shape.set_r(r_);
   }

   void setPinhole(R r){pinhole_=R1(r);}
   R getPinhole()const{ return pinhole_;}

   R getEeff()const{ return bodyProp.E_;}

   R get_dr()const{ return shape.get_dr();}
   void set_dr(R dr){ shape.set_dr(dr);}

   R get_ds()const{ return shape.get_ds();}
   void set_ds(R ds){ shape.set_ds(ds);}

   void set_E(R E){  bodyProp.E_=E; }
   void setCofOfNorRes(R cnr){ bodyProp.scnr_=cnr*cnr;}
      // setting the 'coefficient of normal restitution

   V< X4<Force,Spc,Force,Spc> > forceSpots(const ComPar& cl)const;
      // virtual in ExtSys

   X2<Force,Spc> forceSpot(const PointPar& p)const;
      // virtual in ExtSys

   R gap(PointPar const& p)const;
      // if this is negative, it is an overlap

   R gap(ComPar const& p)const;
      // if this is negative, it is an overlap

// potentials of elastic interaction with (the shape of) *this
   R potElastic(PointPar const& p)const;
      // potential energy of p due to interaction with the body
   R potElastic(ComPar const& p)const;

   virtual void mark(const Camera& sc)const
   {
      if (active && !SysControl::ignoreExternalSystems)
         sc.mark(shape,LOW,color);
   }
      // notice that the field should only be showed if added together
      // with all other external systems (argument 'color' was missing
      // till 03-01-07
```

```
      // In 3-dimensional space the representation of shapes by
      // a 'curtain of light rays' is not trivial to interprete but
      // useful. See explanations to function
      // void mark(const SubsetOfSpace&, SitRec i, const Color& c )const;
      // in cpmcamerax.h for details.

// see SubsetOfSpace for an explanation of the following three functions
   virtual Plane tangentialPlane(const Spc& x)const
      { return shape.tangentialPlane(x);}

   X2<R,Vec> distAndDirection(const Spc& x)const
      { return shape.distAndDirection(x);}
      // the step for numerical differentiation is shape.get_dr().

   X4<Spc,Spc,Vec,R> clsApp(const PointPar& pa)const;
      // closest approach
      // returns the points of closest approach. Let res be the return
      // value, then res.first is the point p on the surface of *this
      // for which the vector pa.get_x()-p is orthogonal to the surface
      // of *this. If *this and pa don't overlap, this is the pair of
      // points of minimum distance.
};

//////////////////// strct HomFilData ////////////////////////////////
// Data for a homogeneous field

struct HomFilData{ // data for a homogeneous field
   VecE fE_;
      // components of a homogeneous E-field
   VecBH fBH_;
      // components of a homogeneous BH-field
   VecG fG_;
      // components of a homogeneous G-field

   HomFilData(){}

   HomFilData(VecE const& fE, VecBH const& fBH, VecG const& fG):
      fE_(fE),fBH_(fBH),fG_(fG){}

   bool prnOn(ostream& str)const
   {
      cpmwt("HomFilData");
      cpmp(fE_);
      cpmp(fBH_);
      cpmp(fG_);
      return true;
   }

   bool scanFrom(istream& str)
   {
```

```
      cpms(fE_);
      cpms(fBH_);
      cpms(fG_);
      return true;
   }
   typedef HomFilData Type;
   CPM_IO_IMPL
};

//////////////////// class HomFil ////////////////////////////////////
// Homogeneous field

class HomFil: public ExtSys{ // homogeneous field
   // not derived from Body, since no shape
   // is associated with it

   HomFilData hfd_;

protected:
   void update();
   void transformNonStreamableData(const Group& g){g;}
public:
   typedef HomFil Type;
   CPM_NAM(HomFil)
   virtual ExtSys* clone()const{ return new HomFil(*this);}
   CPM_IO
   HomFil(){}
   HomFil(ExtSys const& es, HomFilData const& hfd):ExtSys(es),hfd_(hfd)
   { field_only=1; HomFil::update();}

   void freeStep(R dt)
   {
      RigidObject::freeStep(dt);
      update();
   }

   void setOmega(AxVec const& om){ RigidObject::setOmega(om);update();}

   HomFilData getHomFilData()const{ return hfd_;}
   void mark(const Camera& sc)const{}

};

//////////////////////////// class VibData ////////////////////////////////

class VibData{ // vibration data
// vibration data. Representative for present (2006-02-28)
// coding style
   Rdim amp_;
      // amplitude
```

```
   Rdim omg_;
      // omega, angular velocity of vibration
   Rdim del_;
      // delta, phase of vibration
   typedef VibData Type;
public:
   CPM_IO
   CPM_NAM(VibData)
   VibData(){}
   VibData(Record const& rec, Word const& sec);
   // A typical input for vibration taking place in the 1-2-plane only:
/*
   pot vibration data
      // section title
   Rs vibAmp=0.05 0.05 0
      // three amplitudes
   Rs vibFrq=1 1 0
      // three vibration frequencies (in rotations per second)
   Rs vibPhs=0 0.25 0
      // three relative phases (ranging from 0 to 1 for a full rotation)
*/
   // The vibration is a generalized Lissayou motion t |--> shift, where
   // shift[i]:=vibAmp[i]*sin(2*PI*(vibFrq[i]*t+vibPhs[i]));
   // i=1,2(,3)
   // For the data from the example Record above we have a circular motion
   // since the frequencies of x-motion and y-motion are equal and there
   // is a quarter-wave shift between them.

   Group vib(R t)const;
      // vibration shift at time t, in the data format that fits ito the
      // function X::upd() where X is a class derived from Body.
      // See explanation of the previous function for the type of motion
      // and the meaning of the parameters defining it.
};

//////////////////////// class Pot ////////////////////////////////////

struct PotData{ // data for a container vessel
   // auxiliary structure to collect geometric
   // data of a container vessel

   Rdim s;
      // length of the axis-parallel rectangular parallel-epiped
      // in 1,2,(3)-direction
   B sideWalls_;
      // if this is false, there are no side walls which
      // limit motion in 1-direction. This helps to make
      // use of toroidal geometry

   PotData(){}
```

```
      // notice that all data members belong to a class

   PotData(R s1, R s2, R s3):s(s1,s2,s3),sideWalls_(true){}

   explicit PotData(Rdim const& s_):s(s_),sideWalls_(true){}

  void setSideWalls(bool b){ sideWalls_=B(b);}

  bool getSideWalls()const{ return sideWalls_;}

   bool prnOn(ostream& str)const
   {
      cpmwt("PotData");
      cpmp(s);
      cpmp(sideWalls_);
      return true;
   }

   bool scanFrom(istream& str)
   {
      cpms(s);
      cpms(sideWalls_);
      return true;
   }
   typedef PotData Type;
   CPM_IO_IMPL
};

class Pot: public Body{ // box-shaped vessel
   // a vessel having the shape of a rectangular parallel-epiped
   // (Quader in German)
   // Not closed from above; for closing we introduce class Lid.

// independent quantities

   PotData dat;

   VibData vd;

// dependent quantities

   V<Spc> corners;

protected:

   void upd();
      // updates the values of the dependent quantities according to the
      // values of the independent ones
   void transformNonStreamableData(const Group& g){g;upd();}
public:
```

```
typedef Pot Type;
virtual RigidObject* clone()const{ return new Pot(*this);}
CPM_IO
CPM_NAM(Pot)

Pot(){}
   // does not give a consistent object since Pot::upd() is not called

Pot(const Body& bo, const PotData& dat_, VibData const& vd_=VibData()):
   Body(bo),dat(dat_), vd(vd_){ upd();}

void setPotDat(Rdim s_)
{
   dat.s=s_;
   upd();
}

Rdim getPotDat()const
{
   return dat.s;
}

R get_h()const{ return dat.s[2];}

PotData getPotData()const{ return dat;}

void freeStep(R dt)
{
   RigidObject::freeStep(dt);
   upd();
}

bool boxSel(const ComPar& cl, Iv iv, Z i=2);
   // box selector
   // returns true if cl.get_x()[i] belongs to the interval
   //    Iv(iv[1]*get_h(),iv[2]*get_h())
   // Thus a small zone below the Lid gets described as follows:
   //    cl in zone <==> boxSel(cl,Iv(0.9,1),2)==true
   // Notice that this refers to the master system (and not to the
   // Pot system of reference. From the Pot we only use the
   // value of get_h() as a scale factor.

bool boxSel(const PointPar& cl, Iv iv, Z i=2);
   // box selector
   // returns true if cl.get_x()[i] belongs to the interval
   //    Iv(iv[1]*get_h(),iv[2]*get_h())
   // Thus a small zone below the Lid gets described as follows:
   //    cl in zone <==> boxSel(cl,Iv(0.9,1),2)==true
   // Notice that this refers to the master system (and not to the
   // Pot system of reference. From the Pot we only use the
```

```
      // value of get_h() as a scale factor.

   void mark(const Camera& sc)const;
};

////////////////////// class Lid ////////////////////////////////////

struct LidData{ // data of a lid for the container vessel
   // auxiliary structure to collect geometric
   // data of a lid for the container vessel
   Iv iv1 ;
// extension in 1-direction
   Iv iv3 ;
// extension in 3-direction
   R1 y0;
// 2-coordinate of the lid's plane; there is no extension in
// 2-direction

   B isBottom;
   // if this is true, the repulsive body is below the plane
   // y=y0

   LidData(){}
      // notice that all data members belong to a class

   LidData(Iv ivX, Iv ivZ, R y):
      iv1(ivX),iv3(ivZ),y0(y){}

   bool prnOn(ostream& str)const
   {
      CpmRoot::writeTitle("LidData",str);
      cpmp(iv1);
      cpmp(iv3);
      cpmp(y0);
      cpmp(isBottom);
      return true;
   }

   bool scanFrom(istream& str)
   {
      cpms(iv1);
      cpms(iv3);
      cpms(y0);
      cpms(isBottom);
      return true;
   }

   typedef LidData Type;
   CPM_IO_IMPL
};
```

```
class Lid: public Body{ // a cover of the pot

   LidData dat;

   VibData vd;

// dependent quantities

   V<Spc> corners;

protected:

   void upd();
      // updates the values of the dependent quantities according to the
      // values of the independent ones
   void transformNonStreamableData(const Group& g){g;upd();}
public:
   typedef Lid Type;

   virtual RigidObject* clone()const{ return new Lid(*this);}
   CPM_IO
   CPM_NAM(Lid)

   Lid(){}
      // does not give a consistent object since Lid::upd() is not called

   Lid(const Body& bo, const LidData& dat_, VibData const& vd_=VibData()\
   ):
      Body(bo),dat(dat_), vd(vd_){ upd();}

   LidData getData()const{ return dat;}

   void freeStep(R dt)
   {
      RigidObject::freeStep(dt);
      upd();
   }

   void makeBottom(){dat.isBottom=true;}
      // per default any Lid is of ceiling type, i.e. with the
      // free space below its plane (and the repulsive body above
      // above its plane). This function changes the Lid *this
      // such that free space is above the plane (and ...).

   void mark(const Camera& sc)const;
};

////////////////////// class Fen ////////////////////////////////////
```

```
struct FenData{ // data for a grid-like separator (fence) of the pot
   // auxiliary structure to collect geometric
   // data for a grid-like separator that divides the container
   // vessel in two parts.

   Iv iv1 ;
// extension in 1-direction
   Iv iv3 ;
// extension in 3-direction
   R1 r;
// radius of the cylinders, the pickets of the fence
   R1 g;
// free gap between the pickets

   FenData(){}
      // notice that all data members belong to a class

   FenData(Iv ivX, Iv ivZ, R radius, R gap):
      iv1(ivX),iv3(ivZ),r(radius),g(gap){}

   bool prnOn(ostream& str)const
   {
      CpmRoot::writeTitle("FenData",str);
      cpmp(iv1);
      cpmp(iv3);
      cpmp(r);
      cpmp(g);
      return true;
   }

   bool scanFrom(istream& str)
   {
      cpms(iv1);
      cpms(iv3);
      cpms(r);
      cpms(g);
      return true;
   }

   typedef FenData Type;
   CPM_IO_IMPL
};


class Fen: public Body{ // separator (fence) that divides the pot
   // Fen stands for Fence
   // a way to seperate pot into two parts for the study of osmosis.
   // An array of cylindrical obstacles ('pickets') is a more simple
   // semi-permeable divice than a membrane with holes in it.
```

```
// independent data
   FenData dat;
      // determines the geometry

// dependent data
   V<R> cenX;
      // x-values of the centers of the pickets

protected:

   void upd();
      // updates the values of the dependent quantities according to the
      // values of the independent ones
   void transformNonStreamableData(const Group& g){g;upd();}
public:
   typedef Fen Type;
   virtual RigidObject* clone()const{ return new Fen(*this);}
   CPM_IO
   CPM_NAM(Fen)

   Fen(){}
      // does not give a consistent object since Fen::upd() is not
      // called

   Fen(const Body& bo, const FenData& dat_):Body(bo),dat(dat_)
   { upd();}

   FenData getData()const{ return dat;}

   void freeStep(R dt)
   {
      RigidObject::freeStep(dt);
      upd();
   }

   void mark(const Camera& sc)const;
};

//////////////////////////// class SphTank ////////////////////////////////
// Spherical tank
// The container with the most simple and regular mathematical structure
// that is expected to be useful in testing integrators. While the Pot
// container may give rise to exceptional dynamical states when particles
// hit the container walls near to an edge or even near to a corner,
// hitting the wall of the spherical tank should always be a completely
// regular event.

struct SphTankData{ // data for a spherical tank
   // auxiliary structure to collect geometric
```

```
   // data of a container vessel

   R1 r_;
      // length of the axis-parallel rectangular parallel-epiped
      // in 1,2,(3)-direction

   SphTankData(){}
      // notice that all data members belong to a class

   explicit SphTankData(R r):r_(r){}

   bool prnOn(ostream& str)const
   {
      CpmRoot::writeTitle("SphTankData",str);
      cpmp(r_);
      return true;
   }

   bool scanFrom(istream& str)
   {
      cpms(r_);
      return true;
   }
   typedef SphTankData Type;
   CPM_IO_IMPL
};


class SphTank: public Body{ // spherical tank
// independent quantities

   SphTankData dat_;
// no dependent quantities

protected:

   void upd();
      // updates the values of the dependent quantities according to the
      // values of the independent ones
   void transformNonStreamableData(const Group& g){g;upd();}
public:
   typedef SphTank Type;
   virtual RigidObject* clone()const{ return new SphTank(*this);}
   CPM_IO
   CPM_NAM(SphTank)

   SphTank(){}
      // does not give a consistent object since SphTank::upd() is not
      // called
```

```
   SphTank(const Body& bo, const SphTankData& dat):Body(bo),dat_(dat)
   { upd();}

   void setSphTankDat(R r)
   {
      dat_.r_=r;
      upd();
   }

   R get_h()const{ return 2*dat_.r_;}

   SphTankData getSphTankData()const{ return dat_;}

   void freeStep(R dt)
   {
      RigidObject::freeStep(dt);
      upd();
   }

   void mark(const Camera& sc)const;
};

////////////////////// class Drum ////////////////////////////////////

struct DrumData{ // data of a rotating drum
   // auxiliary structure to collect geometric
   // and kinematic data of a rotating drum

   R1 amp;
      // amplitude: amplitude of drum surface roughness

   R1 pointsPerHalfwave;
      // points per halfwave (for visualization of the contour
      // 8 is OK for most cases. Needs not to be integer

   R1 k1;
      // spatial frequency of drum surface roughness in 1-direction

   R1 k3;
      // spatial frequency of drum surface roughness in 3-direction

   R1 del1;
      // delta 1: phase constant of drum surface roughness in 1-direction

   R1 del3;
      // delta 3: phase constant of drum surface roughness in 3-direction

   R1 r0;
      // radius of the drum
```

```
   R1 omg;
      // angular velocity of rotation arround the 3-axis (also in 2D!)

   B hollow;

   DrumData(){}
      // notice that all data members belong to a class

   DrumData(RecordHandler const& rch, Word const& sec, Z ped=1);

   bool prnOn(ostream& str)const
   {
      cpmwt("DrumData");
      cpmp(amp);
      cpmp(pointsPerHalfwave);
      cpmp(k1);
      cpmp(k3);
      cpmp(del1);
      cpmp(del3);
      cpmp(r0);
      cpmp(omg);
      cpmp(hollow);
      return true;
   }

   bool scanFrom(istream& str)
   {
      cpms(amp);
      cpms(pointsPerHalfwave);
      cpms(k1);
      cpms(k3);
      cpms(del1);
      cpms(del3);
      cpms(r0);
      cpms(omg);
      cpms(hollow);
      return true;
   }

   typedef DrumData Type;
   CPM_IO_IMPL
};

class Drum: public Body{ // a rotating rough cylindrical cavity

// independent quantities

   DrumData drd;

   VibData vd;
```

```
// dependent quantities

   Z nP;
   // number of points for visulaization, dependent - needs not
   // to be written to file

protected:
   void upd();
      // updates the values of the dependent quantities according to the
      // values of the independent ones
   void transformNonStreamableData(Group const& g){g;upd();}

public:
   typedef Drum Type;
   virtual RigidObject* clone()const{ return new Drum(*this);}
   CPM_IO
   CPM_NAM(Drum)

   Drum(){}
      // does not give a consistent object since updateDrum is not
      // called

   Drum(Body const& bo, DrumData const& drd_,
      VibData const& vd_=VibData()):Body(bo),drd(drd_),vd(vd_)
   { AxVec om; om[3]=drd.omg; setOmega(om),upd();}

   void setSurRog(R amp, R k1, R k3, R del1, R del3)
      // set surface roughness
   {
      drd.amp=amp;
      drd.k1=k1;
      drd.k3=k3;
      drd.del1=del1;
      drd.del3=del3;
      upd();
   }

   R_Vector getSurRog()const
   {
      R_Vector res(5);
      res[1]=drd.amp;
      res[2]=drd.k1;
      res[3]=drd.k3;
      res[4]=drd.del1;
      res[5]=drd.del3;
      return res;
   }

   DrumData getDrumData()const{ return drd;}
```

```
   R get_r0()const{ return drd.r0;}

   void freeStep(R dt)
   {
      RigidObject::freeStep(dt);
      upd();
   }

   void mark(const Camera& sc)const;
      // Shows the contour only.
};
```

21 cpmc.h

```
//? cpmc.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_C_H_
#define CPM_C_H_
/*
   Description: Declaration of a class of complex numbers
       using the system provided type <complex> for implementation was
       tried. But this class has only very limited ammount of
       functions and my component access via [] can't be implemented
       here since the components are private.
       Division by zero will write a message, return 0 and continue.
       Test_c<C> d(10,1,1) gave a result which ~10-7 (including read
       write difference which is OK.
*/

#include <cpmword.h>

namespace CpmRoot{

class C;
// These are declarations of functions in scope CpmRoot
// most of them have a friend declaration within class C
// which enables a more efficient implementation.
R arg(C const& z);
R radius(C const& z);
C log10(C const& z);
C pow(C const& z, Z const& n);
C pow(C const& z, C const& n);
C sin(C const& z);
C cos(C const& z);
C tan(C const& z);
C cot(C const& z);
C sinh(C const& z);
C cosh(C const& z);
C tanh(C const& z);
C coth(C const& z);
C arcsin(C const& z);
C arccos(C const& z);
C arctan(C const& z);
C arccot(C const& z);
C arcsinh(C const& z);
C arccosh(C const& z);
C arctanh(C const& z);
C arccoth(C const& z);
C operator /(C const& z, R const& s);
```

```
class C { // complex numbers

   R re, im;
   typedef C Type;

public:

//// general infrastructure, in part with inline implementation
// This is essentially the general interface of templates Vr<> and Fr<>
// appart from the fact that here no function is virtual since we don't
// need to derive from this 'ultimate' class.

// order and comparison
   CPM_ORDER

// I/O operations
   CPM_IO

// test devices
   C ranVal(Z j=0)const;
// value of res.re is in the open interval (-|re|,|re|)
// value of res.im is in the open interval (-|im|,|im|)

   R disVal(C const& y)const;
   C testVal(Z)const;
// descriptors
   Word nameOf(void)const{ return Word("C");}

   Word toWord()const;
      //: to word
      // for a nice printable representation

   Z hashVal(void) const;
      //: hash value

   C net(Z i=0)const{ if (i==1) return C(1,0); else return C(0,0);}

   C inv(void)const;
      //: inversion
      // Returns C(0,0) upon division by zero, but creates
      // warnings on cpmcerr if this happens. The total
      // number of warnings that can arrise in this way is limited
      // so that this file can't grow too much by this mechanism.

   C operator !(void)const{ return inv();}

// complex conjugation
   C con(void)const{ return C(re,-im);}
   void con_(){im=-im;}
```

```
   C operator~(void)const{ return C(re,-im);}
   //: ~
   // instead of z.con() we may write ~z

   C operator|(C const& z)const{ return con()*z;}
   //: (|) scalar product
   // The combination ~z1*z2 can also be written as (z1|z2)

// more specific topics

   static C I;  // 'imaginary unit'
   static C one; // number one = C(1,0)
   static C Ibar; // conjugate of I = - I = C(0,-1)

   C(void):re(0.),im(0.){}

   explicit   C(R const& u, R const& v=0. ):re(u),im(v){}
      // constructor from real and imaginary part
      // no automatic conversion from R to C

   C(R r, R phi, Word w):re(r*::cos(phi)),im(r*::sin(phi)){}
      // constructor from polar coordinates
      // In calling the constructor it is helpful to
      // use w="polar"

   R real(void)const{ return re;}

   R imag(void)const{ return im;}

   Z dim(void)const{ return 2;} // dimension as a real linear space
      //: dimension

   R arg(void)const{ R a=::atan2(im,re); return a<=Pi ? a : a-Pi2; }
      //: argument
      // The value is in the interval (-Pi,Pi] independent
      // of the convention followed by the system's ::atan2 function

   R rho(void)const;
      //: rho
      // common symbol for the polar radius

   R rhoFast()const{ return ::sqrt(re*re+im*im);}
      //: rho fast
      // No provision against overflow by sqaring potentially large
      // numbers.

   R absVal(void)const{ return rho();}
      //: absolute value

   R absSqr()const{ return re*re+im*im;}
```

```
      //: absolute (value) squared

   bool isVal()const
      { return CpmRoot::isVal(re)&&CpmRoot::isVal(im);}

   C scaleTo(R const& r)const;
      // returns a complex number of absolute value |r| and the direction
      // of *this for positive r and antidirection for negative r.

   C timesI(void)const{ return C(-im,re); }

   C dividedByI(void)const{ return C(im,-re);}

   C dividedBy2I(void)const{ return C(0.5*im,-0.5*re);}

   void pol_(R r, R phi);
      //: polar
      // changes *this into r*exp(i*phi)
      // 'modern' indication of mutating nature of the function

   static C pol(R r, R phi){C z; z.pol_(r,phi); return z;}
      //: polar
      // construction from polar coordinates as a static
      // function instead of a constructor

   void pol_1_2(R& r, R& phi)const{ r=rho(); phi=arg();}
      //: polar
      // 'modern' indication of reference nature of first
      // and second argument

// heritage versions of functions related to polar coordinates
   void polar(R const& r, R const& phi){ pol_(r,phi);}
      // changes *this into r*exp(i*phi)
      // not conforming to the C+- naming convention

   void toPolar(R& r, R& phi)const;
      // after call r and phi have the values radius() and arg()

   R radius(void)const{ return rho();}
      // radius

   R rFast(void)const{ return rhoFast();}

// end of heritage versions of functions related to polar coordinates

   R nor_(){R r=absVal();if(r!=0){R ri=1./r;re*=ri;im*=ri;}return r;}
      //: normalize
      // If the return value is different from 0, *this gets changed
      // a unit length number. The return value is the original
      // length.
```

```
   C sqrt()const;
   C sqr()const;
   C exp()const;
   C ln()const;

   C pow(C const& p)const;
   C sin()const;
   C cos()const;
   C tan()const;
   C cot()const;
   C sinh()const;
   C cosh()const;
   C tanh()const;
   C coth()const;
   C arcsin()const;
   C arccos()const;
   C arctan()const;
   C arccot()const;
   C arcsinh()const;
   C arccosh()const;
   C arctanh()const;
   C arccoth()const; // these added 2005-08-23

   friend R arg(C const& z);
   friend R radius(C const& z);
   static C expi(R phi){ return C(::cos(phi),::sin(phi));}
   friend C log10(C const& z);
   friend C pow(C const& z, const Z& n);
   friend C pow(C const& z, C const& n);
   friend C sin(C const& z);
   friend C cos(C const& z);
   friend C tan(C const& z);
   friend C cot(C const& z);
   friend C sinh(C const& z);
   friend C cosh(C const& z);
   friend C tanh(C const& z);
   friend C coth(C const& z);
   friend C arcsin(C const& z);
   friend C arccos(C const& z);
   friend C arctan(C const& z);
   friend C arccot(C const& z);
   friend C arcsinh(C const& z);
   friend C arccosh(C const& z);
   friend C arctanh(C const& z);
   friend C arccoth(C const& z);

// components for uniformity with arrays:  z.re=z[1], z.im=z[2]
// no exceptions !!!
```

```
   R& operator[](int i){ return i==2 ? im : re;}

   R const& operator[](int i)const{ return i==2 ? im : re;}

// mutating arithmetics
// There are a few more operations to be defined (such as C+R)
// which are not prepared in the interfaces; so everything
// is done explicitely
   C& operator +=(C const& x){re+=x.re; im+=x.im; return *this;}
   C& operator +=(R const& s){re+=s; return *this;}

   C& operator -=(C const& x){re-=x.re; im-=x.im; return *this;}
   C& operator -=(R const& s){re-=s; return *this;}

   C& operator *=(C const& x)
     {R re0=re; re=re*x.re-im*x.im; im=im*x.re+re0*x.im; return *this;}

   C& operator /=(C const& x){ return operator *=(x.inv());}

   C& operator *=(R const& s){re*=s; im*=s; return *this;}
   C& operator /=(R const& s);

// generating arithmetics
   C operator -(void)const{ return C(-re,-im);}
   C operator +(C const& z)const
      { return C(re+z.re,im+z.im);}
   friend C operator +(R const& s, C const& z2);
   C operator +(R const& s)const
      { return C(re+s,im);}
   C operator -(C const& z)const
      { return C(re-z.re,im-z.im);}
   friend C operator -(R const& s, C const& z2);
   C operator -(R const& s)const
      { return C(re-s,im);}
   C operator *(C const& z)const
      { return C(re*z.re-im*z.im,im*z.re+re*z.im);}
   friend C operator *(R const& s, C const& z2);
   C operator *(R const& s)const
       { return C(s*re,s*im);}
   C operator /(C const& z)const
      { return (*this)*z.inv();}
   friend C operator /(C const& z, R const& s);
};

// These are definitions of functions in scope CpmRoot, declared friend
// in class C.
inline C operator +(R const& s, C const& z2)
      { return C(z2.re+s,z2.im);}
inline C operator -(R const& s, C const& z2)
      { return C(s-z2.re,-z2.im);}
```

```
inline C operator *(R const& s, C const& z2)
       { return C(s*z2.re,s*z2.im);}
// The following functions need not to be friends of C.
// their argument in ways that need them to be declared friends of C.
inline C ranVal(C const& x, Z j){ return x.ranVal(j);}
inline R disVal(C const& x1, C const& x2){ return x1.disVal(x2);}
inline C testVal(C const& x, Z cpl){ return x.testVal(cpl);}
inline Z hashVal(C const& x){ return x.hashVal();}
inline R absVal(C const& x){ return x.radius();}
inline C net(C const& x, Z i=0){ return x.net(i);}
inline C inv(C const& x){ return x.inv();}
inline C con(C const& x){ return x.con();}
inline C sqrt(C const& z){ return z.sqrt();}
inline R Re(C const& z){return z.real();}
inline R Im(C const& z){return z.imag();}
inline C exp(C const& z){return z.exp();}
inline C expI(R phi){ return C(::cos(phi),::sin(phi));}
      // even for Angle alpha, we can write directly expI(alpha) since
      // CpmGeo::Angle gets automatically converted to an R-valued arc
      // So, instead of saying
      // C z; z.polar(r,phi);
      // we may, much nicer, say:
      // C z=r*expI(phi);
inline C ln(C const& z){ return z.ln();}
inline C log(C const& z){ return z.ln();}
inline C asin(C const& z){ return CpmRoot::arcsin(z);}
inline C acos(C const& z){ return CpmRoot::arccos(z);}
inline C atan(C const& z){ return CpmRoot::arctan(z);}
inline C acot(C const& z){ return CpmRoot::arccot(z);}
inline C asinh(C const& z){ return CpmRoot::arcsinh(z);}
inline C acosh(C const& z){ return CpmRoot::arccosh(z);}
inline C atanh(C const& z){ return CpmRoot::arctanh(z);}
inline C acoth(C const& z){ return CpmRoot::arccoth(z);}

} // namespace
#endif
```

22 cpmc.cpp

```
//? cpmc.cpp
//? Status of work 2008-10-25.
//?
#include <cpmc.h>
#include <cpmtypes.h>

using namespace CpmRoot;
using namespace CpmSystem;

C CpmRoot::C::I=C(0,1.);
C CpmRoot::C::Ibar=C(0,-1.);
C CpmRoot::C::one=C(1.,0);

Word C::toWord()const
      // for a nice printable representation
{
   Word sep_i=" "; // initial
   Word sep_f=" "; // final separator
   Word res;
   if (im==0) res=cpmwrite(re);
   else if (re==0){
      if (im>0) res="i*"&cpmwrite(im);
      else res="-i*"&cpmwrite(-im);
   }
   else if (im>0){
      res=cpmwrite(re)&"+i*"&cpmwrite(im);
   }
   else{
      res=cpmwrite(re)&"-i*"&cpmwrite(-im);
   }
   return sep_i&res&sep_f;
}

bool C::prnOn(ostream& str )const
{
   //if (!CpmRoot::writeTitle("C",str)) return false;
   cpmwat;
   cpmp(re);
   cpmp(im);
   return true;
}

bool C::scanFrom(istream& str )
{
   cpms(re);
   cpms(im);
   return true;
```

```
}

// inversions

C C::inv(void)const
// same logic as inv(R) for error handling
// see Press et al. p. 177 (5.4.5) for the algorithm
{
   static const Z maxMes=100;
   static Z mes=1;
   R c_=(re>=0. ? re : -re);
   R d_=(im>=0. ? im : -im);
   if (c_>=d_){
      if (c_==0) goto DIVBYZERO;
      R y=im/re; R z=1./(re+im*y); return C(z,-y*z);
   }
   else{
      if (d_==0) goto DIVBYZERO;
      R y=re/im; R z=1./(re*y+im); return C(y*z,-z);
   }

DIVBYZERO:

      if (mes==maxMes){
         mes++; // needed !
         Message::message(
            "C::inv(): argument is (0,0) ... messages discontinued");
      }
      if (mes<maxMes){
         mes++;
         Message::warning(
            "C::inv(): argument is (0,0);  (0,0) returned");
      }
      return C(0,0);
}

C& C::operator /=(R const& s)
{
   R si=CpmRoot::inv(s);
   re*=si;
   im*=si;
   return *this;
}

C CpmRoot::operator /(C const& z, R const& s)
{
   R si=CpmRoot::inv(s);
   return z*si;
}
```

```
// Functions related to polar representation

R CpmRoot::arg(C const& x)
{
   return x.arg();
}

R C::rho(void)const
// modified from Press et al. p. 177 (5.4.4)
{
   R a_=(re>=0. ? re : -re);
   R b_=(im>=0. ? im : -im);
   if (a_>=b_){
      if (a_==0) return 0.;
      R y=b_/a_; return a_*::sqrt(1+y*y);
   }
   else{
      if (b_==0) return 0.;
      R y=a_/b_; return b_*::sqrt(1+y*y);
   }
}

R CpmRoot::radius(C const& x)
{
   return x.radius();
}

void C::pol_(R r, R phi)

{
   re=r*::cos(phi);
   im=r*::sin(phi);
}


void C::toPolar(R& r, R& phi)const
{
   r=radius();
   phi=arg();
}

// Square

C C::sqr(void)const
{
   C res(*this);
   return res*res;
}

// Square root
```

```
C C::sqrt(void)const
{
   R r=radius(), phi=arg();
   r=::sqrt(r);
   phi*=0.5;
   return C(r*::cos(phi),r*::sin(phi));
}

// Elementary transcendental functions. We need the following real
// functions to be provided by the compiler:

//      exp, log, sin, cos, sqrt, atan. If complex numbers Ca are to be
//      defined based on a class Ra of numbers of arbitrary precission,
//      we have to implement these functions also for Ra.

C C::exp()const
{
   R r=::exp(re);
   return C(r*::cos(im),r*::sin(im));
}

C C::ln()const
{
   R r=radius(), phi=arg();
   r=::log(r);  // library function
   return C(r,phi);
}

C CpmRoot::log10(C const& z)
// logarithm to basis 10
{
   static const R log10_=1./::log(10.);
   return log10_*ln(z);
}

C CpmRoot::pow(C const& z, const Z& n)
{
   C res(1.,0);
   if (n==0) return res;

   C z_; Z n_;
   if (n>0){
      z_=z;
      n_=n;
   }
   else{
      z_=!z;
      n_=-n;
    }
```

```
   for (Z k=1; k<=n_; k++) res*=z_;
   return res;
}

C CpmRoot::pow(C const& z, C const& w)
{
   return exp(w*ln(z));
}

C CpmRoot::sin(C const& z)
{
   return (exp(z.timesI())-exp(z.dividedByI())).dividedBy2I();
}

C CpmRoot::cos(C const& z)
{
   return (exp(z.timesI())+exp(z.dividedByI()))*0.5;
}

C CpmRoot::tan(C const& z){ return sin(z)/cos(z);}

C CpmRoot::cot(C const& z){ return cos(z)/sin(z);}

C CpmRoot::sinh(C const& z){ return 0.5*(exp(z)-exp(-z));}

C CpmRoot::cosh(C const& z){ return 0.5*(exp(z)+exp(-z));}

C CpmRoot::tanh(C const& z){ return sinh(z)/cosh(z);}

C CpmRoot::coth(C const& z){ return cosh(z)/sinh(z);}

C CpmRoot::arcsinh(C const& z)
   // Gradshteyn Ryzhik 1.622 5.
{ return ln(z+(z*z+1).sqrt());}

C CpmRoot::arccosh(C const& z)
   // Gradshteyn Ryzhik 1.622 6.
{ return ln(z+sqrt(z*z-1));}

C CpmRoot::arctanh(C const& z)
   // Gradshteyn Ryzhik 1.622 7.
{ return ln((z+C(1))/(C(1)-z))*0.5;}

C CpmRoot::arccoth(C const& z)
   // Gradshteyn Ryzhik 1.622 8.
{ return ln((z+C(1))/(z-C(1)))*0.5;}

C CpmRoot::arcsin(C const& z)
   // Gradshteyn Ryzhik 1.622 1.
{ return (arcsinh(z.timesI())).dividedByI();}
```

```
C CpmRoot::arccos(C const& z)
   // Gradshteyn Ryzhik 1.622 2.
{ return (arccosh(z)).dividedByI();}

C CpmRoot::arctan(C const& z)
   // Gradshteyn Ryzhik 1.622 3.
{ return (arctanh(z.timesI())).dividedByI();}

C CpmRoot::arccot(C const& z)
   // Gradshteyn Ryzhik 1.622 4.
{ return (arccoth(z.timesI())).timesI();}

C C::pow(C const& p)const{ return CpmRoot::pow(*this,p);}
C C::sin()const{ return CpmRoot::sin(*this);}
C C::cos()const{ return CpmRoot::cos(*this);}
C C::tan()const{ return CpmRoot::tan(*this);}
C C::cot()const{ return CpmRoot::cot(*this);}
C C::sinh()const{ return CpmRoot::sinh(*this);}
C C::cosh()const{ return CpmRoot::cosh(*this);}
C C::tanh()const{ return CpmRoot::tanh(*this);}
C C::coth()const{ return CpmRoot::coth(*this);}
C C::arcsin()const{ return CpmRoot::arcsin(*this);}
C C::arccos()const{ return CpmRoot::arccos(*this);}
C C::arctan()const{ return CpmRoot::arctan(*this);}
C C::arccot()const{ return CpmRoot::arccot(*this);}
C C::arcsinh()const{ return CpmRoot::arcsinh(*this);}
C C::arccosh()const{ return CpmRoot::arccosh(*this);}
C C::arctanh()const{ return CpmRoot::arctanh(*this);}
C C::arccoth()const{ return CpmRoot::arccoth(*this);}
   // these added 2005-08-23

// Indicators

/*************** utility functions ******************************/

C C::testVal(Z cpl)const
{
   const R red=0.01;
      // complex functions like exp() should be included in tests and
      // don't behave uncritical for large arguments
   R a,b;
   a=CpmRoot::testVal<R>(a,cpl);
   b=CpmRoot::testVal<R>(b,cpl);
   a*=red;
   b*=red;
   return C(a,b);
}

C C::ranVal(Z j)const
```

```
{
   R a,b;
   if (j==0){
      a=CpmRoot::ranVal(re,0);
      b=CpmRoot::ranVal(im,0);
   }
   else{
      Z j2=j*2;
      a=CpmRoot::ranVal(re,j2);
      b=CpmRoot::ranVal(im,j2+1);
   }
   return C(a,b);
}

R C::disVal(C const& z)const
{
   return CpmRoot::distFunc(radius(),z.radius(),(*this-z).radius());
}

Z C::com(C const& s)const
{
   if (re<s.re) return 1;
   if (re>s.re) return -1;
   if (im<s.im) return 1;
   if (im>s.im) return -1;
   return 0;
}

Z C::hashVal()const
{
   Z ix=CpmRoot::hashVal<R>(re);
   Z iy=CpmRoot::hashVal<R>(im);

   return ix^iy;
}

C C::scaleTo(R const& r)const
      // returns a complex number of absolute value |r| and the direction
      // of *this for positive r and antidirection for negative r.
{
   R rOrig=absVal();
   if (rOrig==0.) return C(r,0);
   C res=*this;
   return res*(r/rOrig);
}
```

23 cpmcamera.h

```
//? cpmcamera.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_CAMERA_H_
#define CPM_CAMERA_H_
/*

Description: Visualisation tool for 2D and 3D

*/

#include <cpmdim.h>
#include <cpmgraph.h>
#include <cpmimagingtools.h>
#include <cpmvectorgraphics.h>
#include <cpmrecordhandler.h>

namespace CpmColCam{  // collection of names relevant for cameras
   using CpmImaging::STEREORED;
   using CpmImaging::STEREOGREEN;
   using CpmImaging::STEREOBLACK;
   using CpmImaging::STEREO;
   using CpmImaging::INVISIBLE;
   using CpmImaging::BLACK;
   using CpmImaging::WHITE;
   using CpmImaging::Color;
   using CpmGraphics::Frame;
   using CpmGraphics::Graph;
   using CpmGraphics::SitRec;
   using CpmGraphics::LOW;
   using CpmGraphics::UPP;
   using CpmGraphics::LEFT;
   using CpmGraphics::RIG;
   using CpmRootX::RecordHandler;
}

using CpmArrays::X2;

#include <cpmcamera2.h>
#include <cpmcamera3.h>

#endif
```

24 cpmcamera.cpp

```
//? cpmcamera.cpp
//? Status of work 2008-10-25.
//?
#include <cpmcamera.h>
#include <cpmcamera2cpp.h>
#include <cpmcamera3cpp.h>
```

25 cpmcamera2.h

```
//? cpmcamera2.h
//? Status of work 2008-10-25.
//?
/*
     Description: class Camera achieves visualization of 3-dimensional
      objects via anaglyph-technique. Here we have the less
      sophisticated 2D version of it.
*/

namespace CpmCamera2{

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::C;
   using CpmRoot::Word;
   using CpmRoot::Pi;

   using CpmGeo::Angle;
   using CpmGeo::Iv; // Interval

   using CpmRootX::B;
   using CpmRootX::R1;
   using CpmRootX::Z1;

   using CpmLinAlg::R2;
   using CpmLinAlg::R3;
   using CpmLinAlg::RR2;
   using CpmLinAlg::RR3;

   using CpmFunctions::F;
   using CpmFunctions::F1;

   using CpmArrays::V;
   using CpmGraphics::VecGraph;
   using CpmGraphics::VecGraphs;

   using CpmImaging::Response;

using namespace CpmColCam;
using namespace CpmDim2;

//////////////////////////// class Camera ////////////////////////////////

   class Camera{ // sees a rectangular part of infinite 2D object space
      // Tool for visualizing the motion of clusters
      // in space using the 'red green glasses method' (anaglyph method)
```

```
// independent data

      Color color1;
         // color for image corresponding to image 1 (left eye)

      Color color3;
         // color for painting the background. It may be better to have
         // this not completely black in order to reduce visibility of
         // reflexes.

      Spc poi;
         // 'point of interest'= center of field of view in object space
         // all construction methods of Camera make this point the origin
         // of Spc().
         // geometrical transformations enable this to change after
         // construction.

      mutable Graph gr;
         // viewing window in the film plane. Defines the position and
         // size of the 'film gate'.

      mutable Response rsp;
         // a quantity which controls the conversion from R to
         // pixel brightness control values

 // dependent data

      Vec v;
         // unit vector in the vertical direction in the image plane

      Vec h;
         // unit vector in the horizontal direction in the image plane

     void complete(void);
         // assumes that the indpendent data have their valid values
         // and initializes the dependent data accordingly

     B mono;
         // Will be used only in CpmCamera3::Camera. So there is no
         // problem with the initialization as false

   public:
      typedef Camera Type;
      CPM_IO

      Word nameOf()const{ return Word("CpmCamera2::Camera");}

      Camera(R imageHeight=1, R aspectRatio=1.4,
         const Angle& viewAngle=Angle(50), R frac=1.,
         const Angle& conv=Angle(5), const Frame& fr=Frame());
```

```
   // gives s and frac as they are, and the angle of convergence.
   // poi is the origin. The viewpoints are located in the
   // (1,3)-plane and the mean vision direction is the -3
   // direction.
   // This is natural in a coordinate system in which the screen is
   // the (1,2) plane with 1,2 corresponding to conventional x,y
   // ( x-axis to the right, y-direction upwards). In order to
   // follow conventional orientation, the 3 direction points
   // towards the observer of the screen plane, that direction of
   // vision is -3 direction. The angle conv is the deviation of
   // the individual vision directions against the mean vision
   // direction. Thus the angle between the two vision directions
   // is 2*conv. The colors for marking on screen are presently set
   // such that the left eye is assumed to look through a red
   // filter and the right eye through a green filter. Camera
   // position in space different from the one adopted here, is
   // achieved by operator *=(const Group&).

   // To precisely characterize the geometry of the objects, we
   // consider an idealized derived Camera, which resuls from the
   // actual one by shrinking the convergence angle to zero while
   // the point of convergence remains fixed. This associated
   // mono camera has a unique center of field.
   // The overall location
   // of the generated Camera is such that this 'image center'
   // coincides with the origin of Spc. The orientation is just as
   // for the Camera generated by the previous constructor. The
   // returned Graph object has the object space data corresponding
   // to the film piece of the associated mono camera. It is a
   // rectangle of height 'imageHeight' and a width 'imageHeight'*
   // 'aspectRatio'. The viewAngle gives the angular size of the
   // film diagonal as seen from the eye position (distance frac*s
   // from the film center).

C captureMono(const Spc& p)const
   {Vec d=p-poi; return C((d|h),(d|v));}
   // basic imaging function, inline for efficiency

pair< pair<C,R>,pair<C,R> > capture(const Spc& p,
   const R& r)const;
   // the values for radius and position of a sphere at point p
   // with radius r as seen on the two image planes of the stereo
   // camera are returned. Notice that simply marking corresponding
   // circles in red and green on screen is not sufficient to get
   // an anaglyph picture due to the 'solid' colors. In the overlap
   // area of red and green one needs yellow and not the solid
   // color of the circle to be drawn last. The first return data
   // component corresponds to the left eye and the second
   // component to the right eye.
```

```
      pair<C,R> captureMono(const Spc& p, const R& r)const;
         // same as the previous function, but activating only the 1-part
         // of the stereo camera

      pair< C, C > capture(const Spc& p)const;
         // Same a previous function, but ignoring the radius and thus
         // returning two graphical positions only

      Spc inverse(const C& x, const R& mag )const
      { mag; return poi+h*x[1]+v*x[2];}
         // returns the position p in space of a point which on the first
         // stereo-image gets the 'film-plane-coordinates' x and for
         // which an small extended lateral object near p of linear size
         // s gets imaged as one linear size mag*s on the film.
         // abstract definition:
         // captureMono(inverse(x,mag),r)=pair<R2,R>(x,r*mag)  for all R2
         // x and R r, mag

      AxVec getAxis()const{ return AxVec(1.);}
         // returns a unit-vector in viewing direction as a data
         // type which makes sense in 2D and 3D

      void fillCircle(const C& center, R radius,
         const Color& color )const
         {gr.fillCircle(R2(center[1],center[2]), radius, color);}
         // shows a circular disk

      Camera& operator *=(const Group& g);
         // moving the whole camera

      void shift(const R& sx, const R& sy, const R& sz=0.);
         // translation of the whole camera. sx is the shift in direction
         // of h (horizontal image direction)
         // and sy is the shift in direction v (vertical image direction)
         // sz is the shift in direction of e , the direction of view

// in the following 3 operations the 'point of interest' is held fixed
// The teminology for the next three functions follows the book
// Isaac Victor Kerlow: The Art of 3-D Computer Animation and Imaging,
// Van Nostrand Reinhold, 1996
      void tilt(const Angle& a);
         // positive a lets the camera look 'more to the sky'
         // (positive rotation arround h)
      void pan(const Angle& a);
         // positive a lets the camera look 'more to the left'
         // (positive rotation arround v)
      void roll(const Angle& a);
         // positive a lets the 'horizon' (horizontal line before roll)
         // appear ascending from the left to the right
         // (positive rotation arround e)
```

```
      void adjustFieldOfView( R enlarge, R sx=0, R sy=0, R sz=0);
      // enlarge is a factor by which the viewing window gets enlarged
      // compared to the one which is presently valid. (sx,sy) describes
      // a shift of the viewing window such that (1,1) shifts the lower
      // left corner of the enlarged window to the position previously
      // held by the upper-right corner.
      // sz describes a shift in direction of the mean viewing direction
      // in units of the image diagonal. The latter parameter, as
      // enlarge, is capable of changing the visual size of objects,
      // using sz changes the perspective too.

      void shiftGate(R sx=0, R sy=0){gr.shift(sx,sy);}
         // shifts the 'filmgate' in a manner that is clear from th code

      Color getColor1()const{ return color1;}
      Color getColor2()const{ return Color(BLACK);}
         // useful to have uniform interface for 2D and 3D
      Color getColor3()const{ return color3;}

      bool is3D()const{ return false;}
      bool is2D()const{ return true;}
      void setBW_(){ color1=BLACK; color3=WHITE;}

      void mark(const RefSys& rs, Z i)const;
         // marks the positive coordinate i-axis by a chain of points

      #include <cpmcamerax.h>
         // declarations which do not depend on dimension
    };

} // namespace
```

26 cpmcamera2cpp.h

```
//? cpmcamera2cpp.h
//? Status of work 2008-10-25.
//?
////////////////// class Camera ////////////////////////////////////

namespace CpmCamera2{

using namespace CpmStd;
using namespace CpmColCam;
using CpmDim2::AxVec;
using CpmDim2::Vec;
using CpmDim2::Spc;
using CpmDim2::RefSys;
using CpmDim2::Group;
using CpmDim2::SubsetOfSpace;
using CpmDim2::Skeleton;
using CpmDim2::WireFrame;
using CpmDim2::Texture;
//using CpmImaging::YELLOW;

void Camera::complete(void)
// s,frac,e1,e2,poi given
{}

V<Vec> Camera::getVectors()const
{
   V<Vec> res(2);
   res[1]=h;
   res[2]=v;
   return res;
}

Camera::Camera(R imageHeight, R aspectRatio, const Angle& viewAngle,
   R frac_, const Angle& conv, const Frame& fr):
   color1(WHITE), color3(STEREOBLACK),
   h(1,0), v(0,1),poi(),gr(fr)
{
   viewAngle; // not used
   frac_;
   conv;
   R width=imageHeight*aspectRatio;
   gr.setXY(-0.5*width,-0.5*imageHeight,width,imageHeight);
   complete();
}

bool Camera::prnOn(ostream& out)const
{
```

```
   if (!CpmRoot::writeTitle("Camera",out)) return false;
   if (!color1.prnOn(out)) return false;
   if (!color3.prnOn(out)) return false;
   if (!poi.prnOn(out)) return false;
   if (!v.prnOn(out)) return false;
   if (!h.prnOn(out)) return false;
   if (!gr.prnOn(out)) return false;
   return true;
}

bool Camera::scanFrom(istream& in)
{
   if (!color1.scanFrom(in)) return false;
   if (!color3.scanFrom(in)) return false;
   if (!poi.scanFrom(in)) return false;
   if (!v.scanFrom(in)) return false;
   if (!h.scanFrom(in)) return false;
   if (!gr.scanFrom(in)) return false;
   complete();
   return true;
}

Camera& Camera::operator *=(const Group& g)
{
   poi*=g; h*=g; v*=g; complete();
      // using complete here assures constancy of geometry even in the
      // presence
      // of accumulating roundoff errors
   return *this;
}

void Camera::shift(const R& sx, const R& sy, const R& sz)
{
   sz;
   Vec tr=h*sx+v*sy;
   Group g(tr);
   operator *=(g);
//   gr.shiftAbs(sx,sy);
   // looks natural but seems not to do what was expected in my
   // present context (02-09-09) so I don't include it in order
   // not to create side effects
}

void Camera::tilt(const Angle& a)
{
   a;
}

void Camera::pan(const Angle& a)
{
```

```
   a;
}

void Camera::roll(const Angle& a)
{
   AxVec av(a);
   Group g(poi,av);
   operator *=(g);
}

void Camera::adjustFieldOfView( R enlarge, R sx, R sy, R sz)
{
   sz; // not used
   gr.scale(enlarge);
   R sxa=gr.getdX();
   R sya=gr.getdY();
   sxa*=sx;
   sya*=sy;
   shift(sxa,sya,0);
}

void Camera::scale(R enlarge)
{
   gr.scale(enlarge);
}

pair<C,C> Camera::capture(const Spc& p)const
{
   C res=captureMono(p);
   return pair<C,C>(res,res);
}

pair<pair<C,R>, pair<C,R> > Camera::capture(const Spc& p,
   const R& r)const
{
   C res=captureMono(p);
   typedef pair<C,R> prr2;
   prr2 px(res,r);
   return pair<prr2,prr2>(px,px);
}

pair<C,R> Camera::captureMono(const Spc& p, const R& r)const
{
   C pr=captureMono(p);
   return pair<C,R>(pr,r);
}

void Camera::mark(const Spc& x, R r)const
{
   C pp=captureMono(x);
```

```
   fillCircle(pp,r,getColor1());
}

void Camera::mark(const Spc& x, R r, const Color& c)const
{
   pair<C,R> prr=captureMono(x,r);
   fillCircle(prr.first,prr.second,c);
}

bool Camera::mark(const Spc& x, const Color& c)const
{
   const Z size=4;
   C px=captureMono(x);
   if (!c.visible()){
      gr.mark(px,size,color1);
   }
   else{
      gr.mark(px,size,c);
   }
   return gr.indFunc(R2(px[1],px[2]));
}

void Camera::mark(const Spc& x1, const Spc& x2, R rFoot)const
{
   C a=captureMono(x1);
   C b=captureMono(x2);
   gr.draw(R2(a[1],a[2]),R2(b[1],b[2]),color1);
   if (rFoot>0){
      fillCircle(a,rFoot,color1);
   }
}

void Camera::mark(const Spc& x1, const Spc& x2, const Color& c,
   R rFoot)const
   // shows a line from x1 to x2 in color c (as seen by camera
   // *this in 1-direction)
{
   if ( !c.visible() ){
      mark(x1,x2,rFoot);
   }
   else{
      C a=captureMono(x1);
      C b=captureMono(x2);
      gr.draw(R2(a[1],a[2]),R2(b[1],b[2]),c);
      if (rFoot>0) fillCircle(a,rFoot,c);
   }
}

void Camera::mark(const RefSys& rs, Z i)const
{
```

```
   const R step=2e-6;
   Spc p=rs.o();
   Vec sh;
   if (i==1) sh=rs.e1();
   else if (i==2) sh=rs.e2();
   else if (i==-1) sh=-rs.e1();
   else if (i==-2) sh=-rs.e2();
   else cpmerror("Camera::mark(RefSys,Z i), i out of range");
   sh*=step;
   Color c(Z(-1));
   while (isVisible(p)){p+=sh; mark(p,c);}
}

void Camera::dirTo_(Spc const& x, R const& r, R fitFrame,
   Angle tilt, Angle pan, RefSys const& rs)
{
   tilt;
   pan;
   RefSys rsOrig=getRefSys();
   Group g=rs/rsOrig;
   operator *=(g); // now the camera has standard orientation with
      //respect to system rs
   Vec sh=x-poi;
   operator *=(Group(sh));
   R rAct= r<=0 ? 1 : r;
   R imageHeight=2*rAct*fitFrame;
   R aspectRatio=getdX()/getdY();
   R width=imageHeight*aspectRatio;
   gr.setXY(-0.5*width,-0.5*imageHeight,width,imageHeight);
   complete();
}

void Camera::show(const SubsetOfSpace& sos, const Texture& tx,
   bool bodyFix)const
{
    using CpmArrays::VV;
    using CpmLinAlg::Z2;
    using namespace CpmGraphics;

    Z nx=dim1();
    Z ny=dim2();
    R dx=getdX()/nx;
    R dy=getdY()/ny;

    R xStart=getxLeft()+0.5*dx;
    R yStart=getyUpper()-0.5*dy;
         // (x,y) pixel center of the first (top-left) film pixel
         // in object space
    R x=xStart;
    R y;
```

```
    Z i0=gr.a0();
    Z j0=gr.b0();
    R nxInv=1./nx;
    for (Z i=0;i<nx; i++){
       cpmprogress("capture",i*nxInv,3);
       y=yStart;
       RefSys sosrs=sos.getRefSys();
       for (Z j=0; j<ny; j++){
          Spc sp=poi+h*x+v*y;
          bool b=sos.ind(sp);
          R br;
          if (bodyFix){
            br=(b ?  tx.eval(sosrs,sp) : 0);
          }
          else{
             br=(b ?  tx(sp) : 0);
          }
          Color c;
          if (br==0 ) c=color3; else c=br*color1;
          gr.setPel(i+i0,j+j0,c);
          y-=dy;
       }
       x+=dx;
    }
    gr.display();
}

#include <cpmcameraxcpp.h>
   // common part of the implementation

}//namespace
```

27 cpmcamera3.h

```
//? cpmcamera3.h
//? Status of work 2008-10-25.
//?
/*
     Description: class Camera achieves visualization of 3-dimensional
      objects via anaglyph-technique
*/

namespace CpmCamera3{

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::C;
   using CpmRoot::Word;
   using CpmRoot::Pi;

   using CpmGeo::Angle;
   using CpmGeo::Iv; // Interval

   using CpmRootX::B;
   using CpmRootX::R1;
   using CpmRootX::Z1;

   using CpmLinAlg::R2;
   using CpmLinAlg::R3;
   using CpmLinAlg::RR2;
   using CpmLinAlg::RR3;

   using CpmFunctions::F;
   using CpmFunctions::F1;

   using CpmArrays::V;
   using CpmGraphics::VecGraph;
   using CpmGraphics::VecGraphs;

   using CpmImaging::Response;

using namespace CpmColCam;
using namespace CpmDim3;

//////////////////////////// class Camera ////////////////////////////////

   class Camera{ // maps the 3D object space onto a 2D image rectangle
      // Tool for visualizing the motion of clusters
      // in space using the 'red green glasses method' (anaglyph method)

// independent data
```

```
Color color1;
   // color for image corresponding to image 1 (left eye)

Color color2;
   // color for image corresponding to image 2 (right eye)

Color color3;
   // color for painting the background. It may be better to have
   // this not completely black in order to reduce visibility of
   // reflexes.

R s;
   // distance from each of the viewpoints ('eyes') to the point of
   // convergence pc (i.e the point where the optical axes of the
   // eyes intersect, we only consider stereo cameras for which
   // such an intersection point exist)

R frac;
   // s*frac is the distance from the viewpoint to an imaginary
   // transparent coordinate-projection screen on which the
   // intersection of a ray from object point to viewpoint
   // generates the (x,y) record of the point.
   // (1.-frac)*s is the distance from the point of convergence
   // to the point of interest (poi).
   // For frac<1, the topological order is: eye,poi,pc.
   // For frac>1 we have eye, pc, poi
   // frac==1 is a natural situation.

Vec e1;
   // unit vector from viewpoint 1 to the point of convergence

Vec e2;
   // unit vector from viewpoint 2 to the point of convergence

Vec v;
   // unit vector in direction of e1.cross(e2).
   // Vertical direction in the image planes. If e1 and e2 are
   // close together or identical, then they don't determine v,
   // So we have to treat this quantity as independent

Spc poi;
   // 'point of interest'= center of field of view in object space
   // all construction methods of Camera make this point the origin
   // of Spc(). Geometrical transformations enable this to change
   // after construction.

mutable Graph gr;
   // viewing window in the film plane. Defines the position and
   // size of the 'film gate'.
```

```
        // Note that the film gate has a well defined origin.

     mutable Response rsp;
        // a quantity which controls the conversion from R to
        // pixel brightness control values

// dependent data

     Spc vp1; // view point 1, left eye

     Spc vp2; // view point 2, right eye

     Spc vp;  // mean of vp1 and vp2

     Spc pc; // point of convergence

     Vec h1;
        // e1.cross(v), horizontal direction in image plane 1

     Vec h2;
        // e2.cross(v), horizontal direction in image plane 2

     Vec h;
        // unit vector in direction of (h1+h2)/2

     Vec e;
        // unit vector in direction of (e1+e2)/2

     void complete(void);
        // assumes that the indpendent data have their valid values
        // and initializes the dependent data accordingly

     B mono;
        // initialized as false. If true, we use the camera as a
        // mono-camera. This is useful for capturing 3D line art
        // for creating scientific illustrations

  public:
     typedef Camera Type;
     CPM_IO
     Word nameOf()const{ return Word("CpmCamera3::Camera");}
     Camera(R imageHeight=1, R aspectRatio=1.4,
        Angle const& viewAngle=Angle(10), // was 50
        R frac=1., Angle const& conv=Angle(2.5), // was 5
        Frame const& fr=Frame());
  // Takes frac from the argument and sets s from imageHeight,
  // aspectRatio, and viewAngle.   poi is the origin.
  // The viewpoints are located in the (1,3)-plane
  // and the mean vision direction is the -3 direction.
  // This is natural in a coordinate system in which the screen is the
```

```
// (1,2) plane with 1,2 corresponding to conventional x,y ( x-axis to
// the right, y-direction upwards). In order to follow conventional
// orientation, the 3 direction points towards the observer of the
// screen plane, that direction of vision is -3 direction.
// The angle conv is the deviation of the individual vision directions
// against the mean vision direction. Thus the angle between the two
// vision directions is 2*conv. The colors for marking on screen are
// presently set such that the left eye is assumed to look through a
// red filter and the right eye through a green filter. Camera
// position in space different from the one adopted here, is achieved
// by operator *=(Group const&).

// To precisely characterize the geometry of the objects, we consider
// an idealized derived Camera, which results from the actual one by
// shrinking the convergence angle to zero while the point of
// convergence remains fixed. This associated mono camera has a unique
// center of field. The overall location of the generated Camera is
// such that this 'image center' coincides with the origin of Spc.
// The Graph data  member gr has the object space data corresponding
// to
// the film piece of the associated mono camera. It is a rectangle of
// height 'imageHeight' and a width 'imageHeight'*'aspectRatio'.
// The viewAngle gives the angular size of the film diagonal as
// seen from the eye position (distance frac*s from the film center)

   Camera(const Spc& cenFilm, const Spc& cenScn, const Vec& ver,
      R aspectRatio=1.4,
      const Angle& viewAngle=Angle(50),
      const Angle& conv=Angle(5),
      const Frame& fr=Frame());
      // center of film, center of scene (=point of concergence),
      // vertical vector, aspect ratio, viewing angle (corresponds
      // to film diagonal, angle of convergence, display frame on
      // screen

   pair< pair<C,R>,pair<C,R> > capture(const Spc& p,
      const R& r)const;
      // the values for radius and position of a sphere at point p
      // with radius r as seen on the two image planes of the stereo
      // camera are returned. Notice that simply marking corresponding
      // circles in red and green on screen is not sufficient to get
      // an anaglyph picture due to the 'solid' colors. In the overlap
      // area of red and green one needs yellow and not the solid
      // color of the circle to be drawn last. The first return data
      // component corresponds to the left eye and the second
      // component to the right eye.

   pair< C, C > capture(const Spc& p)const;
      // Same a previous function, but ignoring the radius and thus
      // returning two graphical positions only
```

```
      pair<C,R> captureMono(const Spc& p, const R& r)const;
         // same as the previous function, but activating only 'one
         // half' of the stereo camera and using a vision direction
         // which is the mean of the two stereo vision directions
         // (namely e).

      C captureMono(const Spc& p)const;
         // Same a previous function, but ignoring the radius and thus
         // returning one graphical position only

      Spc inverse(const C& x, const R& mag )const;
         // returns the position p in space of a point which on the first
         // stereo-image gets the 'film-plane-coordinates' x and for
         // which an small extended lateral object near p of linear size
         // s gets imaged as one linear size mag*s on the film.
         // Abstract definition:
         // captureMono(inverse(x,mag),r)=pair<R2,R>(x,r*mag)
         // for all R2 x and R r, mag

      AxVec getAxis()const{ return AxVec(e);}
         // returns a unit-vector in viewing direction as a data
         // type which makes sense in 2D and 3D

      bool is3D()const{ return true;}
      bool is2D()const{ return false;}
      void setBW_(){ color1=color2=BLACK; color3=WHITE;}

      void fillCircle(const R2& center, R radius,
         const Color& color )const
         {gr.fillCircle(center, radius, color);}
         // shows a circular disk

      Camera& operator *=(const Group& g);
         // moving the whole camera

      void shift(const R& sx, const R& sy, const R& sz=0.);
         // translation of the whole camera. sx is the shift in direction
         // of h (horizontal image direction)
         // and sy is the shift in direction v (vertical image direction)
         // sz is the shift in direction of -e , the z-direction in a
         // local system of reference in which h=x-direction,
         // v=y-direction.

// in the following 3 operations the 'point of interest' is held fixed
// The teminology for the next three functions follows the book Isaac
// Victor Kerlow: The Art of 3-D Computer Animation and Imaging,
// Van Nostrand Reinhold, 1996
      void tilt(const Angle& a);
         // positive a lets the camera look 'more to the sky'
```

```
         // (positive rotation arround h)
      void pan(const Angle& a);
         // positive a lets the camera look 'more to the left'
         // (positive rotation arround v)
      void roll(const Angle& a);
         // positive a lets the 'horizon' (horizontal line before roll)
         // appear ascending from the left to the right
         // (positive rotation arround e)

      void adjustFieldOfView( R enlarge, R sx=0, R sy=0, R sz=0);
      // enlarge is a factor by which the viewing window gets enlarged
      // compared to the one which is presently valid. (sx,sy) describes
      // a shift of the viewing window such that (1,1) shifts the lower
      // left corner of the enlarged window to the position previously
      // held by the upper-right corner. sz describes a shift in
      // direction of the mean viewing direction in units of the image
      // diagonal. The latter parameter, as enlarge,
      // is capable of changing the visual size of objects, using sz
      // changes the perspective too.

      void shiftGate(R sx=0, R sy=0){gr.shift(sx,sy);}
         // shifts the 'filmgate' in a manner that is clear from th code

      Color getColor1()const{ return color1;}
      Color getColor2()const{ return color2;}
      Color getColor3()const{ return color3;}

      void mark(const RefSys& rs, Z i, R len=1.)const;
         // marks the positive coordinate i-axis by a chain of points
         // If i>3 all axes are shown, then only a coordinate range
         // not yet ready

      #include <cpmcamerax.h>
         // declarations which do not depend on dimension
   };

} // namespace
```

28 cpmcamera3cpp.h

```
//? cpmcamera3cpp.h
//? Status of work 2008-10-25.
//?
/*

Description: specific implementation for 3D

*/

namespace CpmCamera3{

using namespace CpmStd;
using namespace CpmColCam;

using CpmDim3::AxVec;
using CpmDim3::Vec;
using CpmDim3::Spc;
using CpmDim3::RefSys;
using CpmDim3::Group;
using CpmDim3::SubsetOfSpace;
using CpmDim3::Skeleton;
using CpmDim3::WireFrame;
using CpmDim3::Texture;
using CpmGraphics::BLUE;

////////////////// class Camera ////////////////////////////////////

void Camera::complete(void)
// s,frac,e1,e2,poi,v given
{
   e=(e1+e2).polDec().c2();
   R s1f=s*(1.-frac);
   pc=poi+e*s1f;
   vp1=pc-e1*s;
   vp2=pc-e2*s;
   vp=pc-e*s;
   h1=e1.cross(v);
   h2=e2.cross(v);
   h=(h1+h2).polDec().c2();
   gr.stereo();
}

V<Vec> Camera::getVectors()const
{
   V<Vec> res(3);
   res[1]=h;
   res[2]=v;
```

```
   res[3]=-e;
   return res;
}

Camera::Camera(R imageHeight, R aspectRatio, const Angle& viewAngle,
   R frac_, const Angle& conv, const Frame& fr):
   color1(STEREORED), color2(STEREOGREEN),color3(STEREOBLACK),
   frac(frac_),e1(0,0,-1),e2(0,0,-1),v(0,1,0),poi(),gr(fr)
{
   R width=imageHeight*aspectRatio;
   gr.setXY(-0.5*width,-0.5*imageHeight,width,imageHeight);
   C z(imageHeight,width);
   R diag=z.absVal();
   R alpha=viewAngle; // alpha now in rad
   R t=tan(alpha*0.5); // frac*s*tan(alpha/2) = diag/2  defines s
   t*=(2*frac);
   if (t<=0) cpmerror("Camera(...): bad viewing angle");
   s=diag/t;
   AxVec av(conv,v);
   Group g2(av);
   Group g1=!g2;
   e1*=g1; // 1 corresponds to the left eye
   e2*=g2;  // 2 corresponds to the right eye
   complete();
}

Camera::Camera(const Spc& cenFilm, const Spc& cenScn, const Vec& ver,
   R aspectRatio, const Angle& viewAngle, const Angle& conv,
   const Frame& fr):
   color1(STEREORED), color2(STEREOGREEN),color3(STEREOBLACK),
   frac(1),poi(cenFilm),gr(fr)
{
   Z mL=2;
   cpmmessage(mL,"Camera(Spc,Spc,...) started");
   Word loc("Camera(Spc,Spc,Vec,R,Angle,R,Angle,Frame)");
   Vec cc=cenScn-poi;
   s=cc.absVal();
   cpmassert(s>0,loc);
   R sInv=1./s;
   R diag=2*s*tan(R(0.5)*viewAngle);
   R imageHeight=diag/sqrt(1+aspectRatio*aspectRatio);
   R width=imageHeight*aspectRatio;
   gr.setXY(-0.5*width,-0.5*imageHeight,width,imageHeight);
   e=cc*sInv;
   e1=e;
   e2=e;
   Vec ver1=ver-e*(e|ver);
   R ver1av=ver1.absVal();
   cpmassert(ver1av>0.,loc);
   R ver1avInv=1./ver1av;
```

```
   v=ver1*ver1avInv;
   AxVec av(conv,v);
   Group g2(av);
   Group g1=!g2;
   e1*=g1; // 1 corresponds to the left eye
   e2*=g2;  // 2 corresponds to the right eye
   complete();
   cpmmessage(mL,"Camera(Spc,Spc,...) done");
}

bool Camera::prnOn(ostream& out)const
{
   using CpmRoot::write;
   if (!CpmRoot::writeTitle("Camera",out)) return false;
   if (!color1.prnOn(out)) return false;
   if (!color2.prnOn(out)) return false;
   if (!color3.prnOn(out)) return false;
   if (!write(s,out)) return false;
   if (!write(frac,out)) return false;
   if (!poi.prnOn(out)) return false;
   if (!v.prnOn(out)) return false;
   if (!e1.prnOn(out)) return false;
   if (!e2.prnOn(out)) return false;
   if (!gr.prnOn(out)) return false;
   return true;
}

bool Camera::scanFrom(istream& in)
{
   using CpmRoot::read;
   if (!color1.scanFrom(in)) return false;
   if (!color2.scanFrom(in)) return false;
   if (!color3.scanFrom(in)) return false;
   if (!read(s,in)) return false;
   if (!read(frac,in)) return false;
   if (!poi.scanFrom(in)) return false;
   if (!v.scanFrom(in)) return false;
   if (!e1.scanFrom(in)) return false;
   if (!e2.scanFrom(in)) return false;
   if (!gr.scanFrom(in)) return false;
   complete();
   return true;
}

Camera& Camera::operator *=(const Group& g)
{
   poi*=g; e1*=g; e2*=g; v*=g; complete();
      // using complete here assures constancy of geometry even in the
      // presence of accumulating roundoff errors
   return *this;
```

```
}

void Camera::shift(const R& sx, const R& sy, const R& sz)
{
   Vec tr=h*sx+v*sy-e*sz;
      // notice that h,v,-e is a camera-fixed system of reference
   Group g(tr);
   operator *=(g);
}

void Camera::tilt(const Angle& a)
{
   AxVec av(a,h);
   Group g(poi,av);
   operator *=(g);
}

void Camera::pan(const Angle& a)
{
   AxVec av(a,v);
   Group g(poi,av);
   operator *=(g);
}

void Camera::roll(const Angle& a)
{
   AxVec av(a,e);
   Group g(poi,av);
   operator *=(g);
}

void Camera::adjustFieldOfView( R enlarge, R sx, R sy, R sz)
{
   gr.scale(enlarge);
   R sxa=gr.getdX();
   R sya=gr.getdY();
   R sza=( sz==0.  ?  0.  :  sz*sqrt(sxa*sxa+sya*sya) );
   sxa*=sx;
   sya*=sy;
   shift(sxa,sya,sza);
}

void Camera::scale(R enlarge)
{
   gr.scale(enlarge);
   s*=enlarge;
   complete();
}

pair<C,C> Camera::capture(const Spc& p)const
```

```
{
   R f=s*frac;

   Vec d=p-vp;
   R sp=(d|e);
   cpmassert(sp!=0,"Camera::capture(Spc)");
   R demag=f/sp;
      // if one has a different demag for each eye, one gets a curved
      // visual representation of planes which is not yet understood

   Vec s1=(p-vp1)*demag;
   R x1=(s1|h1);
   R y1=(s1|v);
   C res1(x1,y1);

   Vec s2=(p-vp2)*demag;
   R x2=(s2|h2);
   R y2=(s2|v);
   C res2(x2,y2);

   return pair<C,C>(res1,res2);
}

pair< pair<C,R>, pair<C,R> > Camera::capture(const Spc& p,
         const R& r)const
{
   R f=s*frac;

   Vec d=p-vp;
   R sp=(d|e);
   cpmassert(sp!=0,"Camera::capture(Spc,R)");
   R demag=f/sp;
      // if one has a different demag for each eye, one gets a curved
      // visual representation of planes which is not yet understood

   Vec s1=(p-vp1)*demag;
   R x1=(s1|h1);
   R y1=(s1|v);
   C res1(x1,y1);

   Vec s2=(p-vp2)*demag;
   R x2=(s2|h2);
   R y2=(s2|v);
   C res2(x2,y2);

   R rd=r*demag;

   typedef pair<C,R> prr2;
   prr2 p1(res1,rd);
   prr2 p2(res2,rd);
```

```
   return pair<prr2,prr2>(p1,p2);
}

C Camera::captureMono(const Spc& p)const
{
   Vec d=p-vp;
   R f=s*frac;
   R sp=(d|e);
   cpmassert(sp!=0,"Camera::captureMono(Spc)");
   R demag=f/sp;
   d*=demag;
   R x=(d|h);
   R y=(d|v);
   return C(x,y);
}

pair<C,R> Camera::captureMono(const Spc& p, const R& r)const
{
   Vec d=p-vp;
   R f=s*frac;
   R sp=(d|e);
   cpmassert(sp!=0,"Camera::captureMono(Spc,R)");
   R demag=f/sp;
   d*=demag;
   R x=(d|h);
   R y=(d|v);
   C pr(x,y);
   return pair<C,R>(pr,r*demag);
}

void Camera::mark(const Spc& x, R r, const Color& c)const
{
   if (c.transparent()) return;
   else if (c.stereo()){
      pair< pair<R2,R>,pair<R2,R> > ppp=capture(x,r);
      R r1=ppp.first.second;
      R2 x1=ppp.first.first;
      R r2=ppp.second.second;
      R2 x2=ppp.second.first;
      fillCircle(x1,r1,getColor1());
      fillCircle(x2,r2,getColor2());
   }
   else{
      pair<R2,R> prr=captureMono(x,r);
      fillCircle(prr.first,prr.second,c);
   }
}

void Camera::mark(const Spc& x, R r)const
{
```

```
   mark(x,r,Color(INVISIBLE));
}

bool Camera::mark(const Spc& x, const Color& c)const
   // shows the point x in color c (as seen by camera *this in
   // 1-direction)
{
   const Z size=4;
   C px;
   bool stereo=!c.visible()&&is3D()&&!mono;
   if (stereo){
      pair< C, C > ppp=capture(x);
      C p1=ppp.first;
      C p2=ppp.second;
      px=(p1+p2)*0.5;
      gr.mark(p1,size,color1);
      gr.mark(p2,size,color2);
   }
   else{
      px=captureMono(x);
      if (!c.transparent()) gr.mark(px,size,c);
   }
   return gr.indFunc(R2(px));
}

void Camera::mark(const Spc& a, const Spc& b, R rFoot)const
{
   pair<C,C> pa=capture(a);
   C a1=pa.first;
   C a2=pa.second;
   pair<C,C> pb=capture(b);
   C b1=pb.first;
   C b2=pb.second;
   gr.draw(R2(a1),R2(b1),color1);
   gr.draw(R2(a2),R2(b2),color2);
   if (rFoot>0){
      fillCircle(R2(a1),rFoot,color1);
      fillCircle(R2(a2),rFoot,color2);
   }
}

void Camera::mark(const Spc& x1, const Spc& x2,
                  const Color& c, R rFoot)const
   // shows a line from x1 to x2 in color c (as seen by camera *this
   // in 1-direction)
{
   bool stereo=!c.visible()&&is3D()&&!mono;
   if ( stereo ){
      mark(x1,x2,rFoot);
   }
```

```
   else{ // mono
      C y1=captureMono(x1);
      C y2=captureMono(x2);
      gr.draw(R2(y1),R2(y2),c);
         // was draw_simple
      if (rFoot>0) fillCircle(R2(y1),rFoot,c);
   }
}

Spc Camera::inverse(const C& x, const R& mag)const
{
   R f=s*frac;
   Vec vFilm=e*f+h*x[1]+v*x[2];
   if (mag<=0)
      cpmerror("Spc Camera::inverse(...)const: unexpected mag<=0");
   R magInv=1./mag;
   vFilm*=magInv;
   return vp+vFilm;
}

void Camera::mark(const RefSys& rs, Z i, R len)const
{
   len; // not used
   if (i<=3){
      const R step=2e-6;
      Spc p=rs.o();
      Vec sh;
      if (i==1) sh=rs.e1();
      else if (i==2) sh=rs.e2();
      else if (i==3) sh=rs.e3();
      else if (i==-1) sh=-rs.e1();
      else if (i==-2) sh=-rs.e2();
      else if (i==-3) sh=-rs.e3();
      else cpmerror("Camera::mark(RefSys,Z i), i out of range");
      sh*=step;
      Color c(Z(-1));
      while (isVisible(p)){p+=sh; mark(p,c);}
   }
   else{

   }
}

void Camera::dirTo_(Spc const& x, R const& r, R fitFactor,
   Angle tilt, Angle pan, RefSys const& rs)
{
   Z mL=3;
   Word loc("Camera::dirTo_(Spc,R,...)");
   CPM_MA
   RefSys rsOrig=getRefSys();
```

```
   Group g=rs/rsOrig;
   operator *=(g); // now the camera has standard orientation with
      //respect to system rs
   Vec sh=x-poi;
   operator *=(Group(sh));
   operator *=(Group(x,AxVec(tilt,rs.e1())));
   operator *=(Group(x,AxVec(pan,rs.e2())));
// characterizing the present camera:
   R hh=0.5*getdY();
   if (hh<=0) cpmerror(loc&": unexpected hh<=0");
   R wh=0.5*getdX();
   R aspectRatio=wh/hh;
   R dis=s*frac;
   if (dis<=0) cpmerror("Camera::dirTo_(...): unexpected s*frac<=0");
   R alpha_hh=atan(hh/dis);
       // angle under which the half hight of the object is seen
   R nom=sin(alpha_hh)*frac;
   if (nom<=0) cpmerror("Camera::dirTo_(...): unexpected nominator<=0");
// adjusting the camera such that it is capable of framing the
// bounding sphere of sos. Viewing angle and aspect ratio of frame
// remain unchanged.
   R rAct= r<=0 ? 1 : r; // r==0 occurs e.g. in visualizing a single
      // point and this should not disappear, added 2007-10-13
   R s= rAct/nom;
   R hhNew=s*frac*tan(alpha_hh)*fitFactor;
   R imageHeight=2*hhNew;
   R width=imageHeight*aspectRatio;
   gr.setXY(-0.5*width,-0.5*imageHeight,width,imageHeight);
   complete();
   CPM_MZ
}

void Camera::show(const SubsetOfSpace& sos, const Texture& tx,
   bool bodyFix)const
   // areal visualisation of extended bodies under control of
   // method parameters dr, ds.
{
    using CpmArrays::VV;
    using CpmLinAlg::Z2;
    using namespace CpmGraphics;

    const R fineFac=0.25;
    const R coarseFac=5;
      // factors for setting the method parameters of SubsetOfSpace::hit
      // in terms of the basic visual resolution as given by the
      // pixel size.

    const R tWait=2;
      // time for display of a important message
```

```
if (sos(vp1)<=0){
   cpmwarning("Camera::show(...): left eye inside the body");
   cpmwait(tWait);
}

if (sos(vp2)<=0){
   cpmwarning("Camera::show(...): right eye inside the body");
   cpmwait(tWait);
}

Z i0=gr.a0();
Z j0=gr.b0();

Z nx=dim1();
Z ny=dim2();
R dx=getdX()/nx;
R dy=getdY()/ny;
R resolution=CpmRootX::inf(dx,dy);

R sFine=fineFac*resolution;
R sCoarse=coarseFac*resolution;

R xStart=getxLeft()+0.5*dx;
R yStart=getyUpper()-0.5*dy;
     // (x,y) pixel center of the first (top-left) film pixel
     // in object space
R x=xStart;
R y;
R h0=0.1*s*frac;
R nxInv=1./nx;
RefSys sosrs=sos.getRefSys();
for (Z i=0;i<nx; i++){
   cpmprogress("capture",i*nxInv,3);
   y=yStart;
   for (Z j=0; j<ny; j++){
      Spc sp=poi+h*x+v*y;
      Vec dir1=sp-vp1;
      Vec dir2=sp-vp2;
      dir1.normalize();
      dir2.normalize();

      X2<Spc,bool> prz1=sos.hit(vp1,dir1,sFine,sCoarse);
      X2<Spc,bool> prz2=sos.hit(vp2,dir2,sFine,sCoarse);
      R br1,br2;
      if (!bodyFix){
        br1= prz1.c2() ?  tx(prz1.c1()) : 0;
        br2= prz2.c2() ?  tx(prz2.c1()) : 0;
      }
      else{
        br1= prz1.c2() ?  tx.eval(sosrs,prz1.c1()) : 0;
```

```
            br2= prz2.c2() ?  tx.eval(sosrs,prz2.c1()) : 0;
          }

          Color c;
          if (br1==0 && br2==0) c=color3; else c=br1*color1+br2*color2;
          gr.setPel(i+i0,j+j0,c);
          y-=dy;
       }
       x+=dx;
    }
    gr.display();
}

void Camera::mark(SubsetOfSpace const& sos, Texture const& tx,
   bool bodyFix)const
   // areal visualisation of extended bodies under control of
   // method parameters dr, ds.
{
    using CpmArrays::VV;
    using CpmLinAlg::Z2;
    using namespace CpmGraphics;

    const R fineFac=0.25;
    const R coarseFac=5;
      // factors for setting the method parameters of SubsetOfSpace::hit
      // in terms of the basic visual resolution as given by the
      // pixel size.

    const R tWait=2;
      // time for display of a important message

    if (sos(vp1)<=0){
       cpmwarning("Camera::show(...): left eye inside the body");
       cpmwait(tWait);
    }

    if (sos(vp2)<=0){
       cpmwarning("Camera::show(...): right eye inside the body");
       cpmwait(tWait);
    }

    Z i0=gr.a0();
    Z j0=gr.b0();

    Z nx=dim1();
    Z ny=dim2();
    R dx=getdX()/nx;
    R dy=getdY()/ny;
    R resolution=CpmRootX::inf(dx,dy);
```

```
    R sFine=fineFac*resolution;
    R sCoarse=coarseFac*resolution;

    R xStart=getxLeft()+0.5*dx;
    R yStart=getyUpper()-0.5*dy;
         // (x,y) pixel center of the first (top-left) film pixel
         // in object space
    R x=xStart;
    R y;
    R h0=0.1*s*frac;
    R nxInv=1./nx;
    RefSys sosrs=sos.getRefSys();
    for (Z i=0;i<nx; i++){
       cpmprogress("capture",i*nxInv,3);
       y=yStart;
       for (Z j=0; j<ny; j++){
          Spc sp=poi+h*x+v*y;
          Vec dir1=sp-vp1;
          Vec dir2=sp-vp2;
          dir1.normalize();
          dir2.normalize();

          X2<Spc,bool> prz1=sos.hit(vp1,dir1,sFine,sCoarse);
          X2<Spc,bool> prz2=sos.hit(vp2,dir2,sFine,sCoarse);
          R br1,br2;
          if (!bodyFix){
            br1= prz1.c2() ?  tx(prz1.c1()) : 0;
            br2= prz2.c2() ?  tx(prz2.c1()) : 0;
          }
          else{
            br1= prz1.c2() ?  tx.eval(sosrs,prz1.c1()) : 0;
            br2= prz2.c2() ?  tx.eval(sosrs,prz2.c1()) : 0;
          }

          Color c;
          if (br1==0 && br2==0) c=color3; else c=br1*color1+br2*color2;
          gr.setPel(i+i0,j+j0,c);
          y-=dy;
       }
       x+=dx;
    }
}


#include <cpmcameraxcpp.h>
   // common part of the implementation

} // namespace
```

29 cpmcamerax.h

```
//? cpmcamerax.h
//? Status of work 2008-10-25.
//?
/*
Description: declarations common for 2D and 3D
   Recent history:
      02-09-30: Camera::grid(...) added

*/

//////////////// class Camera ////////////////////////////////////////////

   Camera(const RecordHandler& rch,
      const Word& sec, CpmGraphics::Frame& fr);
      // constructor from RecordHandler

   void dis(bool lineArtProcessing=false)const
      //: display
   {
      bool file= gr.getWriteOnShow();
      gr.dis(file,lineArtProcessing);
   }

   void display(bool file=false)const
      // old name, old functionality, no line art processing
      // In order to not break old code
      { gr.dis(file,false); }


   void clear()const{gr.paint(color3);}
         // paints the whole viewing rectangle of gr with a suitable
         // background color

   void clr()const{clear();}
      //: clear

   void setWriteOnShow(bool wos)const{ gr.setWriteOnShow(wos);}
      // setting gr.writeOnShow. If gr.writeOnShow==true
      // a call to gr.show(...) implies a call to gr.img.write()

   void setColBac(Color c){color3=c;}

   void mono_()
      //: mono
      // mutating function
   {
      if (is3D()){
```

```
      mono=B(true);
      setBW_();
   }
}

bool isMono()const{ return mono;}

// getting the object space data of gr

R getxLeft()const{ return gr.getxLeft();}
R getxRight()const{ return gr.getxRight();}
R getyLower()const{ return gr.getyLower();}
R getyUpper()const{ return gr.getyUpper();}
R getdX()const{ return gr.getdX();}
R getdY()const{ return gr.getdY();}
Z dim1()const{ return gr.dim1();}
Z dim2()const{ return gr.dim2();}

void write(const Word& filename)const{ gr.write(filename);}

void write(void)const{ gr.write();}

void rot(const AxVec& axv);
   // rotation of the whole camera such that the 'point of interest'
   // is held fixed

V<Vec> getVectors()const;
   // returns the vectors h,v,-e (last only in 3D) for getting info on
   // the orientation of the camera

Vec get_h()const{ return h;}
Vec get_v()const{ return v;}

Vec getDir()const{ return -getVectors()[3];}
   // main indicator for the direction of the camera
   // returns a unit vector

Spc getPoint()const{ return poi;}
   // returns the 'point of interest'

Spc get_x()const{ return poi;}
   // for uniformity with class Shape

RefSys getRefSys()const;
   // returns the reference system, which naturally belongs to the
   // camera:
   // axes are h,v,-e (we need a positivly oriented system, so the
   // third axis is against the direction of vision.
   // So, the 1 and 2 axes are the normal x and y axes of the
   // film plane considered as a drawing board
```

```
void scale(R enlarge);
   // multiplies 'film size' and s by factor enlarge and adjusts all
   // other quantities accordingly. The viewing angle thus remains
   // unchanged.

bool isVisible(const Spc& x)const;
   // returns true if the point x can be represented within the
   // graphics window of gr

void setScreenArea( const Graph& g)const;
   // This function redefines gr such that it gets its display area on
   // screen from g, without changing the the objectspace-meaning
   // and the attribute newColors of gr. Thus, if g and gr do not
   // match in aspect ratio, we get distortion

Graph& getGraph()const{ return gr;}
   // retrieving the Graph object gr; return value was Graph
   // (instead of Graph&) till 2003-07-31 which had bad influence
   // in function ScalarField::mark()

void captureGraph()const{/* if (!gr.getNewColors()) gr.capture();*/}

Response& getResponse()const{return rsp;}
   // 2003-07-31 same return strategy as in the previous function
   // implemented

Iv getRangeX()const{ return rsp.getRangeX();}

void mark(const Spc& x, R r)const;
      // shows the sphere with center x and radius r in anaglyphic
      // manner (as seen by camera *this)

void mark(const Spc& x, R r, const Color& c)const;
      // shows the sphere with center x and radius r in Color c
      // manner (as seen by camera *this)

void mark(const Spc& x1, const Spc& x2, R rFoot=0.)const;
      // shows a line from x1 to x2 in anaglyphic manner (as seen by
      // camera *this). If rFoot>0, the position of x1 will be marked
      // by the image of a sphere of a radius rFoot. This allows to
      // represent vectors as oriented line pieces.

bool mark(const Spc& x, const Color& c)const;
      // 1. c.stereo()==true
      // shows the point x in  in anaglyphic representation else.
      // Returns false if the mean of the points as seen from the
      // two stereo directions point falls outside the graphics
      // window defined by gr.
      // 2. c.visible()==true.
```

```
      // shows the point in color c and returns false if the point
      // as seen from the 'mono-direction' e falls outside the
      // graphics window defined by gr.

bool mark(const Spc& x)const{ return mark(x,Color(INVISIBLE));}
      // shows the point x in anaglyphic manner (as seen by camera
      // *this)

void mark(const Spc& x1, const Spc& x2, const Color& color,
      R rFoot=0.)const;
      // shows a line from x1 to x2 in color. If color is invisible,
      // anaglyphic representation is done.
      // If rFoot>0, the position of x1 will be marked by the
      // image of a sphere of a radius rFoot where rFoot refers to
      // the metric in the camera's Graph-object

void mark(V<Spc> const& vs, Color const& color)const
      // showing a polygon (list of points)
{
   for (Z i=1;i<vs.dim();i++) mark(vs[i],vs[i+1],color);
}

void mark(Path const& p, Color const& c)const{ mark(p(),c);}

void mark( V<X2<Spc,Vec> > const& svl, R fac, Color const& color)const
{
   Z i,svld=svl.dim();
   for (i=1;i<=svld;++i){
      X2<Spc,Vec> xi=svl[i];
      Spc pi=xi.c1();
      Vec vi=xi.c2();
      Spc pf=pi+fac*vi;
      mark(pi,pf,color,0);
   }
}

void mark(const SubsetOfSpace&, SitRec i, const Color& c )const;
   // visualisation of extended bodies.

// The function calculates a list of points on the surface of the
// body and marks the camera-image of these points as a polygon.
// This list of surface points lies in the 'film plane' of the camera.
// It is created by a 'virtual curtain of laser test rays' emanating
// perpendicularly from site i of the filmplane and being directed
// inwards.
// These parallel test rays have a distance ~ds. Where they hit
// the surface of the SubsetOfSpace, there are the points mentioned
// above.
// The sites denoted i correspond to the rims of the monitor
// viewing window. LOW: lower rim, RIG: right, UPP: upper, LEFT left.
```

```
// dr gives the distance error allowed in localizing the surface of
// the body. If the section of the contour with the film rectangle
// is a line that may be considered a function graph over one of the
// sites, then marking with this side as second argument gives a
// perfect representation of this contour. For more complicated
// contours marking from more than one side (e.g. from all sites k)
// may be necessary for a representation without gaps. See the
// redefinition of this function in class Shell

 void markCen(const SubsetOfSpace&, Z nP, const Color& c )const;
   // mark central
   // Visualisation of extended bodies. Here the 'virtual curtain
   // of laser test rays' emanates from the center of the camera's
   // film gate. It fills the full circle with nP points.
   // Could become generalized to a fan of rays emanating from
   // an adjustable point.

 void markFan(const SubsetOfSpace& sos, Spc const& p, Vec const& e1,
    AxVec const& n, R ang, Z nP, Color const& c)const;
   // mark (by) fan
   // visualisation of extended bodies. Here the 'virtual curtain
   // of laser test rays' emanates from the point p filling the
   // directions starting with e and consisting of nP directions
   // which originate from e by rotation arround axis ax (only
   // the direction of ax counts). Bewteen two adjacent directions
   // the angle is ang/nP. For nP and ang sufficienly large,
   // the 'fan' may cover an 'angle' much larger than one complete
   // turn. All exceptional arguments are treated reasonably and
   // create warnings (no errors).

void show(const SubsetOfSpace&, const Texture& tx=Texture(),
   bool bodyFix=false)const;
   // areal visualisation of extended bodies. If the last argument
   // is true, the texture tx is not considered fixed in space but
   // moving with the SubsetOfSpace.

void mark(SubsetOfSpace const&, Texture const& tx=Texture(),
   bool bodyFix=false)const;
   // areal visualisation of extended bodies. If the last argument
   // is true, the texture tx is not considered fixed in space but
   // moving with the SubsetOfSpace.

void dirTo_(Spc const& x, R const& r, R fitFactor=1.2,
   Angle tilt=Angle(0), Angle pan=Angle(0),
   RefSys const& rs=RefSys());
   // The camera gets changed such that the objects's bounding sphere
   // just fits the gap between lower and upper boundary of an
   // 'virtual' the display frame, the sites of which are those of
   // the actual display frame divided by fitFactor. Thus, if
   // fitFactor=2 the bounding sphere of sos will fill only one half of
```

```
      // the display window's height.The viewing angle and the parameter
      // frac of the camera remain unchanged. The angular orientation in
      // space of the camera after function call is given by angles tilt
      // and pan with respect to reference system rs. Calling this
      // function does not cause the object *this to show the object sos,
      // it only adjusts geometrical data of the camera.

   void dirTo_(SubsetOfSpace const& sos, R fitFactor, Angle tilt,
      Angle pan, RefSys const& rs=RefSys())
      { dirTo_(sos.get_x(),sos.get_r(),fitFactor,tilt,pan,rs);}
      // The camera gets changed such that the objects's bounding sphere
      // just fits the gap between lower and upper boundary of an
      // 'virtual' the display frame, the sites of which are those of
      // the actual display frame divided by fitFactor. Thus, if
      // fitFactor=2 the bounding sphere of sos will fill only one half of
      // the display window's height.The viewing angle and the parameter
      // frac of the camera remain unchanged. The angular orientation in
      // space of the camera after function call is given by angles tilt
      // and pan with respect to reference system rs. Calling this
      // function does not cause the object *this to show the object sos,
      // it only adjusts geometrical data of the camera.

   void lookTo(const SubsetOfSpace& sos, Angle tilt,
      Angle pan, const Texture& tx=Texture(),
      const RefSys& rs=RefSys(),bool bodyFix=false, R fitFactor=1);
      // This function adjusts the camera as described in the previous
      // function and also shows object sos textured by tx on the
      // display.
      // This was the first form of such a function and the previous
      // function was created by separating the code in geometrical
      // adjustment and imaging. For compatibility with old programs
      // the argument structure was not changed with exception of the
      // insertion of the last argument, which can't break old code.

   void mark(const Skeleton& sk, const Color& c)const;
      // visualisation of extended bodies represented as Skeletons.

   void mark(const WireFrame& wf, const Color& c)const;
      // visualisation of extended bodies represented as
      // arrays of Skeletons.

//   void mark(LineArt const& la)const;
      // visualisation of a LineArt object

   void mark(Triangle const& tri, Color const& c)const;

//   void mark(Wrt const& w, Color const& c)const;

   void mark(const VV<Spc>& vvs, const Color& c)const
      { mark(Skeleton(vvs,true),c);}
```

```
void mark(const RefSys& rs, const Color& c, R lengthOfAxes)const;

VecGraph captMono(const Skeleton& sk)const;
   // generating an instance of class VecGraph from a Skeleton.
   // This is the main tool to graphical representations
   // of WireFrames

V<C> captMono(const V<Spc>& vs)const;
   // mono-image of a list of points

pair< V<C>, V<C> > capt(const V<Spc>& vs);
   // stereo-image of a list of points

pair<VecGraph,VecGraph> capt(const Skeleton& sk)const;
   // stereo version of the previous function

VecGraphs captMono(const WireFrame& wf)const;
   // generating an instance of class VecGraphs from a
   // WireFrame.

pair<VecGraphs,VecGraphs> capt(const WireFrame& wf)const;
   // stereo version of the previous function

void addText(const Word& text, R x=0.05, R y=0.15,
   Color c=CpmImaging::WRITECOLOR, R h=0)const
{ gr.addText(text,x,y,c,h);}

V<Spc> grid(Z ny=8)const;
   // greates a family of space points that fill the
   // image window as a 2D cubic lattice of a width determined
   // by the fact that there are by lines per image height
   // These points are give as a one-dimensional array  since
   // the present mode of using the data cooperates with this format.
```

30 cpmcameraxcpp.h

```
//? cpmcameraxcpp.h
//? Status of work 2008-10-25.
//?
/*
Description: see cpmcamerax.h
*/

V<C> Camera::captMono(const V<Spc>& vs)const
{
   Z vsd=vs.dim();
   V<C> res(vsd);
   for (Z i=1;i<=vsd;i++){
      res[i]=captureMono(vs[i]);
   }
   return res;
}

pair< V<C>, V<C> > Camera::capt(const V<Spc>& vs)
{
   Z vsd=vs.dim();
   V<C> res1(vsd);
   V<C> res2(vsd);
   for (Z i=1;i<=vsd;i++){
      pair<C,C> pcc=capture(vs[i]);
      res1[i]=pcc.first;
      res2[i]=pcc.second;
   }
   return pair< V<C>, V<C> >(res1,res2);
}

VecGraph Camera::captMono(const Skeleton& sk)const
{
   Z skd=sk.dim();

   V<C> vr(skd);
   for (Z i=1;i<=skd;i++){
      vr[i]=captureMono(sk[i]);
   }
   VecGraph res(vr);
   res.set_neg(sk.get_neg());
   return res;
}

pair<VecGraph,VecGraph> Camera::capt(const Skeleton& sk)const
{
   const Z mL=5;
   cpmmessage(mL,"Camera::capt(Skeleton) started");
```

```
   Z skd=sk.dim();

   V<C> v1(skd);
   V<C> v2(skd);
   for (Z i=1;i<=skd;i++){
      pair<C,C> prr=capture(sk[i]);
      v1[i]=prr.first;
      v2[i]=prr.second;
   }
   VecGraph res1(v1);
   VecGraph res2(v2);
   res1.set_neg(sk.get_neg());
   res2.set_neg(sk.get_neg());
   cpmmessage(mL,"Camera::capt(Skeleton) done");
   return pair<VecGraph,VecGraph>(res1,res2);
}

VecGraphs Camera::captMono(const WireFrame& wf)const
{
   Z dwf=wf.dim();
   V<VecGraph> vv(dwf);
   for (Z i=1;i<=dwf;i++){
      vv[i]=captMono(wf[i]);
   }
   return VecGraphs(vv);
}

pair<VecGraphs,VecGraphs> Camera::capt(const WireFrame& wf)const
{
   Z dwf=wf.dim();
   V<VecGraph> vv1(dwf);
   V<VecGraph> vv2(dwf);
   for (Z i=1;i<=dwf;i++){
      pair<VecGraph,VecGraph> pvv=capt(wf[i]);
      vv1[i]=pvv.first;
      vv2[i]=pvv.second;
   }
   return pair<VecGraphs,VecGraphs>(VecGraphs(vv1),VecGraphs(vv2));
}

void Camera::mark(const Skeleton& sk, const Color& c)const
{
   const Z mL=3;
   bool stereo=!c.visible()&&is3D()&&!mono;
   cpmmessage(mL,"Camera::mark(Skeleton,Color) started");
   if (stereo){
      pair< VecGraph, VecGraph > pvv=capt(sk);
      pvv.first.setColor(getColor1());
      pvv.second.setColor(getColor2());
      pvv.first.mark(gr);
```

```
      pvv.second.mark(gr);
   }
   else{
      VecGraph vgr=captMono(sk);
      vgr.setColor(c);
      vgr.mark(gr);
   }
   cpmmessage(mL,"Camera::mark(Skeleton,Color) done");
}

void Camera::mark(Triangle const& tri, Color const& c)const
{
   Z mL=3;
   Word loc("Camera::mark(Triangle,Color)");
   CPM_MA
   bool stereo=!c.visible()&&is3D()&&!mono;
   Skeleton sk(tri.p1(),tri.p2(),tri.p3());
   if (stereo){
      pair< VecGraph, VecGraph > pvv=capt(sk);
      pvv.first.setColor(getColor1());
      pvv.second.setColor(getColor2());
      pvv.first.markTriAng(gr);
      pvv.second.markTriAng(gr);
   }
   else{
      VecGraph vgr=captMono(sk);
      vgr.setColor(c);
      vgr.markTriAng(gr);
   }
   CPM_MZ
}


void Camera::mark(const WireFrame& wf, const Color& c)const
{
   Z dwf=wf.dim();
   for (Z i=1;i<=dwf;i++) mark(wf[i],c);
}


void Camera::mark(const RefSys& rs, const Color& c, R lengthOfAxes)const
{
   WireFrame wf=WireFrame::tripod(lengthOfAxes);
   Group g=rs.frmMasSys();
   wf*=g;
   mark(wf,c);
}

void Camera::mark(const SubsetOfSpace& sos, SitRec k,
   const Color& c)const
```

```
{
   Z mL=2;
   Word loc(nameOf()&"::mark(SubsetOfSpace,SitRec,Color)");
   CPM_MA
   const R fineFac=0.25;
   const R coarseFac=5;
   const R steepnessLimit=20;
      // factors for setting the method parameters of
      // SubsetOfSpace::hit()
      // in terms of the basic visual resolution as derived from
      // pixel size and/or sos.get_ds().
      // Same factors appear in Camera::lookTo
   R ds=sos.get_ds();
   R length=(k==LOW || k==UPP ? getdX() : getdY() );
      // length of the k-site of the window in object space
   Z nPix=(k==LOW || k==UPP ? dim1() : dim2() );
   Z np=1+(Z)(0.5+length/ds);
   np=CpmRootX::inf(np,nPix);
   if (np<=1)
      cpmerror("Camera::mark(SubsetOfSpace& ...): np<=1");
   R ds0=length/(np-1);
      // adjusted interval ds ~ ds0, ds0 fraction of length
   C vr2Start=( k==LOW||k==RIG ?
      C(getxRight(),getyLower()) : C(getxLeft(),getyUpper()) );
   V<C> vr2(np);
   vr2[1]=vr2Start;
   C step;
   if (k==LOW) step[1]=-ds0; else if (k==RIG) step[2]=ds0;
      else if (k==UPP) step[1]=ds0; else step[2]=-ds0;
   Z i;
   for (i=2;i<=np;i++) vr2[i]=vr2[i-1]+step;
   V<Spc> spc(np);
   R mag=1;
   for (i=1;i<=np;i++) spc[i]=inverse(vr2[i],mag);
   Vec dir;
   if (k==LOW) dir=v; else if (k==RIG) dir=-h;
     else if (k==UPP) dir=-v; else dir=h;
   R sFine=fineFac*ds0;
   R sCoarse=coarseFac*ds0;
   R dLim=steepnessLimit*ds0;
   std::vector<Spc> valid;
      // list of points to display. Class vector provides fast
      // appending at the end ('push_back')
   for (i=1;i<=np;i++){
      X2<Spc,bool> psz=sos.hit(spc[i],dir,sFine,sCoarse);
      if (psz.c2()){ // then the point is OK
         valid.push_back(psz.c1());
      }
   }
   Z nValid=(Z)valid.size();
```

```
   for (i=1;i<nValid;i++){
      Spc sa=valid[i-1];
      Spc sb=valid[i];
      R len=(sb-sa).absVal();
      bool ok=(len<dLim);
         // notice that the coordinate difference in scan direction
         // of adjacent points is just ds0. So we give no OK for
         // steep steps in projection direction.
         // If such steps happen to be real, then they become
         // regularly imaged for a new value of k
       if (ok)mark(sa,sb,c);
   }
   CPM_MZ
}

void Camera::markCen(const SubsetOfSpace& sos, Z nP, const Color& c)const
{
   Z mL=2;
   Word loc(nameOf()&"::markCen(SubsetOfSpace,Z,Color)");
   CPM_MA
   R ds=sos.get_ds();
   R dr=sos.get_dr();
   cpmassert(ds>0,loc);
   if (dr==0){
      dr=10*ds;
      cpmwarning(loc&": dr was 0, set to "&cpm(dr));
   }
   Spc sc=get_x();
   R dPhi=cpm2pi/nP;
   Vec dir=h;
   X2<Spc,bool> spb=sos.hit(sc,dir,ds,dr);
   Spc spFirst=spb.c1();
   bool succ=spb.c2();
   cpmassert(succ==true,loc);
   R phi=0;
   for (Z i=1;i<=nP;++i){
      phi+=dPhi;
      dir=h*cos(phi)+v*sin(phi);
      spb=sos.hit(sc,dir,ds,dr);
      Spc spAct=spb.c1();
      succ=spb.c2();
      cpmassert(succ==true,loc);
      mark(spFirst,spAct,c);
      spFirst=spAct;
   }
   CPM_MZ
}

void Camera::markFan(const SubsetOfSpace& sos, Spc const& p,
   Vec const& e, AxVec const& ax, R ang, Z nP, Color const& c )const
```

```
{
   Z mL=2;
   Word loc(nameOf()&"::markFan(...)");
   CPM_MA
   R r=sos.get_r();

   if (r==0){
      cpmwarning(loc&": sos.r==0 not useful, nothing done");
      CPM_MZ
      return;
   }
   if (nP<1){
      cpmwarning(loc&": np="&cpm(nP)&" not useful, nothing done");
      CPM_MZ
      return;
   }
   Vec dir=e;
   R dd=dir.normalize();
   if (dd==0.){
      cpmwarning(loc&": no direction defined, nothing done");
      CPM_MZ
      return;
   }
   R ds=sos.get_ds();
   R dr=sos.get_dr();
   cpmassert(ds>0,loc);
   if (dr==0){
      dr=10*ds;
      cpmwarning(loc&": dr was 0, set to "&cpm(dr));
   }
   X2<Spc,bool> spb;
   bool succ;
   if (ang==0.){ // only one direction, thus mark a point
         // (and not a line)
      cpmwarning(loc&": angle is zero, have to generate only one point");\

      spb=sos.hit(p,dir,ds,dr);
      succ=spb.c2();
      cpmassert(succ==true,loc&" at ang==0.");
      if (succ) mark(spb.c1(),c);
      CPM_MZ
      return;
   }
   AxVec axis=ax;
   R aa=axis.normalize();
   if (aa==0.){ // only one direction, thus mark a point
         // (and not a line)
      cpmwarning(loc&
         ": no valid rotation axis, have to generate only one point");
      spb=sos.hit(p,dir,ds,dr);
```

```
      succ=spb.c2();
      cpmassert(succ==true,loc&" at aa==0.");
      if (succ) mark(spb.c1(),c);
      CPM_MZ
      return;
   }
   Angle dPhi(ang/nP,RAD);
   AxVec rot(dPhi,axis);
   spb=sos.hit(p,dir,ds,dr);
   Spc spFirst=spb.c1();
   succ=spb.c2();
   cpmassert(succ==true,loc&" before i-loop");
   for (Z i=1;i<=nP;++i){
      dir*=rot;
      spb=sos.hit(p,dir,ds,dr);
      Spc spAct=spb.c1();
      succ=spb.c2();
      cpmassert(succ==true,loc&" i="&cpm(i));
      mark(spFirst,spAct,c);
      spFirst=spAct;
   }
   CPM_MZ
}

void Camera::rot(const AxVec& axv)
{
   Group g(poi,axv);
   operator *=(g);
}

RefSys Camera::getRefSys()const
{
   Z d=Spc::dimension();
   RefSys master;
   V<Vec> b1=master.getVectors();
   V<Vec> b2=getVectors();
   Vec a=poi-Spc();
   AxVec axv(b1,b2,false); // bases are OK, have not to be rebuilt
   Group g(axv,a);
   return master*=g;
}

bool Camera::isVisible( const Spc& x)const
{
   C px=captureMono(x);
   return gr.indFunc(R2(px));
}

Camera::Camera(const RecordHandler& rch, const Word& sec,
   Frame& fr)
```

```
{
   const Z mL=2;
   Word loc(nameOf()&"::Camera(RecordHandler,Word,Frame)");
   cpmmessage(mL,loc&" started");
   R fieldOfViewHeight=1, fieldOfViewCenterX=0;
   R fieldOfViewCenterY=0, fieldOfViewCenterZ=0;
   R tiltDeg=0, panDeg=0, rollDeg=0, shiftX=0, shiftY=0;
   R viewingAngleDegree=20, convergenceAngleDegree=3.5, frac=1,
      aspectRatio=1;
   R responseGamma=-1,responseLowerThreshold=0,responseUpperThreshold=1;
   Z yPixel=400,responseMode=6;
   bool writeOnShow=false;
   bool isMono=false;
   cpmrhf(isMono);
   Z ped=1;
// so one needs not to read any of the following quantities from file
   rch.read(sec,"fieldOfViewHeight", fieldOfViewHeight,ped);
   rch.read(sec,"fieldOfViewCenterX", fieldOfViewCenterX,ped);
   rch.read(sec,"fieldOfViewCenterY", fieldOfViewCenterY,ped);
   rch.read(sec,"fieldOfViewCenterZ", fieldOfViewCenterZ,ped);
   rch.read(sec,"shiftX", shiftX,ped);
   rch.read(sec,"shiftY", shiftY,ped);
   rch.read(sec,"tiltDeg",tiltDeg,ped);
   rch.read(sec,"panDeg",panDeg,ped);
   rch.read(sec,"rollDeg",rollDeg,ped);
   rch.read(sec,"viewingAngleDegree", viewingAngleDegree,ped);
   rch.read(sec,"convergenceAngleDegree", convergenceAngleDegree,ped);
   rch.read(sec,"frac", frac,ped);
   rch.read(sec,"aspectRatio", aspectRatio,ped);
   rch.read(sec,"writeOnShow", writeOnShow,ped);
      // entry not available in ini-files older than 03-01-23
   rch.read(sec,"yPixel", yPixel,ped);
   if (yPixel<1){ // 'deactivates' the camera if it is
         // introduced to a program by calling this constructor
      *this=Camera();
      return;
   }
   rch.read(sec,"responseMode", responseMode,ped);
   rch.read(sec,"responseGamma", responseGamma,ped);
   rch.read(sec,"responseLowerThreshold", responseLowerThreshold,ped);
   rch.read(sec,"responseUpperThreshold", responseUpperThreshold,ped);

   fr.cut(aspectRatio,yPixel); // this places fr

   Angle viewingAngle(viewingAngleDegree);
   Angle conv(convergenceAngleDegree);
   *this=Camera(fieldOfViewHeight,aspectRatio,viewingAngle,frac,conv,fr);
   rsp.setGamma(responseGamma);
   rsp.setMode(responseMode);
   rsp.setRangeX(responseLowerThreshold,responseUpperThreshold);
```

```
   rsp.setRangeY(0,255);
   shift(fieldOfViewCenterX, fieldOfViewCenterY, fieldOfViewCenterZ);
   shiftGate(shiftX,shiftY);
   pan(Angle(panDeg));
   tilt(Angle(tiltDeg));
   roll(Angle(rollDeg));
   V<Z> rgbBackground;
   bool succ=rch.read(sec,"rgbBackground", rgbBackground,1);
   if (succ) color3=Color(rgbBackground);
   if (Vec::dimension==2 || isMono) gr.setColors();
      // colors are bodycolors, 'film is assumed to be virginal'
      // i.e. no background has to be taken into account.
      // colors get simply set
   else gr.stereo(); // in 3D always fast stereo
   gr.setWriteOnShow(writeOnShow);
   if (isMono) mono_();
   gr.setColorActive(color3);
   gr.paint(color3);
   if (cpmverbose>=mL){
      ostringstream ost;
      ost<<loc<<" done: "<<"dim1="<<gr.dim1()<<" dim2="<<gr.dim2();
      cpmmessage(ost);
   }
}

void Camera::setScreenArea( const Graph& g)const
{
   Graph gTemp=g;
   gTemp.getObjectSpaceFrom(gr);
   gr=gTemp;
}

V<Spc> Camera::grid(Z ny)const
{
   Z mL=3;
   Word loc("Camera::grid(Z)");
   cpmmessage(mL,loc&" started");
   Z nx=(Z)(ny*gr.getAspRat());
   Iv ivx=gr.getIvX();
   Iv ivy=gr.getIvY();
   V<R> vx=ivx.centers(nx);
   V<R> vy=ivy.centers(ny);
   V<Spc> res(nx*ny);
   Z i,j;
   C xij;
   Z count=0;
   for (i=1;i<=nx;i++){
      xij[1]=vx[i];
      for (j=1;j<=ny;j++){
         xij[2]=vy[j];
```

```
        R magFac=1.;
        res[++count]=inverse(xij,magFac);
      }
   }
   cpmmessage(mL,loc&" done");
   return res;
}

void Camera::lookTo(const SubsetOfSpace& sos, Angle tilt, Angle pan,
   const Texture& tx, const RefSys& rs, bool bodyFix, R fitFactor)
{
   dirTo_(sos,fitFactor,tilt,pan,rs);
   show(sos,tx,bodyFix);
}
```

31 cpmclinalg.h

```
//? cpmclinalg.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_CLINALG_H_
#define CPM_CLINALG_H_
/*
   Description: classes C1, C2, C3, C4, CC2, CC3, CC4
   Basis for effective geometry in 1,2,3,4-dimensional spaces.

*/
#include <cpmlinalg.h>

namespace CpmLinAlg{

   using namespace CpmStd;

   using CpmRoot::R;
   using CpmRoot::Pi;
   using CpmRoot::Pi2;
   using CpmRoot::PiInv;
   using CpmRoot::PiHalf;
   using CpmRoot::Z;
   using CpmRoot::C;
   using CpmRoot::Word;
   using CpmArrays::V;

//////////////////////////// class C2 ////////////////////////////////
class C4;
class CC2;

class C2{  // tuples of two complex elements
   // In physics this is also the state space of the spin degree of
   // freedom for spin 1/2. Some of the functions to be defined here deal
   // with this situation.
public:
   C x1;
   C x2;
   void ini_(R theta, R phi){
      R ph=phi*0.5, th=theta*0.5;
      x1=C::expi(-ph)*::cos(th);
      x2=C::expi(ph)*::sin(th);
   }
public:
   friend class C4;
   friend class CC2;
   typedef C2 Type;
```

```
// constructors
   C2():x1(),x2(){}
   C2(C const& a, C const& b):x1(a),x2(b){}
   explicit C2(const V<C>&); // construction from dimension-independent
   // lists
      // no restiction to the argument
   explicit C2(R theta, R phi=0){ini_(theta,phi);}
      // construction from the usual polar coordinates on the Bloch sphere
      // Gives always a unit vector. phi does not change the proportion
      // |x1| : |x2|. theta==0 => x2=0, ('north pole')
      // theta==pi => x1=0 ('south pole')
      // see function blcAng for getting the angles from a C2 object.

   C2(R theta, R phi, R chi){
      ini_(theta,phi);
      C omega=C::expi(chi);
      x1*=omega;
      x2*=omega;
   }

// access
   C& operator[](Z i){ if (i>=2) return x2; else return x1;}
   C const& operator[](Z i)const{ if (i>=2) return x2; else return x1;}
   void set(C t){ x1=t; x2=t;}
// mutating arithmetics (linear algebra)
   C2& operator +=(C2 const& a)
   { x1+=a.x1; x2+=a.x2; return *this;}
   C2& operator -=(C2 const& a)
   { x1-=a.x1; x2-=a.x2; return *this;}
   C2& operator *=(C const& r)
   { x1*=r; x2*=r; return *this;}
   C2& operator *=(R const& r)
   { x1*=r; x2*=r; return *this;}
   C2& neg(void)
   { x1=-x1; x2=-x2; return *this;  }
   C operator |(C2 const& a)const{ return ~x1*a.x1+~x2*a.x2;}
   C trc(C2 const& a)const{ return x1*a.x1+x2*a.x2;}
   R absVal(bool careful=true)const;
   R absSqr()const{ return x1.absSqr()+x2.absSqr();}
   R disVal(C2 const&)const;
   C2 ranVal(Z j=0)const{
      if (j==0) return C2(x1.ranVal(),x2.ranVal());
      else{
         Z j2=j*2; return C2(x1.ranVal(j2),x2.ranVal(j2+1));
      }
   }
   C2 con()const{ return C2(x1.con(),x2.con());}
   C2 swap()const{ return C2(x2,x1);}
      //: swap
   C2& swap_(){ C temp=x1; x1=x2; x2=temp; return *this;}
```

```
   //: swap
C2 operator-()const{ return C2(-x1,-x2);}
pair<R,C2 > polDec()const;
   //: polar decomposition
   // First component (res.first if res is the
   // return value) is the absolute value r and the second component
   // a unit vector n such that *this = n*r. If *this is the zero
   // vector, n is arbitrarily chosen as (1,0)
R normalize()
   { pair<R,C2> p=polDec(); *this=p.second;  return p.first;}
   // turns *this into a unit vector and returns the norm of the
   // original in order not to loose information
R nor_(){ return normalize();}
   //: normalize

R nor1_(){ R a=x1.arg(); C w=C::expi(-a); operator*=(w); return a;}
   //: normalize 1 (i.e. component 1)
   // normalizing means making real in this case

R2 blcAng()const;
   //: bloch angles R2(theta,phi)

C blc()const{ R2 ba=blcAng(); return C::expi(ba[2])*(Pi-ba[1])*PiInv;}
   // Bloch angles coded as a single C.
   // Usefull for conversion to display color
   // Then theta modulates luminance: theta=0 (north pole) bright
   // theta=Pi (south pole) dark. Phi determines hue along
   // the color circle.

R3 toR3()const
   //: to R3
   // returns (y1,y2,y3) with 0<=yi<=1;
   // Let C2n := { c \in C2 | c.absVal()==1}
   // and R3n := { r \in R3 | r[1],r[2],r[3] \in Iv(0,1)}
   // then c |--> c.toR3() defines a surjective and continuous
   // mapping A: C2n --> R3n. As a surjective mapping it has a
   // right inverse.
   // The main reason for considering this function (and the next)
   // is that R3 can be naturally converted to display color.
   // See class CpmImaging::Color.
{
   Word loc("C2::toR3()");
   R r1,r2,phi1,phi2;
   x1.pol_1_2(r1,phi1);
   x2.pol_1_2(r2,phi2);
   R phi3=C(r1,r2).arg();
   R y1=(::sin(phi1)+1.)*0.5;
   R y2=(::sin(phi2)+1.)*0.5;
   R y3=::sin(phi3);
   return R3(y1,y2,y3);
```

```
   }

   C2(R3 const& y)
      // constructor from y \in R3n (see previous function).
      // This function B: R3n --> C2n, y |--> C2n(y) is the
      // right inverse of A (as defined above) i.e.
      // A(B(y)==y
   {
      Word loc("C2(R3)");
      R r2=y[3];
      R r1=::sqrt(1-r2*r2);
      R phi1=::asin(2*y[1]-1);
      R phi2=::asin(2*y[2]-1);
      x1=C::pol(r1,phi1);
      x2=C::pol(r2,phi2);
   }

   C2 operator*(R r)const{ return C2(x1*r,x2*r);}

   C sel(bool second)const{ return second ? x2 : x1;}

   static R blcTest(Z n=100);
      //: Bloch test
      // tests with random input whether
      // for any R2 r, we have z ~ z' for
      // z:=C2(r), r':= z.blcAng, z':=C2(r')
      // '~' means equality up to a common phase.
      // The accumulated error (which should be numerical
      // noise) is returned. Worked with result ~ 1e-13
      // for n=1000 (2007-12-05)

   static R codTest(Z n=1000);
      //: coding test
      // tests whether going from R3 to C2 and back to R3 gives
      // the identity (see C2(R3 const& y) for the context).

   CPM_IO
   CPM_ORDER
   CPM_NAM(C2)
   Z dim()const{ return 2;}
      //: dimension
};

inline C2 operator +(C2 const& x1, C2 const& x2)
{ C2 res=x1; return res+=x2;}

inline C2 operator -(C2 const& x1, C2 const& x2)
{ C2 res=x1; return res-=x2;}

inline C2 operator -(C2 const& x)
```

```
{ C2 res=x; return res.neg();}

inline C2 operator *(C2 const& x1, C const& s)
{ C2 res=x1; return res*=s;}

inline C2 operator *(C const& s, C2 const& x1)
{ C2 res=x1; return res*=s;}

///////////////////////////// class C3 ////////////////////////////////

class C3{ // tuples of 3 complex elements
public:
   typedef C3 Type;
   C x1;
   C x2;
   C x3;
public:
// constructors
   C3():x1(0.),x2(0.),x3(0.){}
   C3(C const& a, C const& b, C const& c):x1(a),x2(b),x3(c){}
   C3(const V<C>&); // construction from dimension-independent lists
      // no restiction to the argument
// access
   C& operator[](Z i)
   { if (i>=3) return x3; else if (i==2) return x2; else return x1;}
   C const& operator[](Z i)const
   { if (i>=3) return x3; else if (i==2) return x2; else return x1;}
   void set(C t){ x1=t; x2=t; x3=t;}
// mutating arithmetics (linear algebra)
   C3& operator +=(C3 const& a)
   { x1+=a.x1; x2+=a.x2; x3+=a.x3; return *this;}
   C3& operator -=(C3 const& a)
   { x1-=a.x1; x2-=a.x2; x3-=a.x3; return *this;}
   C3& operator *=(C const& r)
   { x1*=r; x2*=r; x3*=r; return *this;}
   C3& operator *=(const R& r)
   { x1*=r; x2*=r; x3*=r; return *this;}
   C3& neg(void)
   { x1=-x1; x2=-x2; x3=-x3; return *this;  }
   C trc(C3 const& a)const{ return x1*a.x1+x2*a.x2+x3*a.x3;}
   C operator |(C3 const& a)const
   { return ~x1*a.x1+~x2*a.x2+~x3*a.x3;}
   R absVal(bool careful=true)const;
   R disVal(C3 const&)const;
   pair<R,C3 > polDec()const;
      // polar decomposition: First component (res.first if res is the
      // return value) is the absolute value r and the second component
      // a unit vector n such that *this = n*r. If *this is the zero
      // vector, n is arbitrarily chosen as (1,0,0)
   R normalize()
```

```
   { pair<R,C3> p=polDec(); *this=p.second;  return p.first; }
      // turns *this into a unit vector and returns the norm of the
      // original in order not to loose information
   R nor_(){ return normalize();}
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 3;}
};

inline C3 operator +(C3 const& x1, C3 const& x2)
{ C3 res=x1; return res+=x2;}

inline C3 operator -(C3 const& x1, C3 const& x2)
{ C3 res=x1; return res-=x2;}

inline C3 operator -(C3 const& x)
{ C3 res=x; return res.neg();}

inline C3 operator *(C3 const& x1, C const& s)
{ C3 res=x1; return res*=s;}

inline C3 operator *(C const& s, C3 const& x1)
{ C3 res=x1; return res*=s;}

//////////////////////////// class C4 ////////////////////////////////

class C4{  //  tuples of 4 complex elements
public:
   typedef C4 Type;
   C x1;
   C x2;
   C x3;
   C x4;
public:
// constructors
   C4():x1(0),x2(0),x3(0),x4(0){}
   C4(C const& a, C const& b, C const& c, C const& d):x1(a),x2(b),
      x3(c),x4(d){}
   C4(const V<C>&); // construction from dimension-independent lists
      // no restiction to the argument
   C4(C2 const& a, C2 const& b):x1(a.x1*b.x1),x2(a.x1*b.x2),x3(a.x2*b.x1),
      x4(a.x2*b.x2){}
      // Here we consider C4 as the tensor product C2 times C2 (not the
      // direct sum.
   C4(Word const& idle, C2 const& a, C2 const& b):x1(a.x1),x2(a.x2),
      x3(b.x1),x4(b.x2){}
      // Here we consider C4 as the direct sum of  C2 and C2
      // The dummy argument is to discern this (less important)
      // construct from the more canonical tensor product
// access
```

```
   C& operator[](Z i)
   { if (i>=4) return x4; else if (i==3) return x3;
      else if (i==2) return x2; else return x1;}
   C const& operator[](Z i)const
   { if (i>=4) return x4; else if (i==3) return x3;
      else if (i==2) return x2; else return x1;}
   void set(C t){ x1=t; x2=t; x3=t; x4=t;}
// mutating arithmetics (linear algebra)
   C4& operator +=(C4 const& a)
   { x1+=a.x1; x2+=a.x2; x3+=a.x3; x4+=a.x4; return *this;}
   C4& operator -=(C4 const& a)
   { x1-=a.x1; x2-=a.x2; x3-=a.x3; x4-=a.x4; return *this;}
   C4& operator *=(C const& r)
   { x1*=r; x2*=r; x3*=r; x4*=r; return *this;}
   C4& operator *=(const R& r)
   { x1*=r; x2*=r; x3*=r; x4*=r; return *this;}
   C4& neg(void)
   { x1=-x1; x2=-x2; x3=-x3; x4=-x4; return *this;  }
   C operator |(C4 const& a)const
   { return ~x1*a.x1+~x2*a.x2+~x3*a.x3+~x4*a.x4;}
   C trc(C4 const& a)const{ return x1*a.x1+x2*a.x2+x3*a.x3+x4*a.x4;}
   R absVal(bool careful=true)const;
   R disVal(C4 const&)const;
   pair<R,C4 > polDec()const;
      // polar decomposition: First component (res.first if res is the
      // return value) is the absolute value r and the second component
      // a unit vector n such that *this = n*r. If *this is the zero
      // vector, n is arbitrarily chosen as (1,0,0,0)
   R normalize(){ pair<R,C4> p=polDec(); *this=p.second;
      return p.first; }
      // turns *this into a unit vector and returns the norm of the
      // original in order not to loose information
   R nor_(){ return normalize();}
   C4 con()const{ return C4(~x1,~x2,~x3,~x4);}
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 4;}
};

inline C4 operator +(C4 const& x1, C4 const& x2)
{ C4 res=x1; return res+=x2;}

inline C4 operator -(C4 const& x1, C4 const& x2)
{ C4 res=x1; return res-=x2;}

inline C4 operator -(C4 const& x)
{ C4 res=x; return res.neg();}

inline C4 operator *(C4 const& x1, C const& s)
{ C4 res=x1; return res*=s;}
```

```
inline C4 operator *(C const& s, C4 const& x1)
{ C4 res=x1; return res*=s;}

//////////// lean square matrices (2*2) - (4*4) ////////////////////////

#define CPM_MAT_ALG(X,XX)\
friend X operator*(const XX&, const X&);\
friend XX operator*(const XX& , const XX&);\
friend XX operator+(const XX& , const XX& );\
friend XX operator~(const XX& );\
friend X operator*(const X& x, const XX& a){ return ~a*x;}

// comment to the last function where matrices are acting on vectors
// from the right side. This kind of action on a vector space gives a
// representation of the multiplicative monoid of matrices if the product
// law for endomorphisms of linear spaces is defined as contravariant.
// In our specific notation here:
// x*(a*b)=(x*a)*b
// Proof: x*(a*b)=~(a*b)*x=((~b)*(~a))*x=(~b)*((~a)*x)=((~a)*x)*b=(x*a)*b

/// notice: operator ~ not yet analysed and tested for the complex case
// !!!!!!!!!

//////////////////////////// class CC2 ////////////////////////////////
class CCC2;
class CCCC2;
class CC2{  //  complex 2*2 matrices
public:
   friend class CCC2;
   friend class CCCC2;
   typedef CC2 Type;
   C2 x1;
   C2 x2;
   static CC2 zero;
   static CC2 unit;
   static CC2 sig0,sig1,sig2,sig3;
      // Pauli sigma matrices, also named sigma_x, sigma_y, sigma_z;
      // sig0 is the unit matrix.
// constructors
   CC2():x1(),x2(){}
   CC2( C2 const& a1, C2 const& a2):x1(a1),x2(a2){}
   CC2(C z11, C z12, C z21, C z22):x1(z11,z12),x2(z21,z22){}
// access
   C2& operator[](Z i){ if (i>=2) return x2; return x1;}
   C2 const& operator[](Z i)const{ if (i>=2) return x2; return x1;}
   C det()const{ return x1.x1*x2.x2-x1.x2*x2.x1;}
   CC2 operator !()const;
      // inverse matrix can be expressed easily in this case
   CC2 her()const;
```

```
    //: hermitean (conjugate)
    // returns the hermitean conjugate
 CC2 trn()const;
    //: transpose
 C operator *(C z)const{ return (x1.x1*z+x1.x2)/(x2.x1*z+x2.x2);}
    // CC2 acts on C as a 'broken linear transformation'
 C4 toC4()const{ return C4(x1.x1,x1.x2,x2.x1,x2.x2);}
    // one has: CC2 x => x.toC4().toCC2()==x
    // and:     C4 y => y.toCC2().toC4()==y
// C2 trc1(C2 const& a)const;
    //
 C2 trc2(C2 const& a)const{
    C u1=a.x1.con();
    C u2=a.x2.con();
    return C2(u1*x1.x1+u2*x1.x2,u1*x2.x1+u2*x2.x2);
 }

 C2 trc1(C2 const& a)const{
    C u1=a.x1.con();
    C u2=a.x2.con();
    return C2(u1*x1.x1+u2*x2.x1,u1*x1.x2+u2*x2.x2);
 }

 C trc()const{ return x1.x1+x2.x2;}
    //: trace

 C app(C2 const& a, bool second)const
    //: apply
    // returns the first or the second component of (*this)*a
    // depending of the speakingly named second argument.
    // A boolean argument is chosen here since it is available in this
    // form most directly in function Qreg::app_
 {
    return
       second ? x2.x1*a.x1+x2.x2*a.x2 : x1.x1*a.x1+x1.x2*a.x2;
 }

 C biLinForm(C2 const& a, C2 const& b)const
    //: bi-linear form
 {
    return x1.x1*a.x1*b.x1 +
           x1.x2*a.x1*b.x2 +
           x2.x1*a.x2*b.x1 +
           x2.x2*a.x2*b.x2 ;
 }

 C operator()(bool i, bool j)const{
    if (i && j)  return x2.x2;
    if (i && !j) return x2.x1;
    if (!i && j) return x1.x2;
```

```
      if (!i && !j)return x1.x1;
      return C();
   }

   CPM_IO
   CPM_MAT_ALG(C2,CC2)

   CC2 operator*(R r)const
   {
      return CC2(x1*r,x2*r);
   }

   static CC2 dydPrd(C2 const& a, C2 const& b)
      //: dyadic product
      // building a matrix as a dyadic product of vectors
      // (tensor product)
   {
      CC2 r;
      C2 c=b.con();
      r.x1.x1=a.x1*c.x1;
      r.x1.x2=a.x1*c.x2;
      r.x2.x1=a.x2*c.x1;
      r.x2.x2=a.x2*c.x2;
      return r;
   }
};

//////////////////////////// class CC3 ////////////////////////////////

class CC3{  //  complex 3*3 matrices
public:
   typedef CC3 Type;
   C3 x1;
   C3 x2;
   C3 x3;
public:
   static CC3 zero;
   static CC3 unit;
// constructors
   CC3(){}
   CC3(C3 const& a1, C3 const& a2, C3 const& a3):x1(a1),x2(a2),x3(a3){}
// access
   C3& operator[](Z i)
   { if (i>=3) return x3; if (i==2) return x2; return x1;}

   C3 const& operator[](Z i)const
   { if (i>=3) return x3; if (i==2) return x2; return x1;}

   CPM_IO
   CPM_MAT_ALG(C3,CC3)
```

```
};

//////////////////////////// class CC4 ////////////////////////////////

class CC4{  // complex 4*4 matrices
public:
   typedef CC4 Type;
   C4 x1;
   C4 x2;
   C4 x3;
   C4 x4;
public:
   static CC4 zero;
   static CC4 unit;
// constructors
   CC4(){}
   CC4(C4 const& a1, C4 const& a2, C4 const& a3, C4 const& a4):
      x1(a1),x2(a2),x3(a3),x4(a4){}

   CC4(C4 const& a, C4 const& b):x1(a.x1*b),x2(a.x2*b),
      x3(a.x3*b),x4(a.x4*b){}
         // terse but correct coding of c_ij = a_i * b_j

   CC4(CC2 const& a, CC2 const& b)
   {
      *this=CC4(a.toC4(),b.toC4());
   }

// access
   C4& operator[](Z i)
      { if (i>=4) return x4; if (i==3) return x3; if (i==2) return x2;
       return x1;}

   C4 const& operator[](Z i)const
       { if (i>=4) return x4; if (i==3) return x3; if (i==2) return x2;
        return x1;}

   CPM_IO
   CPM_MAT_ALG(C4,CC4)
};
/// special topic: tensor products of C2: CCC2, CCCC2
/// for my spin interaction application I do not longer need CC4
/// and use CCCC2 instead

class CCC2{ // pairs of CC2
   CC2 x1,x2;
   CCC2(CC2 const& y1,CC2 const& y2):x1(y1),x2(y2){}
public:
   friend class CCCC2;
   CCC2():x1(),x2(){}
```

```
   CCC2 operator + (CCC2 const& a)const{ return CCC2(x1+a.x1,x2+a.x2);}
   CC2 sel(bool second)const{ return second ? x2 : x1;}
      //: select
   CCC2 operator* (R r)const
   {
      return CCC2(x1*r,x2*r);
   }
};

class CCCC2{ // pairs of CCC2, for implementation
   // of tensor products of sigma matrices
   CCC2 x1,x2;
   CCCC2(CCC2 const& y1,CCC2 const& y2):x1(y1),x2(y2){}
public:
   CCCC2():x1(),x2(){}
   CCCC2(CC2 const& a, CC2 const& b):x1(),x2()
   {
      x1.x1.x1.x1 = a.x1.x1 * b.x1.x1;
      x1.x1.x1.x2 = a.x1.x1 * b.x1.x2;
      x1.x1.x2.x1 = a.x1.x1 * b.x2.x1;
      x1.x1.x2.x2 = a.x1.x1 * b.x2.x2;

      x1.x2.x1.x1 = a.x1.x2 * b.x1.x1;
      x1.x2.x1.x2 = a.x1.x2 * b.x1.x2;
      x1.x2.x2.x1 = a.x1.x2 * b.x2.x1;
      x1.x2.x2.x2 = a.x1.x2 * b.x2.x2;

      x2.x1.x1.x1 = a.x2.x1 * b.x1.x1;
      x2.x1.x1.x2 = a.x2.x1 * b.x1.x2;
      x2.x1.x2.x1 = a.x2.x1 * b.x2.x1;
      x2.x1.x2.x2 = a.x2.x1 * b.x2.x2;

      x2.x2.x1.x1 = a.x2.x2 * b.x1.x1;
      x2.x2.x1.x2 = a.x2.x2 * b.x1.x2;
      x2.x2.x2.x1 = a.x2.x2 * b.x2.x1;
      x2.x2.x2.x2 = a.x2.x2 * b.x2.x2;
   }

   CCCC2 operator + (CCCC2 const& a)const{ return CCCC2(x1+a.x1,x2+a.x2);}

// the isotropic interaction term between two spin 1/2
// magnetic moments (or Heisenberg exchange interaction
// between spins)
   static CCCC2 spinSpin(R q, bool full=true){
      CCCC2 resPrel=CCCC2(CC2::sig1*q,CC2::sig1)
              + CCCC2(CC2::sig2*q,CC2::sig2);
      if (full) return resPrel+CCCC2(CC2::sig3*q,CC2::sig3);
      else return resPrel;
   }
```

```
   static CCCC2 spinSpinXY(){ return spinSpin(false);}

   CC2 operator()(C2 const& a, C2 const& b)const
   {
      CC2 res;
      res.x1.x1= x1.x1.x1.x1 * a.x1 * b.x1
               + x1.x1.x1.x2 * a.x1 * b.x2
               + x1.x1.x2.x1 * a.x2 * b.x1
               + x1.x1.x2.x2 * a.x2 * b.x2 ;

      res.x1.x2= x1.x2.x1.x1 * a.x1 * b.x1
               + x1.x2.x1.x2 * a.x1 * b.x2
               + x1.x2.x2.x1 * a.x2 * b.x1
               + x1.x2.x2.x2 * a.x2 * b.x2 ;

      res.x2.x1= x2.x1.x1.x1 * a.x1 * b.x1
               + x2.x1.x1.x2 * a.x1 * b.x2
               + x2.x1.x2.x1 * a.x2 * b.x1
               + x2.x1.x2.x2 * a.x2 * b.x2 ;

      res.x2.x2= x2.x2.x1.x1 * a.x1 * b.x1
               + x2.x2.x1.x2 * a.x1 * b.x2
               + x2.x2.x2.x1 * a.x2 * b.x1
               + x2.x2.x2.x2 * a.x2 * b.x2 ;

      return res;
   }

   CCC2 sel(bool second)const{ return second ? x2 : x1;}
      //: select

 //  CCCC2 operator* (R r)const{ return CCCC2(x1*r,x2*r);}
   // dubious; due to the intended tensor product character
   // only one factor should be multiplied; no longer needed

   CC2 operator * (CC2 const& a)const;
      // strangling the second and the fourth index of *this by the
      // 2*2 matrix a and thus getting a 2*2 matrix, which gets returned.

   C app(CC2 const& a, bool b1, bool b2)const;
      // returns the matrix element (i,j) of (*this)*a
      // where
      // i=1 for b1=false, i=2 for b1=true
      // j=1 for b2=false, j=2 for b2=true
      // Only those products and sums are computed which
      // make up the requested matrix element. Thus if not
      // all matrix elements of (*this)*a are needed, we do less
      // work than computing the whole matrix (*this)*a.
};
```

```
#undef CPM_MAT_ALG

} // namespace

#endif
```

32 cpmclinalg.cpp

```
//? cpmclinalg.cpp
//? Status of work 2008-10-25.
//?
#include <cpmclinalg.h>
#include <cpmlinalg.h>

using namespace CpmRoot;
using namespace CpmLinAlg;
using CpmRoot::distFunc;

//////////////////////////// class C2 ////////////////////////////////////

C2::C2(const V<C>& v)
{
   Z d=v.dim();
   if (1<=d) x1=v[1];
   if (2<=d) x2=v[2];
}

bool C2::prnOn(ostream& out)const
{
   if (!CpmRoot::writeTitle("C2",out)) return false;
   if (!x1.prnOn(out)) return false;
   if (!x2.prnOn(out)) return false;
   return true;
}

bool C2::scanFrom(istream& in)
{
   if (!x1.scanFrom(in)) return false;
   if (!x2.scanFrom(in)) return false;
   return true;
}

Z C2::com(const C2& a)const
{
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   if (x2<a.x2) return 1;
   if (x2>a.x2) return -1;
   return 0;
}

R C2::absVal(bool careful)const
{
   CpmLinAlg::R2 r(x1.absVal(),x2.absVal());
   return r.absVal(careful);
```

```
}

R C2::disVal(const C2& x)const
{
   R a=absVal();
   R b=x.absVal();
   R c=(*this-x).absVal();
   return distFunc(a,b,c);
}

std::pair<R,C2 > C2::polDec()const
{
   R r=absVal(true);
   C2 pol=*this;
   if (r!=0){
     R rInv=1./r; pol*=rInv;
     return pair<R,C2 >(r,pol);
   }
   else{
      return pair<R,C2 >(0.,C2(C(1),C()));
   }
}

R2 C2::blcAng()const
{
   R r1,phi1,r2,phi2;
   x1.pol_1_2(r1,phi1);
   x2.pol_1_2(r2,phi2);
   R thetah=C(r1,r2).arg();
   R phi=phi2-phi1;
   if (phi<0) phi+=Pi2;
   return R2(thetah*2,phi);
}

/*
// instructive but inefficient implementation
R2 C2::blcAng()const
{
   CC2 m=CC2::dydPrd(*this,*this);
   R u1=(m*CC2::sig1).trc()[1];
   R u2=(m*CC2::sig2).trc()[1];
   R u3=(m*CC2::sig3).trc()[1];
   R3 u(u1,u2,u3);
   R3 res=u.carToPol();
   return R2(res[2],Pi-res[3]);
}
*/

R C2::blcTest(Z n)
{
```

```
   const R a=100;
   R2 r0(-a,a);
   R err=0;
   for (Z i=1;i<=n;++i){
      R2 r1=r0.ranVal(i); // random pair of arguments
      C2 z1(r1[1],r1[2]);
      z1.nor1_(); // rephase to make first component real
      R2 r2=z1.blcAng();
      C2 z2(r2[1],r2[2]);
      z2.nor1_(); // rephase to make first component real
         // now z1 and z2 should differ only by numerical noise
      err+=z1.disVal(z2);
   }
   return err;
}

/**********
 R C2::codTest(Z n)
 {
    C z1(1);
    C z2(1,-1);
    C2 a(z1,z2);
    R err=0;
    for (Z i=1;i<=n;++i){
       C2 b=a.ranVal(i);
       b.nor_();
       R3 r3=b.toR3();
       C2 c(r3);
       R d=b.disVal(c);
       err+=d;
       cpmdebug(b);
       cpmdebug(c);
       cpmdebug(d);
    }
    return err;
 }
 ************/

R C2::codTest(Z n)
 {
    R err=0;
    for (Z i=1;i<=n;++i){
       R r1=CpmRoot::randomR();
       R r2=CpmRoot::randomR();
       R r3=CpmRoot::randomR();
       C2 a(R3(r1,r2,r3));
       R3 b=a.toR3();
       R d=a.disVal(b);
       err+=d;
       cpmdebug(a);
```

```
        cpmdebug(b);
        cpmdebug(d);
     }
     return err;
  }

//////////////////////////// class C3 ////////////////////////////////////

C3::C3(const V<C>& v)
{
   Z d=v.dim();
   if (1<=d) x1=v[1];
   if (2<=d) x2=v[2];
   if (3<=d) x2=v[3];
}

bool C3::prnOn(ostream& out)const
{
   if (!CpmRoot::writeTitle("C3",out)) return false;
   if (!x1.prnOn(out)) return false;
   if (!x2.prnOn(out)) return false;
   if (!x3.prnOn(out)) return false;
   return true;
}

bool C3::scanFrom(istream& in)
{
   if (!x1.scanFrom(in)) return false;
   if (!x2.scanFrom(in)) return false;
   if (!x3.scanFrom(in)) return false;
   return true;
}


Z C3::com(const C3& a)const
{
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   if (x2<a.x2) return 1;
   if (x2>a.x2) return -1;
   if (x3<a.x3) return 1;
   if (x3>a.x3) return -1;
   return 0;
}

R C3::absVal(bool careful)const
{
   CpmLinAlg::R3 r;
   r.x1=x1.absVal();
   r.x2=x2.absVal();
```

```
   r.x3=x3.absVal();
   return r.absVal(careful);
}

R C3::disVal(const C3& x)const
{
   R a=absVal();
   R b=x.absVal();
   R c=(*this-x).absVal();
   return distFunc(a,b,c);
}

std::pair<R,C3 > C3::polDec()const
{
   R r=absVal(true);
   C3 pol=*this;
   if (r!=0){
     R rInv=1./r; pol*=rInv;
     return pair<R,C3 >(r,pol);
   }
   else{
     return pair<R,C3 >(0.,C3(C(1),C(),C()));
   }
}

//////////////////////////// class C4 ////////////////////////////////////

C4::C4(const V<C>& v)
{
   Z d=v.dim();
   if (1<=d) x1=v[1];
   if (2<=d) x2=v[2];
   if (3<=d) x2=v[3];
   if (4<=d) x2=v[4];
}

bool C4::prnOn(ostream& out)const
{
   if (!CpmRoot::writeTitle("C4",out)) return false;
   if (!x1.prnOn(out)) return false;
   if (!x2.prnOn(out)) return false;
   if (!x3.prnOn(out)) return false;
   if (!x4.prnOn(out)) return false;
   return true;
}

bool C4::scanFrom(istream& in)
{
   if (!x1.scanFrom(in)) return false;
   if (!x2.scanFrom(in)) return false;
```

```
   if (!x3.scanFrom(in)) return false;
   if (!x4.scanFrom(in)) return false;
   return true;
}

Z C4::com(const C4& a)const
{
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   if (x2<a.x2) return 1;
   if (x2>a.x2) return -1;
   if (x3<a.x3) return 1;
   if (x3>a.x3) return -1;
   if (x4<a.x4) return 1;
   if (x4>a.x4) return -1;
   return 0;
}

R C4::absVal(bool careful)const
{
   CpmLinAlg::R4 r;
   r.x1=x1.absVal();
   r.x2=x2.absVal();
   r.x3=x3.absVal();
   r.x4=x4.absVal();
   return r.absVal(careful);
}

R C4::disVal(const C4& x)const
{
   R a=absVal();
   R b=x.absVal();
   R c=(*this-x).absVal();
   return distFunc(a,b,c);
}

std::pair<R,C4 > C4::polDec()const
{
   R r=absVal(true);
   C4 pol=*this;
   if (r!=0){
     R rInv=1./r; pol*=rInv;
     return pair<R,C4 >(r,pol);
   }
   else{
      return pair<R,C4 >(0.,C4(C(1),C(),C(),C()));
   }
}

C2 CpmLinAlg::operator*(const CC2& a, const C2& x)
```

```
// avoid trc for utmost efficiency
{
   return C2(a.x1.x1 * x.x1 + a.x1.x2 * x.x2,
             a.x2.x1 * x.x1 + a.x2.x2 * x.x2);
}

CC2 CC2::operator !()const
{
   C detInv=det().inv();
   CC2 res;
   res.x1.x1=x2.x2;
   res.x2.x2=x1.x1;
   res.x1.x2=-x1.x2;
   res.x2.x1=-x2.x1;
   return res;
}

CC2 CpmLinAlg::operator*(const CC2& a, const CC2& b)
   {
      CC2 res;
      res.x1.x1=a.x1.x1 * b.x1.x1 + a.x1.x2 * b.x2.x1;
      res.x1.x2=a.x1.x1 * b.x1.x2 + a.x1.x2 * b.x2.x2;
      res.x2.x1=a.x2.x1 * b.x1.x1 + a.x2.x2 * b.x2.x1;
      res.x2.x2=a.x2.x1 * b.x1.x2 + a.x2.x2 * b.x2.x2;
      return res;
   }

CC2 CpmLinAlg::operator+(const CC2& a, const CC2& b)
   {
      CC2 res=a;
      res.x1+=b.x1;
      res.x2+=b.x2;
      return res;
   }

CC2 CpmLinAlg::operator~(const CC2& a)
// transpose
   {
      CC2 res=a;
      res.x1.x2=a.x2.x1;
      res.x2.x1=a.x1.x2;
      return res;
   }

CC2 CC2::trn()const
// transpose
{
   return CC2(x1.x1,x2.x1,x1.x2,x2.x2);
}
```

```
CC2 CC2::her()const
// transpose
{
   return CC2(x1.x1.con(),x2.x1.con(),x1.x2.con(),x2.x2.con());
}

CC2 CC2::zero;

CC2 CC2::unit=CC2(C(1),C(0),C(0),C(1));
CC2 CC2::sig0=CC2::unit;
CC2 CC2::sig1=CC2(C(0),C(1),C(1),C(0));
CC2 CC2::sig2=CC2(C(0),C::Ibar,C::I,C(0));
CC2 CC2::sig3=CC2(C(1),C(0),C(0),C(-1));

bool CC2::prnOn(ostream& out)const
{
     if (!x1.prnOn(out)) return false;
     if (!x2.prnOn(out)) return false;
     return true;
}

bool CC2::scanFrom(istream& in)
{
   if (!x1.scanFrom(in)) return false;
   if (!x2.scanFrom(in)) return false;
   return true;
}

C3 CpmLinAlg::operator*(const CC3& a, const C3& x)
   {
      C3 res;
      res.x1=a.x1.trc(x);
      res.x2=a.x2.trc(x);
      res.x3=a.x3.trc(x);
      return res;
   }

CC3 CpmLinAlg::operator*(const CC3& a, const CC3& b)
   {
      CC3 res;
      res.x1.x1=a.x1.x1*b.x1.x1 + a.x1.x2*b.x2.x1 + a.x1.x3*b.x3.x1;
      res.x1.x2=a.x1.x1*b.x1.x2 + a.x1.x2*b.x2.x2 + a.x1.x3*b.x3.x2;
      res.x1.x3=a.x1.x1*b.x1.x3 + a.x1.x2*b.x2.x3 + a.x1.x3*b.x3.x3;

      res.x2.x1=a.x2.x1*b.x1.x1 + a.x2.x2*b.x2.x1 + a.x2.x3*b.x3.x1;
      res.x2.x2=a.x2.x1*b.x1.x2 + a.x2.x2*b.x2.x2 + a.x2.x3*b.x3.x2;
      res.x2.x3=a.x2.x1*b.x1.x3 + a.x2.x2*b.x2.x3 + a.x2.x3*b.x3.x3;

      res.x3.x1=a.x3.x1*b.x1.x1 + a.x3.x2*b.x2.x1 + a.x3.x3*b.x3.x1;
      res.x3.x2=a.x3.x1*b.x1.x2 + a.x3.x2*b.x2.x2 + a.x3.x3*b.x3.x2;
```

```
      res.x3.x3=a.x3.x1*b.x1.x3 + a.x3.x2*b.x2.x3 + a.x3.x3*b.x3.x3;

      return res;
   }

CC3 CpmLinAlg::operator+(const CC3& a, const CC3& b)
   {
      CC3 res=a;
      res.x1+=b.x1;
      res.x2+=b.x2;
      res.x3+=b.x3;
      return res;
   }

CC3 CpmLinAlg::operator~(const CC3& a)
   {
      CC3 res(a);

      res.x1.x2=a.x2.x1;
      res.x1.x3=a.x3.x1;

      res.x2.x3=a.x3.x2;
      res.x2.x1=a.x1.x2;

      res.x3.x1=a.x1.x3;
      res.x3.x2=a.x2.x3;

      return res;
   }

CC3 CC3::zero;

CC3 CC3::unit=CC3(C3(C(1),C(),C()),C3(C(),C(1),C()),C3(C(),C(),C(1)));

bool CC3::prnOn(ostream& out)const
{
     if (!x1.prnOn(out)) return false;
     if (!x2.prnOn(out)) return false;
     if (!x3.prnOn(out)) return false;
     return true;
}

bool CC3::scanFrom(istream& in)
{
   if (!x1.scanFrom(in)) return false;
   if (!x2.scanFrom(in)) return false;
   if (!x3.scanFrom(in)) return false;
   return true;
}
```

```
C4 CpmLinAlg::operator*(const CC4& a, const C4& x)
   {
      C4 res;
      res.x1=a.x1.trc(x);
      res.x2=a.x2.trc(x);
      res.x3=a.x3.trc(x);
      res.x4=a.x4.trc(x);
      return res;
   }

CC4 CpmLinAlg::operator*(const CC4& a, const CC4& b)
   {
      CC4 res;
      res.x1.x1=a.x1.x1*b.x1.x1 + a.x1.x2*b.x2.x1 + a.x1.x3*b.x3.x1 +\
       a.x1.x4*b.x4.x1;
      res.x1.x2=a.x1.x1*b.x1.x2 + a.x1.x2*b.x2.x2 + a.x1.x3*b.x3.x2 +\
       a.x1.x4*b.x4.x2;
      res.x1.x3=a.x1.x1*b.x1.x3 + a.x1.x2*b.x2.x3 + a.x1.x3*b.x3.x3 +\
       a.x1.x4*b.x4.x3;
      res.x1.x4=a.x1.x1*b.x1.x4 + a.x1.x2*b.x2.x4 + a.x1.x3*b.x3.x4 +\
       a.x1.x4*b.x4.x4;

      res.x2.x1=a.x2.x1*b.x1.x1 + a.x2.x2*b.x2.x1 + a.x2.x3*b.x3.x1 +\
       a.x2.x4*b.x4.x1;
      res.x2.x2=a.x2.x1*b.x1.x2 + a.x2.x2*b.x2.x2 + a.x2.x3*b.x3.x2 +\
       a.x2.x4*b.x4.x2;
      res.x2.x3=a.x2.x1*b.x1.x3 + a.x2.x2*b.x2.x3 + a.x2.x3*b.x3.x3 +\
       a.x2.x4*b.x4.x3;
      res.x2.x4=a.x2.x1*b.x1.x4 + a.x2.x2*b.x2.x4 + a.x2.x3*b.x3.x4 +\
       a.x2.x4*b.x4.x4;

      res.x3.x1=a.x3.x1*b.x1.x1 + a.x3.x2*b.x2.x1 + a.x3.x3*b.x3.x1 +\
       a.x3.x4*b.x4.x1;
      res.x3.x2=a.x3.x1*b.x1.x2 + a.x3.x2*b.x2.x2 + a.x3.x3*b.x3.x2 +\
       a.x3.x4*b.x4.x2;
      res.x3.x3=a.x3.x1*b.x1.x3 + a.x3.x2*b.x2.x3 + a.x3.x3*b.x3.x3 +\
       a.x3.x4*b.x4.x3;
      res.x3.x4=a.x3.x1*b.x1.x4 + a.x3.x2*b.x2.x4 + a.x3.x3*b.x3.x4 +\
       a.x3.x4*b.x4.x4;

      res.x4.x1=a.x4.x1*b.x1.x1 + a.x4.x2*b.x2.x1 + a.x4.x3*b.x3.x1 +\
       a.x4.x4*b.x4.x1;
      res.x4.x2=a.x4.x1*b.x1.x2 + a.x4.x2*b.x2.x2 + a.x4.x3*b.x3.x2 +\
       a.x4.x4*b.x4.x2;
      res.x4.x3=a.x4.x1*b.x1.x3 + a.x4.x2*b.x2.x3 + a.x4.x3*b.x3.x3 +\
       a.x4.x4*b.x4.x3;
      res.x4.x4=a.x4.x1*b.x1.x4 + a.x4.x2*b.x2.x4 + a.x4.x3*b.x3.x4 +\
       a.x4.x4*b.x4.x4;
      return res;
   }
```

```
CC4 CpmLinAlg::operator+(const CC4& a, const CC4& b)
   {
      CC4 res=a;
      res.x1+=b.x1;
      res.x2+=b.x2;
      res.x3+=b.x3;
      res.x4+=b.x4;
      return res;
   }

CC4 CpmLinAlg::operator~(const CC4& a)
   {
      CC4 res(a);

      res.x1.x2=a.x2.x1;
      res.x1.x3=a.x3.x1;
      res.x1.x4=a.x4.x1;

      res.x2.x1=a.x1.x2;
      res.x2.x3=a.x3.x2;
      res.x2.x4=a.x4.x2;

      res.x3.x1=a.x1.x3;
      res.x3.x2=a.x2.x3;
      res.x3.x4=a.x4.x3;

      res.x4.x1=a.x1.x4;
      res.x4.x2=a.x2.x4;
      res.x4.x3=a.x3.x4;

      return res;
   }

CC4 CC4::zero;

CC4 CC4::unit=
CC4(C4(C(1),C(),C(),C()), C4(C(),C(1),C(),C()),
    C4(C(),C(),C(1),C()), C4(C(),C(),C(),C(1)));

bool CC4::prnOn(ostream& out)const
{
     if (!x1.prnOn(out)) return false;
     if (!x2.prnOn(out)) return false;
     if (!x3.prnOn(out)) return false;
     if (!x4.prnOn(out)) return false;
     return true;
}

bool CC4::scanFrom(istream& in)
```

```
{
   if (!x1.scanFrom(in)) return false;
   if (!x2.scanFrom(in)) return false;
   if (!x3.scanFrom(in)) return false;
   if (!x4.scanFrom(in)) return false;
   return true;
}

CC2 CCCC2::operator * (CC2 const& a)const
{
   C r11=x1.x1.x1.x1*a.x1.x1 + x1.x1.x1.x2*a.x1.x2
      + x1.x2.x1.x1*a.x2.x1 + x1.x2.x1.x2*a.x2.x2;
   C r12=x1.x1.x2.x1*a.x1.x1 + x1.x1.x2.x2*a.x1.x2
      + x1.x2.x2.x1*a.x2.x1 + x1.x2.x2.x2*a.x2.x2;
   C r21=x2.x1.x1.x1*a.x1.x1 + x2.x1.x1.x2*a.x1.x2
      + x2.x2.x1.x1*a.x2.x1 + x2.x2.x1.x2*a.x2.x2;
   C r22=x2.x1.x2.x1*a.x1.x1 + x2.x1.x2.x2*a.x1.x2
      + x2.x2.x2.x1*a.x2.x1 + x2.x2.x2.x2*a.x2.x2;
   return CC2(r11,r12,r21,r22);
}

C CCCC2::app(CC2 const& a, bool b1, bool b2)const
{
   if (!b1 && !b2)
      return x1.x1.x1.x1*a.x1.x1 + x1.x1.x1.x2*a.x1.x2
      + x1.x2.x1.x1*a.x2.x1 + x1.x2.x1.x2*a.x2.x2;
   if (!b1 && b2)
      return x1.x1.x2.x1*a.x1.x1 + x1.x1.x2.x2*a.x1.x2
      + x1.x2.x2.x1*a.x2.x1 + x1.x2.x2.x2*a.x2.x2;
   if ( b1 && !b2)
      return x2.x1.x1.x1*a.x1.x1 + x2.x1.x1.x2*a.x1.x2
      + x2.x2.x1.x1*a.x2.x1 + x2.x2.x1.x2*a.x2.x2;
   if (b1 && b2)
      return x2.x1.x2.x1*a.x1.x1 + x2.x1.x2.x2*a.x1.x2
      + x2.x2.x2.x1*a.x2.x1 + x2.x2.x2.x2*a.x2.x2;
   return C(); // never active
}
```

33 cpmcompar.h

```
//? cpmcompar.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_COMPAR_H_
#define CPM_COMPAR_H_
/*
   Description: see cpmcomparx.h
*/
#include <cpmmacros.h>
#include <cpmrmatrix.h>
#include <cpmdimdef.h>
#include <cpmpardef.h>
#include <cpmdim.h>
#include <cpmcamera.h>
#include <cpmforces.h>

#include <cpmshapes.h>
#include <cpmdistribution.h>

#if defined(CPM_DIMDEF2)

#define CPM_DIM CpmDim2
#define CPM_CAMERA CpmCamera2
#define CPM_PHYSICS CpmPhysics2
#define CPM_RIGID_BODY CpmRigidBody2

namespace CPM_RIGID_BODY {
   #include <cpmcomparx.h>
}

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS
#undef CPM_RIGID_BODY

#endif

#if defined(CPM_DIMDEF3)

#define CPM_DIM CpmDim3
#define CPM_CAMERA CpmCamera3
#define CPM_PHYSICS CpmPhysics3
#define CPM_RIGID_BODY CpmRigidBody3

namespace CPM_RIGID_BODY {
   #include <cpmcomparx.h>
}
```

```
#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS
#undef CPM_RIGID_BODY

#endif

#endif
```

34 cpmcompar.cpp

```
//? cpmcompar.cpp
//? Status of work 2008-10-25.
//?
/*
Description: See cpmcomparx.h
*/
#include <cpmbody.h>
#include <cpmconstphys.h>
#include <cpmalgorithms.h>
#include <cpmrfunc.h>
#include <cpmhistogr.h>

#if defined(CPM_DIMDEF2)

#define CPM_DIM CpmDim2
#define CPM_CAMERA CpmCamera2
#define CPM_PHYSICS CpmPhysics2
#define CPM_RIGID_BODY CpmRigidBody2

namespace CPM_RIGID_BODY{
   #include <cpmcomparcpp.h>
}

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS
#undef CPM_RIGID_BODY

#endif

//////////////////////////////////////////////////////////////////////

#if defined(CPM_DIMDEF3)

#define CPM_DIM CpmDim3
#define CPM_CAMERA CpmCamera3
#define CPM_PHYSICS CpmPhysics3
#define CPM_RIGID_BODY CpmRigidBody3

namespace CPM_RIGID_BODY{
   #include <cpmcomparcpp.h>
}

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS
#undef CPM_RIGID_BODY
```

```
#endif
```

35 cpmcomparcpp.h

```
//? cpmcomparcpp.h
//? Status of work 2008-10-25.
//?
using CPM_CAMERA::Camera;

using CPM_DIM::Spc;
using CPM_DIM::Vec;
using CPM_DIM::AxVec;
using CPM_DIM::Group;
using CPM_DIM::RefSys;
using CPM_DIM::dimVec;
using CPM_DIM::SubsetOfSpace;
using CPM_DIM::Sphere;
using CPM_DIM::WireFrame;

using CPM_PHYSICS::Force;
using CPM_PHYSICS::ForTor;
using CPM_PHYSICS::VectorField;
using CPM_PHYSICS::ExtSys;
using CPM_PHYSICS::VecE;
using CPM_PHYSICS::VecBH;
using CPM_PHYSICS::VecG;

using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmLinAlg;
using namespace CpmGraphics;
using namespace CpmAlgorithms;
using namespace CpmArrays;
using namespace CpmFunctions;
using namespace CpmImaging;
using namespace CpmProperties;
using namespace CpmPhysics;
using CpmImaging::Histogram;
   // this is the dimension-independent part (mainly physical constants,
   // so far)

//////////////////// variables of the namespace ////////////////////

bool accExp=false;
// false is OK
// accurate exponential function (from the Lie algebra of the rotation
// group to this group). If this is false a fast and generally
// sufficient approximation is used.

bool markComParMec=false;
   // if this is true, the function ComPar::mark(Camera...) shows
```

```
   // the axes of inertia and the vectors of angular velocity and
   // angular momentum

using  CpmProperties::SysControl;
   // new rigid body integrator is defined there, true is OK

bool balance=false;

namespace{
const R facEl=sqrt(2.)/3.;
   // factor appearing in Hertz' formula

} // anon namespace

///////////////// class RigidObject ////////////////////////////////////

void RigidObject::make(R r, const Spc& x, const Vec& v,
         const AxVec& ori, const AxVec& omega)
{
   if (r<0){
      r=0;
      cpmwarning(
         "RigidObject::make(R r,...) r was negative, set to 0");
   }
   r_=r;
   x_=x;
   v_=v;
   ori_=ori;
   omega_=omega;
   RigidObject::update();
}

void RigidObject::transformStreamableData(const Group& g)
{
   x_*=g;
   AxVec rt=g.rot();
   v_*=rt;
   ori_*=rt;
   omega_=omega_.adjointAction(rt);
      // 'omega_=' was missing till 2005-02-14
}

void RigidObject::freeStep(R dt)
{
   if (dt==0.) return;
   t_+=dt;
   if (locked_) return;
// adjusting position from linear velocity
   Vec sh=v_*dt;
   x_+=sh;
```

```
// adjusting angular position from constant angular velocity
   AxVec rt=omega_.exp(dt);
   ori_*=rt;
   RigidObject::update();
}

void RigidObject::adjustOmega(const Spc& xp, const Vec& vp)
// omega = (xRel x vRel)/(xRel|xRel) , 'x'= cross product
{
   Vec vRel=vp-v_;
   Vec xRel=xp-x_;
   omega_=xRel.cross(vRel)*CpmRoot::inv(xRel|xRel);
}

bool RigidObject::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("RigidObject",str);
   cpmp(r_);
   cpmp(x_);
   cpmp(v_);
   cpmp(ori_);
   cpmp(omega_);
   cpmp(t_);
   return true;
}

bool RigidObject::scanFrom(istream& str)
{
   cpms(r_);
   cpms(x_);
   cpms(v_);
   cpms(ori_);
   cpms(omega_);
   cpms(t_);
   RigidObject::update(); // we did print only the independent
      // quantities
      // notice that update is virtual and we have to prevent that the
      // versions of derivates get called here
      // scope resolution was missing till 2000-12-21
   return true;
}

R RigidObject::disVal(const RigidObject& ro)const
{
   using CpmRootX::sup;
   using CpmRootX::inf;
   R dTrans=(x_-ro.x_).absVal();
   R phi=(ori_*(-ro.ori_)).phi();
   R dRot=sup(r_,ro.r_)*phi;
      // we transform a rotation into a peripheral distance in order to
```

```
      // make spatial distances and rotational ones directly comparable.
      // In actual applications we always compare the same cluster at
      // different points in time, so the two radii r and ro.r are
      // typically the same so that it makes no difference whether we
      // combine those by sup or inf or mean
   R dPos=dTrans+dRot; // adding is 'more continuous' than sup()
   dPos=dPos/(2.*inf(r_,ro.r_));   // divided by diameter is more natural
   // than by radius
   return dPos;
}

//////////////////////// struct PhyParPrp ////////////////////////////

bool PhyParPrp::prnOn(ostream& str)const
{
   if (!CpmRoot::writeTitle("PhyParPrp",str) ) return false;
   cpmp(m_);
   cpmp(r_);
   cpmp(q_);
   cpmp(E_);
   cpmp(U0_);
   cpmp(r0_);
   cpmp(a_);
   cpmp(scnr_);
   cpmp(forceType_);
   return true;
}

bool PhyParPrp::scanFrom(istream& str)
{
   cpms(m_);
   cpms(r_);
   cpms(q_);
   cpms(E_);
   cpms(U0_);
   cpms(r0_);
   cpms(a_);
   cpms(scnr_);
   cpms(forceType_);
   return true;
}

//////////////////// struct PointParC ////////////////////////////////

bool PointParC::prnOn(ostream& str)const
{
   if (!CpmRoot::writeTitle("PointParC",str) ) return false;
   cpmp(p_);
   cpmp(id_);
   cpmp(idValid_);
```

```
   cpmp(color_);
   return true;
}

bool PointParC::scanFrom(istream& str)
{
   cpms(p_);
   cpms(id_);
   cpms(idValid_);
   cpms(color_);
   return true;
}

//////////////////// struct PointParV ////////////////////////////////

bool PointParV::prnOn(ostream& str)const
{
   if (!CpmRoot::writeTitle("PointParV",str) ) return false;
   cpmp(owner_);
   cpmp(x_);
   cpmp(v_);
   cpmp(vDot_);
   cpmp(t_);
   return true;
}

bool PointParV::scanFrom(istream& str)
{
   cpms(owner_);
   cpms(x_);
   cpms(v_);
   cpms(vDot_);
   cpms(t_);
   return true;
}

//////////////////// class PointPar ///////////////////////////////////

PointPar::PointPar(RecordHandler const& rch, Word const& sec, Z j)
{
   Word loc("PointPar(RecordHandler,Word,Z)");
   PhyParPrp ppp;
   // PhyParPrp being a state of the art struct, all its components are
   // initialized
   ContactProperties cp(rch,sec);
   ppp.set_cp(cp);
   Z ped=2; // ped = 'pedantic'
   R r;
   rch.read(sec,"r",r,ped);
      // we need r
```

```
   ped=1;
   R dr=0;
   rch.read(sec,"dr",dr,ped); // we don't need dr
   Iv ivr(r-dr,r+dr); // works also for dr<0
   r=ivr.ran(j);
   cpmassert(r>0,loc);
   ppp.r_=r;
// getting the mass
   R m;
   ped=1;
   bool b=rch.read(sec,"m",m,ped);
   if (b && m>0) ppp.m_=m;
   else{
      ped=2;
      // if we don't get rho, we are lost, since m<=0 is not OK
      R rho;
      b=rch.read(sec,"rho",rho,ped);
      R vol=CpmDim3::Spc::vol(r);
      m=vol*rho;
      ppp.m_=m;
   }
// getting the charge. Unlike for m, the value 0 is OK
// so there is no problem if all input fails
   R q=0;
   ped=1;
   b=rch.read(sec,"q",q,ped);
   if (b && q!=0) ppp.q_=q;
   else{
      R sigma=0;
      b=rch.read(sec,"sigma",sigma,ped);
      R sur=CpmDim3::Spc::sur(r);
      q=sur*sigma;
      ppp.q_=q;
   }
   ped=1;
   // read all relevant data that are there;
   // this then overrides the initialization.
   Z forceType=4; // if forceType is not there, the behavior
      // should be the same as for ComPar
   rch.read(sec,"forceType",forceType,ped);
   ppp.forceType_=forceType;
   if (forceType!=4){
      R U0=0,r0=0,a=0;
      b=rch.read(sec,"U0",U0,ped);
      if (b) ppp.U0_=U0;
      b=rch.read(sec,"r0",r0,ped);
      if (b) ppp.r0_=r0;
      b=rch.read(sec,"a",a,ped);
      if (b) ppp.a_=a;
   }
```

```
   *this=PointPar(ppp);
   Z color;
   b=rch.read(sec,"color",color,ped);
   if (b) cc.color_=color;
 //  cpmdebug(ppp);
}

AxVec PointPar::angMom(Spc const& p)const
{
   Vec xp=get_x()-p;
   AxVec res=xp.cross(mom());
   return res;
}

bool PointPar::prnOn(ostream& str)const
{
   if (!CpmRoot::writeTitle("PointPar",str) ) return false;
   cpmp(cc);
   cpmp(cv);
   return true;
}

bool PointPar::scanFrom(istream& str)
{
   cpms(cc);
   cpms(cv);
   return true;
}

void PointPar::set_id(Z id1)
{
   if (!cc.idValid_){ // a valid id gets never changed
      cc.id_=id1;
      cc.idValid_=B(true);
   }
}

B PointPar::isVal(const V<PointParV>& vv)
   // returns true if there are no reasons to
   // reject data vv
   // Present implementation based on comparing velocities with
   // SysControl::vCrit
{
   if (SysControl::vCrit<=0) return B(true);
   Z mL=2;
   const R lazyFactor=1.5;
      // since the velocity never exceeds 2*SysControl::vCrit
      // setting this factor >2 prevents any influence from it.
   static Word loc("PointPar::isVal(const V<PointParV>& vv)");
   CPM_MA
```

```
   R vStop=lazyFactor*SysControl::vCrit;
   Z i,j,d=Spc::dimension(),n=vv.dim();
   R_Matrix vel(d,n);
   Z count=0;
   for (j=1;j<=n;j++){
      Vec vj=vv[j].v_;
      if (vj.absVal()>vStop) count++;
      for (i=1;i<=d;i++){
         vel[i][j]=cpmabs(vj[i]);
      }
   }
   V<R> vMax(d);
   for (i=1;i<=d;i++){
      vMax[i]=vel[i].sup();
   }
   ostringstream ost;
   ost<<endl<<"Output of PointPar::isVal(const V<PointParV>& vv)";
   ost<<endl<<"concerning maximum velocity in i-direction, i=1,2,(3)";

   for (i=1;i<=d;i++){
      ost<<endl<<"vMax["<<i<<"]="<<vMax[i];
   }
   B res(true);
   if (count>0) res=B(false);
   if (!res){
      cpmwarning(loc&" "&cpm(count)&" particles move unexpectedly fast");
      cpmmessage("Details: "&Word(ost.str()));
   }
   CPM_MZ
   return res;
}

B PointPar::isVal(const V<PointPar>& vc)
{
   Z vd=vc.dim();
   V<PointParV> vv(vd);
   for (Z i=1;i<=vd;i++) vv[i]=vc[i].cv;
   return PointPar::isVal(vv);
}

namespace{
   const R redFac=0.25;
   // If a particle velocity is found to be larger than
   // vCrit, this is considered an exceptonal situation and
   // the velocity gets reduced (by multiplication by a
   // suitable real factor) to a value redFac*vCrit;
}

void PointPar::reg(void)
{
```

```
   if (SysControl::vCrit<=0) return;
   R v1_a=cv.v_.absVal();
   if (v1_a>SysControl::vCrit){
      R red1=redFac*SysControl::vCrit/v1_a;
      cv.v_*=red1;
   }
}

void PointPar::operator *=(R fac)
{
   cpmassert(fac>0,"PointPar::operator *=(R fac)");
   R fac3=fac*fac*fac;
   cc.p_.r_*=fac;
   cc.p_.r0_*=fac;
   cc.p_.a_*=fac;
   cc.p_.m_*=fac3;
}

void PointPar::mir_(Z i) // mirror (reflection), clear from code.
{
   if (i>=1 && i<=3){
      cv.mir_(i);
   }
}

void PointPar::intAct(R dt, const Force& fs, const FieldSpace& fval)
{
   static Word loc("PointPar::intAct(...)");
   fval; // to avoid warning
   R m=cc.p_.m_;
   cpmassert(m>0,loc);
   R mInv=1./m;
   Vec acc=fs.f*mInv;
   R acca=acc.absVal();
   if (SysControl::aCrit>0 && acca>SysControl::aCrit)
      acc*=(SysControl::aCrit/acca);
   Vec dv=acc*dt;
   if (SysControl::accMem){
      R dth=0.5*dt;
      dv-=cv.vDot_*dth;
      cv.vDot_=acc;
   }
   cv.v_+=dv;
}

void PointPar::mark(Graph& g, const Color& color)const
{
   Spc x=get_x();
   R r=get_r();
   R2 center(x[1],x[2]);
```

```
   g.fillCircle(center,r,color);
}

void PointPar::mark(const Camera& sc, const Color& col)const
{
   if (col.transparent()) return;

   R r=get_r();
   Spc x=get_x();
   if (!SysControl::coloredParticles){
      if (SysControl::howToMarkParticles==1){
         WireFrame globe=WireFrame::globe(r,0,3,2,12);
         Vec shift=x-Spc();
         globe*=Group(shift);
         sc.mark(globe,col);
      }
      else{
         RefSys rs;
         V<Vec> tp=rs.getVectors();
         for (Z k=1;k<=tp.dim();k++) tp[k]*=r;
         sc.mark(x-tp[1],x+tp[1],col);
         sc.mark(x-tp[2],x+tp[2],col);
         if (tp.dim()==3) sc.mark(x-tp[3],x+tp[3],col);
      }
   }
   else{
      sc.mark(x,r,col);
   }
   if (SysControl::dtMult>0){
      Vec v=get_v();
      R dt=SysControl::tStep*SysControl::dtMult;
      // time used for visualizing the velocity as a distance
      sc.mark(x,x+v*dt,col);
   }
}

Spc PointPar::frcCen(const PointPar& pa)const
{
   X2<R,Rdim> prv=(pa.cv.x_-cv.x_).polDec();
   Vec n(prv.second());
      // vector from x to pa.x
   R s=(cc.p_.r_-pa.cc.p_.r_)*0.5;
   return cv.x_.cen(pa.cv.x_)+n*s;
}

Force PointPar::forceFrictionAdhesionElastic( R h, const Vec& n12,
         R r1, R r2, const Vec& v1, const Vec& v2,
         const ContactProperties& cp1, const ContactProperties& cp2 )
{
   if (h>SysControl::intactRange) return Force();
```

```
   R rHarm=harmonicMean(r1,r2);
   if (h>=rHarm) return Force();
   ContactProperties cp=cp1.combine(cp2);
   R eRatio=cp.scnr_;
   R friction=cp.mu_;
      // friction coefficient
   R v0=cp.v0_;
      // tangential reaction force is nearly 0. Values <=0
      // have to be avoided; class SysControl dos not guarantee this
   R v0Min=1e-6;
   if (v0<v0Min) v0=v0Min;
   R fAdhesive=0;
   if (!SysControl::noAdhesion){
      // here fAdhesive is implemented trivially. A proper
      // implementation is essential for the electrophotographic
      // application which was among the origins of the present program.
   }
// elastic repulsion of elastic spheres at loaded contact (formulas of
// Heinrich Hertz) (only contributing if overlapping)
   R fElastic=0;
   Vec fTan;
   if (!SysControl::noElasticRepulsion && h<0.){
      R Eeff=cp.E_;
      fElastic=-facEl*Eeff*sqrt(rHarm)*pow(-h,R(1.5));
         // should repell 1 from 2, thus minus sign.
      if (!SysControl::noFriction){
         Vec v12=v2-v1;
         R vn=(v12|n12); // scalar product
         if (vn>0) fElastic*=eRatio;
         // inelasticity is described by letting the force
         // in release phase be by a factor smaller than in the
         // compression phase. Since energy is force times distance,
         // this should imply for normal collisions that
         // kinetic energy after the collision =
         // kinetic energy before the collision * eRatio
         Vec vTan=v12-n12*vn;
         R avt=vTan.normalize_(1);
         R red=-avt/(v0+avt); // 0 for avt==0 and -1 for avt >> v0
         fTan=vTan*(friction*fElastic*red);
      }
   }
   return Force(n12*(fAdhesive+fElastic)+fTan);
}

R PointPar::elasticPotential( R h, const Vec& n12, R r1, R r2,
   ContactProperties const& cp1, ContactProperties const& cp2 )
{
   R res=0;
   n12; // not used here
   if (h>SysControl::intactRange) return res;
```

```
   R rHarm=harmonicMean(r1,r2);
   if (h>=rHarm) return res;
   ContactProperties cp=cp1.combine(cp2);
   R eRatio=cp.scnr_;
   if (!SysControl::noElasticRepulsion && h<0.){
      R Eeff=cp.E_;
      res+=0.4*facEl*Eeff*sqrt(rHarm)*pow(-h,R(2.5));
   }
   return res;
}

R PointPar::collisionTime(R r1, R r2, R m1, R m2, R v,
      const ContactProperties& cp1, const ContactProperties& cp2)
{
   Word loc("PointPar::collisionTime(...)");
   R Eeff=harmonicMean(cp1.E_,cp2.E_);
      // =3/(2D) with LL's D
   R k=(8./15.)*Eeff*::sqrt(0.5*harmonicMean(r1,r2));
         // 0.5*harmonicMean(r1,r2)=r1*r2/(r1+r2)
         // (8/15) comes from (4/5)*(2/3) see LL p.39
   R mu=0.5*harmonicMean(m1,m2);
   cpmassert(k>0,loc);
   R res=(mu/k);
   res*=res;
   cpmassert(v>0,loc);
   res/=v;
   return (30*CpmRoot::Pi/16)*pow(res,R(0.2));
}

Force PointPar::forceContact(const PointPar& p,
   const Vec& vThis, const Vec& vp)const
{
   X2<R,Vec> prv=gapAndDirection(p);
   R h=prv.first();
   Vec n=prv.second();
   ContactProperties cp1=get_cp();
   ContactProperties cp2=p.get_cp();
   R r1=cc.p_.r_;
   R r2=p.cc.p_.r_;
   return forceFrictionAdhesionElastic(h,n,r1,r2,vThis,vp,cp1,cp2);
}

Force PointPar::forces( const PointPar& pp )const
{
   Force f;
   if (cc.id_==pp.cc.id_) return f;
    // i.e. particle does not directly interact with itself
   Vec xRel=pp.cv.x_-cv.x_;
   R dRel=xRel.normalize();
      // now xRel is the unit vector pointing from *this to pp
```

```
if (dRel > SysControl::intactRange) return f;

PointCharge pc,pc_pp;
if(!SysControl::noInteractionChargeToCharge ||
   !SysControl::noMagneticInteractionChargeToCharge){
   pc=getPointCharge();
   pc_pp=pp.getPointCharge();
}
if(!SysControl::noInteractionChargeToCharge){
   // Coulomb force
   f.addForce(pc.force(pc_pp));
}
if(!SysControl::noMagneticInteractionChargeToCharge){
   // this control structure implies that one may enable the magnetic
   // interaction while disabling the electrostatic one. This is
   // expected to be useful since for non-relativistic velocities
   // the magnetic interaction would else be dominated by the
   // electrostatic one
   f.addForce(pc.forceMag(cv.v_,pc_pp,pp.get_v()));
      // this contains the Lorentz force from the magnetic
      // field which pp generates (if it moves)
}
if(!SysControl::noInteractionMassToMass){
   // Newton force
   PointMass pm=getPointMass();
   PointMass pm_cl=pp.getPointMass();
   f.addForce(pm.force(pm_cl));
}
if(!SysControl::noElasticRepulsion){
   // 'elastic force'
   Z fT1=cc.p_.forceType_;
   Z fT2=pp.cc.p_.forceType_;
   R fM=0; // not matching force types should produce
      // no interaction
   if (fT1==fT2){
      if (fT1==0){
         // no force, fM=0 on initialization
         ;
      }
      else if (fT1==4){
         // Hertz formula for elastic repulsion of spheres
         R r1=get_r();
         R r2=pp.get_r();
         R overlap=r1+r2-dRel;
         if (overlap<=0);
         else{
            R E=harmonicMean(cc.p_.E_,pp.cc.p_.E_);
            R r=harmonicMean(r1,r2);
            fM=-facEl*E*sqrt(r)*::pow(overlap,R(1.5));
         }
```

```
         }
         else{ // chemical forces
            R U0=harmonicMean(cc.p_.U0_,pp.cc.p_.U0_);
            R r0=0.5*(cc.p_.r0_+pp.cc.p_.r0_);
            R a=0.5*(cc.p_.a_+pp.cc.p_.a_);
            if (fT1==1) fM=R_Func::dPotLJ(U0,r0)(dRel);
            else if (fT1==2) fM=R_Func::dPotMorse(U0,r0,a)(dRel);
            else if (fT1==3) fM=R_Func::dPotUM(U0,r0,a)(dRel);
            else cpmwarning
            ("Invalid force type in PointPar::forces(PointPar)");
         }
      }
      if (fM!=0){
         Vec vRel=pp.cv.v_-cv.v_;
         R scnr=harmonicMean(cc.p_.scnr_,pp.cc.p_.scnr_);
         R dirFac=(vRel|xRel);
         if (dirFac>0){ // action only for particles that move
         // as to increasing their distance.
            if (fM>0){ // attraction
               if (scnr>0) fM/=scnr; // increase attraction (0< res<=1)
            }
            else{ // repulsion
               fM*=scnr; // reduce repulsion
            }
         }
         Vec vf=xRel*fM;
            // fM>0 attraction, fM<0 repulsion
         f+=vf;
      }
   }
   return f;
}

R PointPar::pot( const PointPar& pp )const
{
   R res=0;
   if (cc.id_==pp.cc.id_) return res;
   Vec xRel=pp.cv.x_-cv.x_;
   R dRel=xRel.normalize();
   if (dRel > SysControl::intactRange) return res;

   if(!SysControl::noInteractionChargeToCharge){
      PointCharge pc=getPointCharge();
      PointCharge pc_pp=pp.getPointCharge();
      res+=pc.pot(pc_pp);
   }

   if(!SysControl::noInteractionMassToMass){
      PointMass pm=getPointMass();
      PointMass pm_cl=pp.getPointMass();
```

```
      res+=pm.pot(pm_cl);
   }
   if(!SysControl::noElasticRepulsion){
      Z fT1=cc.p_.forceType_;
      Z fT2=pp.cc.p_.forceType_;
      if (fT1==fT2){
         if (fT1==0){
            // no force, fM=0 on initialization
            ;
         }
         else if (fT1==4){
            // Hertz formula for elastic repulsion of spheres
            R r1=get_r();
            R r2=pp.get_r();
            R overlap=r1+r2-dRel;
            if (overlap<=0);
            else{
               R E=harmonicMean(cc.p_.E_,pp.cc.p_.E_);
               R r=harmonicMean(r1,r2);
               res+=0.4*facEl*E*sqrt(r)*::pow(overlap,R(2.5));
            }
         }
         else{ // chemical forces
            R U0=harmonicMean(cc.p_.U0_,pp.cc.p_.U0_);
            R r0=0.5*(cc.p_.r0_+pp.cc.p_.r0_);
            R a=0.5*(cc.p_.a_+pp.cc.p_.a_);
            if (fT1==1) res+=R_Func::potLJ(U0,r0)(dRel);
            else if (fT1==2) res+=R_Func::potMorse(U0,r0,a)(dRel);
            else if (fT1==3) res+=R_Func::potUM(U0,r0,a)(dRel);
            else cpmwarning
            ("Invalid force type in PointPar::pot(PointPar)");
         }
      }
   }
   return res;
}

Force PointPar::forces( const FieldSpace& fs)const
{
   Vec  res;
   if (!SysControl::noInteractionMagneticField){
      VecBH fB=fs.c2();
      Vec LorentzForce=fB.cross(cv.v_)*(-cc.p_.q_);
      res+=LorentzForce;
   }
   if (!SysControl::noInteractionElectricField){
      Vec fieldE=fs.c1();
      res+=fieldE*cc.p_.q_;
   }
   if (!SysControl::noInteractionGravitationField){
```

```
      Vec fieldG=fs.c3();
      res+=fieldG*cc.p_.m_;
   }
   return Force(res);
}

R PointPar::pot( const FieldSpace& f)const
{
   R res=0;
   if(!SysControl::noInteractionElectricField){
      res+=getPointCharge().potEnr(f.c1());
   }
   if(!SysControl::noInteractionGravitationField){
      res+=getPointMass().potEnr(f.c3());
   }
   return res;
}

R PointPar::potElastic(PointPar const& p)const
   // potential of the elastic forces contributing to forceContact
{
   if (idn(p)) return 0;
      // a particles doesn't interact with itself
   R h=gap(p);
   R res=0;
   if (!SysControl::noElasticRepulsion && h<0.){
      h=-h; // now we have overlap and h>0
      const R fac=0.4*sqrt(2.)/3; // not negative here
      R Eeff=harmonicMean(cc.p_.E_,p.cc.p_.E_);
      R rHarm=harmonicMean(cc.p_.r_,p.cc.p_.r_);
      res=fac*Eeff*sqrt(rHarm)*pow(h,R(2.5));
      // integral of h^1.5= (h^2.5)/2.5 = (h^2.5)*0.4
   }
   return res;
}

Force PointPar::forces( const VectorField& f)const
{
   FieldSpace fs=f(cv.x_);
   return PointPar::forces(fs);
}

Force PointPar::forces( const RigidObject& ro)const
{
   if (SysControl::eta==0. || SysControl::noFriction) return Force();
   R eta=SysControl::eta;
   using CpmRoot::Pi;
   Vec vRel=cv.v_-ro.get_v();
   R vRelAbs=vRel.absVal();
   R pi_eta_r=Pi*eta*cc.p_.r_;
```

```
   return Force(-vRel*6*pi_eta_r);
}

SourceOfVectorField PointPar::source()const
{
   ElcDipole ed;
   MagDipole md;
   return SourceOfVectorField(getPointCharge(),ed,md,getPointMass());
}

R PointPar::sclVal(const Word& w)const
{
   if (w=="mass") return cc.p_.m_;
   else if (w=="velocity") return cv.v_.absVal();
   else if (w=="kinetic_energy") return kinEnr();
   else if (w=="charge") return cc.p_.q_;
   else cpmerror("PointPar::sclVal("&w&") not defined");
   return 0;
}

void PointPar::showProperty(const Word& theme, const V<PointPar>& vp,
   const Camera& ca, const R_Vector& parameters)
{
   Z mL=3;
   Word loc("PointPar::showProperty(...)");
   CPM_MA
   const Z nBinMax=200;
   const R electricMomentCompareFactor=4;
   const bool histogramCumulative=true;
   Z i,j,nPlanes,nBins,n;
   R_Matrix val;
   Iv iv;
   if (theme=="HistogramVelocity"){
      nPlanes=3;
      n=vp.dim();
      nBins=inf<Z>(n,nBinMax);
      val=R_Matrix(nPlanes,n);
      R maxv=parameters[1];
      iv=Iv(-maxv,maxv);
      for (i=1;i<=nPlanes;i++)
         for (j=1;j<=n;j++)
            val[i][j]=vp[j].get_v()[i];
   }
   else if (theme=="Histogram_xPosition"){
      nPlanes=1;
      n=vp.dim();
      nBins=inf<Z>(n,nBinMax);
      val=R_Matrix(nPlanes,n);
      R maxv=parameters[1];
      iv=Iv(-maxv,maxv);
```

```
      for (i=1;i<=nPlanes;i++)
         for (j=1;j<=n;j++)
            val[i][j]=vp[j].get_x()[1];
   }
   else if (theme=="Histogram_zPosition"){
      if (Spc::dimension()==2) return;
      nPlanes=1;
      n=vp.dim();
      nBins=inf<Z>(n,nBinMax);
      val=R_Matrix(nPlanes,n);
      R maxv=parameters[1];
      iv=Iv(-maxv,maxv);
      for (i=1;i<=nPlanes;i++)
         for (j=1;j<=n;j++)
            val[i][j]=vp[j].get_x()[3];
   }
   Histogram his(nPlanes,nBins,iv[1],iv[2],theme);
   for (i=1;i<=nPlanes;i++)
      for (j=1;j<=n;j++)
         his.put(i,val[i][j]);
   his.show(ca.getGraph(),histogramCumulative);
   CPM_MZ
}

//////////////////// struct ComParC ////////////////////////////////////
// constant data of ComPar

bool ComParC::prnOn(ostream& str)const
{
   cpmwt("ComParC");
   cpmp(r);
   cpmp(volume);
   cpmp(ri);
   cpmp(xr0);
   cpmp(I0);
   cpmp(ej0);
   cpmp(prop);
   cpmp(id);
//   cpmassert(id!=0,"ComParC::prnOn(istream&)");
   cpmp(idValid);
   return true;
}

bool ComParC::scanFrom(istream& str)
{
   cpms(r);
   cpms(volume);
   cpms(ri);
   cpms(xr0);
   cpms(I0);
```

```
   cpms(ej0);
   cpms(prop);
   cpms(id);
//   cpmassert(id!=0,"ComParC::scanFrom(istream&)");
   cpms(idValid);
   return true;
}

//////////////////// struct ComParV ////////////////////////////////////
// variable data of ComPar

bool ComParV::prnOn(ostream& str)const
{
   cpmwt("ComParV");
   cpmp(owner);
   cpmp(x);
   cpmp(v);
   cpmp(vDot);
   cpmp(ori);
   cpmp(omega);
   cpmp(omgDot);
   cpmp(qd);
   cpmp(t);
   return true;
}

bool ComParV::scanFrom(istream& str)
{
   cpms(owner);
   cpms(x);
   cpms(v);
   cpms(vDot);
   cpms(ori);
   cpms(omega);
   cpms(omgDot);
   cpms(qd);
   cpms(t);
   return true;
}

//////////////////// class ComPar ////////////////////////////////////////
//
// Since also in the 'two-dimensional' model, the bodies are 3
// dimensional one needs 3-dimensional computation also in 2D case
typedef CpmDim3::Vec Vec3;
typedef CpmDim3::Spc Spc3;

namespace{ // anonymous namespace for hiding names

RR3 inertia(R m, const Vec3& r)
```

```
// Huette E 11 (7), in 2-dimensional case only res.x3.x3 is important
{
   RR3 res;
   res.x1.x1=m*(r[2]*r[2]+r[3]*r[3]);
   res.x2.x2=m*(r[1]*r[1]+r[3]*r[3]);
   res.x3.x3=m*(r[1]*r[1]+r[2]*r[2]);

   res.x1.x2=-m*r[1]*r[2];
   res.x1.x3=-m*r[1]*r[3];
   res.x2.x3=-m*r[2]*r[3];

   res.x3.x1=res.x1.x3;
   res.x3.x2=res.x2.x3;
   res.x2.x1=res.x1.x2;
   return res;
}

Spc3 f2Create(const Z&i, const Vec3& vL, const Vec3& vU)
{
   using CpmRoot::ranVal;
   Z i0=i;
   Z shift=23;
   Z d=vL.dim();
   R x0=1;
   Vec3 vr;
   for (Z j=1;j<=d;j++){
      R xr=ranVal(x0,i0);
      xr=(xr+1.)*0.5;
      i0+=shift;
      vr[j]=vL[j]+xr*(vU[j]-vL[j]);
   }
   return Spc3()+vr;
}

bool chi0(const Spc3& x, const Spc3& x0, const R& r)
// indicator function of a sphere arround x0 with radius r (true inside
// the sphere)
{
   return (x-x0).absVal()<=r;
}

bool f2chi(const Spc3& x, const V<Spc3>& xList, const V<R>& rList)
{
   Z n=xList.dim();
   for (Z i=1;i<=n;i++){
      bool val=chi0(x,xList[i],rList[i]);
      if (val==true) return true; // fast end, needs not to test all
      // spheres
   }
   return false;
```

```
}

R boolToR(bool x){ return (x==true ? 1 : 0);}

R c0(const Spc3& x){ return 1;}
R c1(const Spc3& x){ return x[1];}
R c2(const Spc3& x){ return x[2];}
R c3(const Spc3& x){ return x[3];}
R c11(const Spc3& x){ return x[2]*x[2]+x[3]*x[3];}
R c12(const Spc3& x){ return -x[1]*x[2];}
R c13(const Spc3& x){ return -x[1]*x[3];}
R c22(const Spc3& x){ return x[1]*x[1]+x[3]*x[3];}
R c23(const Spc3& x){ return -x[2]*x[3];}
R c33(const Spc3& x){ return x[1]*x[1]+x[2]*x[2];}

RR3 Steiner(const RR3& J, const Vec3& x, R m)
//TT Huette E 31 (8). Argument is J^A and result J^S in the notation
// there
{
   RR3 res;
   R x11=x[1]*x[1]*m;
   R x22=x[2]*x[2]*m;
   R x33=x[3]*x[3]*m;
   R x12=x[1]*x[2]*m;
   R x13=x[1]*x[3]*m;
   R x23=x[2]*x[3]*m;
   res[1][1]=J[1][1]-x22-x33;
   res[2][2]=J[2][2]-x11-x33;
   res[3][3]=J[3][3]-x11-x22;
   res[1][2]=J[1][2]+x12;
   res[1][3]=J[1][3]+x13;
   res[2][3]=J[2][3]+x23;
   res[2][1]=res[1][2];
   res[3][1]=res[1][3];
   res[3][2]=res[2][3];

   return res;
}

} // anonymous namespace

void ComPar::make(R rs, const V<Vec>& xr0_, const V<R>& ri_,
   const ParticleProperties& prop_, const Vec& ej0_, const Vec& v_,
   const AxVec& omega_)
// as this is implemented here, the computation is completely determined
// by
// the arguments of the function.
{
// method parameters
   const Z nT=20000;
```

```
      // so many useful data accquired in stochastical integration
   const Z nMax=1000000;
      // max of trials in stochastic integration
   const Palette colorCarrier=BLUE;
   const Palette colorToner=RED;

// Setting some data directly from the arguments of the function
   Z mL=3;
   Word loc("ComPar::make(...)");
   CPM_MA
   cv.v=v_;
   cv.omega=omega_;
   cc.xr0=xr0_;
   cc.ri=ri_;
   cc.ej0=ej0_;
   cc.prop=prop_;

   cv.ori=AxVec();
   cv.x=Spc();

   n=cc.xr0.dim(); // number of particles, is a dependent data element
      // will be set again at the end, does not cause harm
   cpmassert(n==cc.ri.dim(),loc); // consistency check

   if (n==1){ // here is no need for stochastical integration
      cc.r=rs;
      cc.xr0[1]=Vec();
      cc.ri[1]=rs;
      cc.volume=Spc3::vol(rs);
      R m=cc.volume*cc.prop.rho;
      R IxxSphere=0.4*m*rs*rs; // (2/5)*m*r^2
      RR3 matInertia;
      matInertia[1][1]=IxxSphere;
      matInertia[2][2]=IxxSphere;
      matInertia[3][3]=IxxSphere;
      cc.I0=InTens(matInertia);
         // cv.x already correct
   }
   else{

// Threedimensional part: Even for the 2-dimensional model, particles are
// made out of three-dimensional spheres; their motion is restricted to
// a plane, though.

// First we find a rectangular box (German 'Quader') in which the whole
// body is fully contained.
      Vec3 vUpper;
      Vec3 vLower;
      Z d=vUpper.dim(); // d=3
```

```
      vUpper.set(-hugeNumber);
      vLower.set(hugeNumber);
      Vec3 vVal;
      R rVal;
      R vU,vL;
      Z i,k;
      for (i=1;i<=n;i++){
         for (k=1;k<=dimVec;k++) vVal[k]=cc.xr0[i][k];
         rVal=cc.ri[i];
         for (k=1;k<=d;k++){
            vU=vVal[k]+rVal;
            vL=vVal[k]-rVal;
            if ( vU>vUpper[k] ) vUpper[k]=vU;
            if ( vL<vLower[k] ) vLower[k]=vL;
         }
      }
      Vec3 sites=vUpper-vLower;
      R volumeBox=1;

      for (k=1;k<=d;k++) volumeBox*=sites[k];
      cpmmessage(mL,"volumeBox="&cpm(volumeBox));

// What we need now is a sequence of Vecs xr0_ which is uniformely
// distributed in vLower[i]<=xr0_[i]<=vUpper[i]

      F<Z,Spc3> create=F2<Z,Vec3,Vec3,Spc3>(vLower,vUpper)(f2Create);
      V<Spc3> xi(n); // initialized by Spc3()
      Vec3 shift; // only Vec-components will become initialized
      for (i=1;i<=n;i++){
         for (k=1;k<=dimVec;k++){ shift[k]=cc.xr0[i][k];}
         xi[i]+=shift;
      }
      F<Spc3,bool> chi=F2<Spc3,V<Spc3>,V<R>,bool>(xi,cc.ri)(f2chi);
      Z norMet=1;
      RandomIntegration<Spc3> ran(create,chi,norMet);
      // third argument introduced 2006-01-05. Prior to this
      // modification, the effect of this call was as it would
      // be now for norMet=0. This old and wrong version made
      // the quantity xx[1] equal to 1 in the following! It has
      // to come out as the volume of the particle divided by the
      // volume of the enclosing box.
      // was wrong till 2006-01-06
      Z nList=1+d+(d*(d+1))/2;
      // the list contains the volume, center-of-mass position, and the
      // non-redundant components of the tensor of inertia
      V<F<Spc3,R> > f(nList);
      Z iList=1;
      f[iList++]=F<Spc3,R>(c0);
      f[iList++]=F<Spc3,R>(c1);
      f[iList++]=F<Spc3,R>(c2);
```

```
      f[iList++]=F<Spc3,R>(c3);
      f[iList++]=F<Spc3,R>(c11);
      f[iList++]=F<Spc3,R>(c12);
      f[iList++]=F<Spc3,R>(c13);
      f[iList++]=F<Spc3,R>(c22);
      f[iList++]=F<Spc3,R>(c23);
      f[iList++]=F<Spc3,R>(c33);

      V<R> xx=ran(f,nT,nMax);

      cpmassert(xx[1]<1,loc);
      // any union of spheres is smaller in volume than the enclosing box
      // Was not satisfied till 2006-01-06
      cc.volume=xx[1]*volumeBox;
      // xx[1] should be smaller than 1
      R mPrel=cc.volume*cc.prop.rho;
      cpmassert(mPrel>0,loc);

      Vec3 xCoM; // position vector of center of mass with respect to
      // master
   // origin
      xCoM[1]=xx[2];
      xCoM[2]=xx[3];
      xCoM[3]=xx[4]; // otherwise all centers are in a plane so
      // that this component is zero

      Vec xCoM_; // 'copy' of xCoM with proper dimension
      for (k=1;k<=dimVec;k++){ xCoM_[k]=xCoM[k];}

      for (i=1;i<=n;i++)  cc.xr0[i]-=xCoM_;
      // aha! now the xr0 refer to the center of mass
      R_Vector dis(n);
      for (i=1;i<=n;i++) dis[i]=cc.xr0[i].absVal()+cc.ri[i];
      cc.r=dis.sup();
      // This gives rise to the bounding sphere centered at the
      // center of mass. Has to be multiplied by a factor
      // for the final result.
//   cpmdata<<endl<<"First attempt for radius is "<<cc.r<<endl; //
// temporary
      // this is the right radius for the center of mass as middle-point
      RR3 matInertia;
      matInertia[1][1]=mPrel*xx[5];
      matInertia[1][2]=mPrel*xx[6];
      matInertia[1][3]=mPrel*xx[7];
      matInertia[2][2]=mPrel*xx[8];
      matInertia[2][3]=mPrel*xx[9];
      matInertia[3][3]=mPrel*xx[10];
      matInertia[3][2]=matInertia[2][3];
      matInertia[2][1]=matInertia[1][2];
      matInertia[3][1]=matInertia[1][3];
```

```
      matInertia=Steiner(matInertia,xCoM,mPrel);
      // now the inertia tensor (matrix) refers to the center of mass
      cc.I0=InTens(matInertia);
      if (balance) cc.I0.balance();
// Now we have to adjust to the volume we want
      R volumeAim=Spc3::vol(rs);
      cpmassert(cc.volume>0,loc);
      R scaleFactor3=volumeAim/cc.volume;
      R scaleFactor=pow(scaleFactor3,R(1.)/3);
      R scaleFactor5=pow(scaleFactor3,R(5.)/3);
      cc.I0.scale(scaleFactor5);
      cc.volume=volumeAim;
      cpmassert(cc.volume>0,loc);
      cc.r*=scaleFactor;
      for (i=1;i<=n;i++) cc.ri[i]*=scaleFactor;
      xCoM_*=scaleFactor;
      cv.x+=xCoM_;
      // now x is the center of mass
      for (i=1;i<=n;i++) cc.xr0[i]*=scaleFactor;
   }
   createAllDependentData();
   CPM_MZ
}

void ComPar::spin(const AxVec& omega_)
{
   cv.omega+=omega_;
   updateInternals();
}

// tools for creating unit vectors which rather uniformely fill the
// surface of a sphere

namespace{

Vec fCreate(Z const& i, R const& r, Z const& j)
// create Vec's of length r randomly in i
// For different values of j one should get unrelated functions
// of i
{
   using CpmRoot::ranVal;
   Z shift1=ranVal(Z(137137),j);
   Z shift2=ranVal(Z(13713),j);
   Z shift3=ranVal(Z(1371),j);
   R r0=1;
   Z i1=i+shift1;
   R x=ranVal(r0,i1);
   Z i2=13*i+shift2;
   R y=ranVal(r0,i2);
   Vec a(x,y);
```

```
   if (dimVec==3){
      Z i3=17*i+shift3;
      a[3]=ranVal(r0,i3);
   }
   return (a.polDec().second())*r;
}

bool fCond1(Vec const& x){ x; return true;}
   // trivial condition, needed to satisfy the interface of class Place

bool fCond2(X2<Vec,Vec> const& x, R const& r)
   // condition to ensure minimum distance r
{
   Vec rel=x.c1()-x.c2();
   R d=rel.absVal();
   return (d>=r ? true : false);
}

} // namespace


ComPar::ComPar(R d, const ParticleProperties& pp,
   const DumbbellData& dumData, Z j)
{
   j; // not used
   R rSphere=0.5*d;
   Vec e1(1,0);
   R rc=dumData.rc;
   R re=dumData.re;
   R phi=dumData.phi;
   Vec eMagDir(cos(phi),sin(phi));
   V<Vec> vp(3);
   vp[1]=e1;
   vp[3]=-e1;
   V<R> rList(3);
   rList[1]=re;
   rList[2]=rc;
   rList[3]=re;
   make(rSphere,vp,rList,pp,eMagDir);
}

ComPar::ComPar(R d, const ParticleProperties& pp,
   const PairData& pairData, Z j)
{
   j; // not used
   R rSphere=0.5*d;
   Vec e1(1,0);
   V<Vec> vp(2);
   vp[2]=e1;
```

```
   R r1=pairData.r1;
   R r2=pairData.r2;
   R phi=pairData.phi;

   V<R> rList(2);
   rList[1]=r1;
   rList[2]=r2;

   Vec eMagDir(cos(phi),sin(phi));
   make(rSphere,vp,rList,pp,eMagDir);
}

ComPar::ComPar(R d, const ParticleProperties& pp,
   const NuggetData& nugData, Z j)
{
   Z mL=3;
   Word loc("ComPar(R,ParticleProperties,NuggetData)");
   CPM_MA
   using CpmRoot::ranVal;
   R rSphere=0.5*d;
   Z np=nugData.np;
   cpmassert(np>0,loc);
   if (np==1){
      V<Vec> vp(1); // one zero-vector
      V<R> rp(1,rSphere); // one radius
         // actually we only need vp and rp to have dimension 1,
         // the value of the components does not matter due to the
         // definition of function make.
      Vec dir_j;
      if (pp.J>0){ // the we change dir_j into a randomly selected unit
         // vector
         dir_j=fCreate(np+1,1.,j);
      }
      make(rSphere,vp,rp,pp,dir_j);
   }
   else{
      R overlapFac=nugData.overlapFac;
      R isotropyFac=nugData.isotropyFac;
      Z trialMax=nugData.trialMax;

// now we want to place np points on the surface of a sphere in a
// stochastical manner but nevertheless in a way that even for np small
// this surface will be filled as completly as possible. Since the
// corresponding logic is available with our Place-class, we will assure
// this by prescribing a certain minimum distance. The proper selection
// of this has to be dimension-dependent. Since (irrespective of the fact
// that all spheres are three-dimensional in our situation) the points
// under present consideration have to fill a disk in the 2-dimensional
// version and a sphere in the 3-dimensional version. We divide the
// available 'space' (area or volume) into np equal parts and convert
```

```
// this 'space per point' back into a radius. In this proceture the Pi's
// and numerical factors cancel out and we get the following simple
// prescription for radiusPerPoint.
      Vec v0;
      Z dimSurface=v0.dim()-1;
      // dimension of the surface of a 'sphere' in Spc
      R radiusPerPoint=rSphere*pow((R)np,R(-1.)/dimSurface);
      R safety=(dimSurface==1 ? 3. : 2.);
      // semi-empirical factors
      R dMin=safety*radiusPerPoint*isotropyFac;
      // dMin will serve a s a minimum distance for np points to be
      // placed stochastically within the sphere of radius rSphere.
      // For dimSurface=1, rSphere=1, and np=4, we have radiusPerPoint
      // =1/4 dMin=savety*1/4. Since dMin=sqrt(2) is the upper value
      // compatible with 4 particles (maximum mutual separation), savety
      // has to be smaller than 5.657 presently I use 3. In 3 dimensions
      // one has to reduce this in order to get not too many toners made
      // out of two spheres instead of 3. One could either invest a bit
      // more work in the theory of the process (polyeders for lower
      // n's or to use repelling dynamics for equalizing the distances
      // between random initial points.
      F<Z,Vec> create=F2<Z,R,Z,Vec>(rSphere,j)(fCreate);
      // creates points on the surface of the sphere in a stochastic
      // manner
      F<Vec,bool> cond1(fCond1);
      F<X2<Vec,Vec>,bool> cond2=F1<X2<Vec,Vec>, R, bool>(dMin)(fCond2);
      Place<Vec> pl(create,cond1,cond2); // creating the placing object
      // this function determines whether ComPar::ComPar will always
      // create the same stochastical geometry or always a new one. This,
      // again, depends on function create. Presently it is made to
      // always
      // create new geometries.
      V<Vec> vp=pl(np,trialMax); // invocing the action of the placing
   // object
      np=vp.dim();   // pl may not be able to deliver np positions
      R rPart=safety*radiusPerPoint*overlapFac;
      // its better to define this as a multiple of dMin instead of
      // radiusPerPoint since this allows to correct for different form
      // factors in 2 and 3 dimensions
      V<R> rp(np,rPart);
      Vec dir_j;
      if (pp.J>0){ // the we change dir_j into a randomly selected unit
   // vector
         dir_j=fCreate(np+1,1.,j);
      }
      make(rSphere,vp,rp,pp,dir_j);
   }
   CPM_MZ
}
```

```
ComPar::ComPar(RecordHandler const& rch, Word const& sec, Z j)
{
   Word loc("ComPar(RecordHandler,Word)");
   ContactProperties cp(rch,sec);
   ParticleProperties pp(cp);
   NuggetData nd(rch,"nugget data");
   R r;
   Z ped=2;
   bool b;
   rch.read(sec,"r",r,ped);
   ped=1;
   R dr=0;
   b=rch.read(sec,"dr",dr,ped);
   Iv ivr(r-dr,r+dr); // works also for dr<0
   r=ivr.ran(j);
   cpmassert(r>0,loc);

// getting rho or m trying m first
   R m,rho;

   b=rch.read(sec,"m",m,ped);
   if (b && m>0){
      R vol=CpmDim3::Spc::vol(r);
      cpmassert(vol>0,loc);
      rho=m/vol;
   }
   else{
      ped=2;
      rch.read(sec,"rho",rho,ped);
   }
   pp.rho=rho;
// getting q or sigma trying q first.
   R q=0,sigma=0;
   ped=1;
   b=rch.read(sec,"q",q,ped);
   if (b && q!=0){ // q==0 is considered a way to say that sigma
      // should be tried directly; if also this fails, sigma
      // will be be set to 0, which is OK
      R sur=CpmDim3::Spc::sur(r);
      cpmassert(sur>0,loc);
      sigma=q/sur;
   }
   else{
      ped=1;
      rch.read(sec,"sigma",sigma,ped);
   }
   pp.sigma=sigma;
   ped=1;

   Z color;
```

```
   b=rch.read(sec,"color",color,ped);
   if (b) pp.color=Color::toPal(color);

   R J=0,eps=1,fRelax=0;
   pp.J=J; pp.eps=eps,pp.fRelax=fRelax;

   b=rch.read(sec,"J",pp.J,ped);
   b=rch.read(sec,"eps",pp.eps,ped);
   b=rch.read(sec,"fRelax",pp.fRelax,ped);

// we should not simply read the particle properties
// since rho and sigma might not be there and be replaced by
// q and m
   *this=ComPar(2*r,pp,nd,j);
}

void ComPar::set_m_r(R m, R r,const NuggetData& nd)
{
   if (r<=0) r=SysControl::singRange;
   cpmassert(m>0,"ComPar::set_m_r_E(...)");
   R d=2*r;
   R vol=CpmDim3::Spc::vol(r);
   R rho=m/vol;
   ParticleProperties pp;
   pp.rho=rho;
  // pp.E_=effE;
   ComPar prel(d,pp,nd);
      // this constructor together with the initializations
      // of rho and vol assures that get_m() actually returns
      // m (tested!).
   R rAct=prel.get_r();
   R fac=r/rAct;
   prel*=fac;
      // this adjusts the size but changes the mass
      // now we have to adjust the density to account for this
   R massPrel=prel.get_m();
   R fac2=m/prel.get_m();
   prel.multiplyParameter("rho",fac2);
   R massAct=prel.get_m();
   *this=prel;
}

X2<V<ComPar>,R> ComPar::makePair(R dC, R dT, Z npC, Z npT,
   ParticleProperties ppC, ParticleProperties ppT,
   R overlapFacC, R overlapFacT, R isotropyFacC, R isotropyFacT,
   R facSafty, const Vec& dir )
{
// We understand the arguments 'diameter' as diameter of a sphere having
// the same mass as the irregularly shaped particle.
// for TC=0 we create a single carrier particle with a charge given by
```

```
// ppC.sigma. The settling down-time dt is 0 in this case.

   const R dtDefault=1e-6;
   const R tinyTC=1e-6;
   const Z trialMax=4000;

   Word loc("ComPar::makePair(...)");
   NuggetData ndC;
   ndC.np=npC;
   ndC.overlapFac=overlapFacC;
   ndC.isotropyFac=isotropyFacC;
   ndC.trialMax=trialMax;
   ComPar car(dC, ppC, ndC);
   NuggetData ndT;
   ndT.np=npT;
   ndT.overlapFac=overlapFacT;
   ndT.isotropyFac=isotropyFacT;
   ndT.trialMax=trialMax;
   ComPar ton(dT, ppT, ndT);
   V<ComPar> res(2);
   res[1]=ton;
   res[2]=car;
   R rtAct=ton.get_r();
   R rcAct=car.get_r();
   R dEff=facSafty*(rtAct+rcAct);
   Vec vv=dir*dEff;
   res[1]*=Group(vv); // only toners get moved
// estimating the time needed for settling down
   R s=dEff-rcAct+rtAct; // gap + (security = rtAct)
   R qt=ton.get_q();
   R qc=car.get_q();
   R mt=ton.get_m();
   R mc=car.get_m();
   R mProd=mc*mt;
   cpmassert(mProd>0,loc);
   R mRedInv=(mc+mt)/mProd;
   R Force=influence_PHYS*qt*qc/(dEff*dEff);
      // the charge qt
   R acc=CpmRoot::absVal(Force*mRedInv);
   cpmassert(acc>0,loc);
   R dt=sqrt(2*s/acc);

   return X2<V<ComPar>,R>(res,dt);
}

void ComPar::set_id(Z id1)
{
   if (!cc.idValid){ // a valid id gets never changed
      cc.id=id1;
      cc.idValid=B(true);
```

```
   }
}

void ComPar::updateInternals()
// x, v, ori, omega, qd are assumed to have valid values.
// All quanties (except those that are constant anyway) get changed to
// their actual value
{
// dependent quantities to be buitlt from body-frame data
// (name ending in '0')

   I=cc.I0;
   I*=cv.ori;

   Iinv=Iinv0;
   Iinv*=cv.ori;

   for (Z i=1;i<=n;i++){
      xr[i]=cc.xr0[i]*cv.ori;
      part[i].set_x(cv.x+xr[i]);
      part[i].set_v(cv.v+cv.omega.cross(xr[i]));
   }

// The following treatment of magnetic moment and charge is a stripped
// down version of that of particles. If one would introduce more point
// charges at positions different from x this would be more apparent

   j.setDir(cc.ej0*cv.ori);
   j.set_x(cv.x);
   q.set_x(cv.x);
   cv.qd.set_x(cv.x);
}

void ComPar::createAllDependentData()
// x, v, ori, omega are assumed to have valid values.
// All quanties (except those that are constant anyway) get changed to
// their actual value. This function will be called with data read from
// file. These may be corrupt. This should not stop the program but
// should'nt go unnoticed either. So we will throw exceptions.
{
// dependent quantities to be buitlt from body-frame data
// (name ending in '0')

   I=cc.I0;
   I*=cv.ori;

   Iinv0=!cc.I0;
   Iinv=Iinv0;
   Iinv*=cv.ori;
```

```
   m=cc.volume*cc.prop.rho;
   cpmassert(m>0,"ComPar::createAllDependentData()");
   mInv=1./m;
   n=cc.ri.dim();
   R qVal=cc.get_q();
      // total charge
 // building the SphPar's
   xr=V<Vec>(n);
   part=V<PointPar>(n);
   V<R> frac(n);
   Z i;
   R sum=0;
   for (i=1;i<=n;i++){
      R rii=cc.ri[i];
      rii*=(rii*rii);
      frac[i]=rii;
      sum+=rii;
   }
   cpmassert(sum>0,"ComPar::createAllDependentData()");
   R fracInv=1./sum;
   ContactProperties cp=(ContactProperties)cc.prop;
   for (i=1;i<=n;i++){
      Vec xri=cc.xr0[i]*cv.ori;
      xr[i]=xri;
      part[i].set_x(cv.x+xri);
      part[i].set_v(cv.v+cv.omega.cross(xri));
      part[i].set_r(cc.ri[i]);
      R volumeFraction=frac[i]*fracInv;
      part[i].set_q(volumeFraction*qVal);
      part[i].set_m(volumeFraction*m);
      part[i].set_cp(cp); // does this set E ?
   }
   q=PointCharge(qVal,cv.x);
   q.set_r(SysControl::singRange);
      // important finding 2004-11-02: if the regularization radius
      // varies from particle to particle, the electrostatic forces
      // violate action = reaction
   R jVal=cc.volume*cc.prop.J;
   j=MagDipole((cc.ej0*cv.ori)*jVal,cv.x);
   isMag_=(cc.prop.J!=0.);
   R epsr=cc.prop.eps;
   alpha=cc.volume*3*CpmPhysics::eps0_PHYS*(epsr-1)/(epsr+2);
   cv.qd.set_x(cv.x);
   cv.qd.set_r(SysControl::singRange);
}

void ComPar::operator *=(const Group& g)
{
// analysing argument g, for having in place the most elementary parts
// for efficient usage
```

```
   if (g==Group()) return;
   Vec sh=g.shift();
   AxVec rt=g.rot();

// center of mass
   cv.x*=g;
   cv.v*=rt;
   cv.vDot*=rt;
// electric Dipole Moment
   cv.qd*=rt;

// orientation
   cv.ori*=rt;
   cv.omega=cv.omega.adjointAction(rt);
   cv.omgDot=cv.omgDot.adjointAction(rt);
   // 'cv.omega=' was missing till 2005-02-14

// for x,v; ori, omega having their final values, we update all other
// quantities
   updateInternals();
}

void ComPar::mir_(Z i) // mirror (reflection), clear from code.
{
   if (i>=1 && i<=3){
      cc.mir_(i);
      cv.mir_(i);
      updateInternals();
    }
}

void ComPar::ranRot(R angleRange, Z j)
{
   Group g(Angle(angleRange,DEG));
   Group gj=g.ranVal(get_x(),j);
   operator *=(gj);
}

void ComPar::operator *=(R fac)
{
   Z i;
   cpmassert(fac>0,"ComPar::operator *=(R fac)");
   R fac2=fac*fac;
   R fac3=fac2*fac;
   R fac5=fac3*fac2;
   cc.r*=fac;
   m*=fac3;
   cc.volume*=fac3;
   mInv/=fac3;
   q*=fac2; // surface charge density constant
```

```
   j*=fac3; // volume density of magnetic charge constant
   I.scale(fac5);
   Iinv.scale(1./fac5);
   cc.I0.scale(fac5);
   Iinv0.scale(1./fac5);

   for (i=1;i<=n;i++){
      xr[i]*=fac;
      cc.xr0[i]*=fac;
      cc.ri[i]*=fac; // 2006-04-28
      part[i].cc.p_.r_*=fac;
      part[i].cc.p_.q_*=fac2;
      part[i].cc.p_.m_*=fac3;
      part[i].cv.x_=cv.x+xr[i];
      part[i].cv.v_=cv.v+cv.omega.cross(xr[i]);
   }
}

void ComPar::freeStep(R dt)
{
   if (dt==0.) return;
   cv.t+=dt;
   cv.x+=cv.v*dt;
   if (SysControl::newRigBodyInt){
      R tau=0.5*dt;
      AxVec gc=cv.omega.exp(tau,accExp);
      InTens Ic=I.adjointAction(gc);
      AxVec dIw=(Ic*cv.omega).cross(cv.omega);
      // notice for L:=I*omega, we have with Euler: L'= M
      // here M(=torque)=0 hence L'=0 (' = time derivative)
      // L'=I*omega'+omega.cross(I*omega)
      // Hence I*omega' = (I*omega).cross(omega),
      // omega' = Iinv*((I*omega).cross(omega)).
      // This formula gives much poorer conservation of angular momentum
      // and energy than the formula omega' =
      // Iinv*((Ic*omega).cross(omega))
      // adopted here. The correction I->Ic is semi-empirical
      AxVec dw=Iinv*dIw*dt;
      //AxVec dw=(!Ic)*dIw*dt; // experiment, inferior behavior
         // this is the change of  angular velocity over the whole step.
         // I would have set !Ic instead of Iinv. For unknown
         // reasons this gives energy errors of first order in dt,
         // wheras the Iinv version gives second order energy error.
         // Both variants give only first order error for the
         // conserved angular momentum component for rotation of a
         // magnetized particle in a homogeneous magnetic field.
         // They also give first order for all angular momentum
         // components in the case of free rotation.
      AxVec wc=cv.omega+dw*0.5;
         // only half the velocity increase is to be taken for
```

```
         // generating the position change as a uniform motion over
         // the whole timestep
      AxVec rt=wc.exp(dt,accExp);
      cv.ori*=rt;
      cv.omega+=dw;
   }
   else{
      AxVec rt=cv.omega.exp(dt,accExp);
      cv.ori*=rt;
      // v and omega remain unchanged (this corresponds to using
      // the 'zero-times iteration variant' of the publication
      // which is of first order only if forces depend on velocity
      // (as our friction forces do, also in the hysteresis treatment
      // of friction). One could change v and omega according to
      // memorized values of vDot and omegaDot which presently are not
      // available and would presumably introduce instability.
      // The electric moment is mainly controlled by the direction of the
      // external electric field; so it is not natural to rotate it
      // with the particle. Therefore, don't include qd*=rt;
   }
   updateInternals();
}

void ComPar::reg(void)
{
   if (SysControl::vCrit<=0) return;
   R v1_a=cv.v.absVal();
   R v2_a=cv.omega.absVal()*cc.r; // linear velocity at periphery
   if (v1_a>SysControl::vCrit){
      R red1=redFac*SysControl::vCrit/v1_a;
      cv.v*=red1;
   }
   if (v2_a>SysControl::vCrit){
      R red2=redFac*SysControl::vCrit/v2_a;
      cv.omega*=red2;
      for (Z i=1;i<=n;i++){
         part[i].set_v(cv.v+cv.omega.cross(xr[i]));
      }
   }
}

void ComPar::intAct(R dt, ForTor const& ft, FieldSpace const& fv)
{
   if (dt==0.) return;
// computing accelerations
   Vec force=ft.get_force();
   Vec acc1=force*mInv;
   if (SysControl::aCrit>0){ // only then the correction mechanism
      // applies
      R acc1a=acc1.absVal();
```

```
      if (acc1a>SysControl::aCrit) acc1*=(SysControl::aCrit/acc1a);
   }
   AxVec torque=ft.get_torque();
   AxVec acc2=Iinv*torque;
   if (SysControl::aCrit>0){ // only then the correction mechanism
      // to change acc2 applies
      R acc2a=cc.r*acc2.absVal();
      if (acc2a>SysControl::aCrit) acc2*=(SysControl::aCrit/acc2a);
   }
   Vec dv=acc1*dt;
   AxVec domg=acc2*dt;
   if (SysControl::accMem){
      R dth=0.5*dt;
      dv-=cv.vDot*dth;
      domg-=cv.omgDot*dth;
      cv.vDot=acc1;
      cv.omgDot=acc2;
   }
   cv.v+=dv;
   cv.omega+=domg;

   VecE fe=fv.c1();
   if (fe.absVal()!=0) setElcDipole(dt,fe);
}

ForTor ComPar::forces( const ComPar& cl )const
{
   if (cc.id==cl.cc.id) return ForTor();
    // i.e. particle does not directly interact with itself
   Vec xRel=cl.cv.x-cv.x;
   R dRel=xRel.absVal();
   R rSum=cc.r+cl.cc.r;
   R gapCoarse=dRel-rSum;
   if (gapCoarse > SysControl::intactRange) return ForTor();
      // dRel-r-cl.r is the distance of the clusters if those are
      // idealized as spheres (see meaning of r)
   bool farAway=(gapCoarse>0.);
      // no contact force if farAway==true
   Z i,k;
   ForTor ft1;
// contact interaction
   if (!SysControl::noAdhesion||!SysControl::noElasticRepulsion){
      if (!farAway){
         for (i=1;i<=n;i++){
            PointPar p1=part[i];
            for (k=1;k<=cl.n;k++){ // notice that i and k refer
            //to different clusters, so that
            // i==k plays no special role
               PointPar p2=cl.part[k];
               Spc sp12=p1.frcCen(p2);
```

```
                Vec v1=get_v(sp12);
                Vec v2=cl.get_v(sp12);
                Force f12=p1.forceContact(p2,v1,v2);
                ft1.add(f12,sp12,get_x());
             }
          }
       }
    }
    if(!SysControl::noInteractionChargeToCharge ){
       ft1.addForce(q.force(cl.q));
    }
    if(!SysControl::noMagneticInteractionChargeToCharge){
       // this control structure implies that one may enable the magnetic
       // interaction while disabling the electrostatic one. This is
       // expected to be useful since for non-relativistic velocities
       // the magnetic interaction would else be dominated by the
       // electrostatic one
       ft1.addForce(getPointCharge().forceMag(get_v(),
          cl.getPointCharge(),cl.get_v()));
          // this contains the Lorentz force from the magnetic
          // field which pp generates (if it moves)
    }
    // magnetic interaction
    if (!SysControl::noInteractionMagnetToMagnet){
       ft1+=j.forceAndTorque(cl.j);
    }
    if (!SysControl::noInteractionElectricDipole){
       // electric dipole interaction
       ft1+=cv.qd.forceAndTorque(cl.cv.qd);
    }
    // gravitational interaction
    if(!SysControl::noInteractionMassToMass){
       PointMass pm=getPointMass();
       PointMass pm_cl=cl.getPointMass();
       ft1.addForce(pm.force(pm_cl));
    }
    return ft1;
}

R ComPar::potElastic( const ComPar& cl )const
{
    if (idn(cl)) return 0;
     // a particle does not directly interact with itself
    Vec xRel=cl.cv.x-cv.x;
    R dRel=xRel.absVal();
    R rSum=cc.r+cl.cc.r;
    R gapCoarse=dRel-rSum;
    R res=0;
    if (gapCoarse > 0) return res;
    for (Z i=1;i<=n;i++){
```

```
      PointPar pi=part[i];
      for (Z k=1;k<=cl.n;k++){ // notice that i and k refer
            //to different clusters, so that
            // i==k plays no special role
         PointPar pk=cl.part[k];
         res+=pi.potElastic(pk);
      }
   }
   return res;
}

R ComPar::pot( const ComPar& cl )const
{
   // electric dipole term yet missing; will be added after the
   // magnetic term has been tested
   // also elastic and adhesive part missing
   if (cc.id==cl.cc.id) return 0;
    // i.e. particle does not directly interact with itself
   R res=0;
   if(!SysControl::noInteractionChargeToCharge){
      res+=getPointCharge().pot(cl.getPointCharge());
   // electrostatic energy
   }
   if (!SysControl::noInteractionMagnetToMagnet){
      res+=get_j().pot(cl.get_j());
      // magnetostatic energy (included 2004-10-31)
   }
   if(!SysControl::noInteractionMassToMass){
      res+=getPointMass().pot(cl.getPointMass());
      // gravitational energy
   }
   res+=potElastic(cl);
   return res;
}

R ComPar::pot( const FieldSpace& f)const
{
   R res=0;
   if(!SysControl::noInteractionElectricField){
      res+=q.potEnr(f.c1());
   }
   if(!SysControl::noInteractionMagneticField){
      res+=j.potEnr(f.c2());
   }
   if(!SysControl::noInteractionGravitationField){
      res+=getPointMass().potEnr(f.c3());
   }
   return res;
}
```

```
AxVec ComPar::angMom(Spc const& p)const
{
   Vec xp=get_x()-p;
   AxVec res=xp.cross(mom())+I*cv.omega;
   return res;
}

ForTor ComPar::forces( const VectorField& f)const
{
   static const R c=0.2; // controls differentiation step,
      //should be smaller than 0.5
   R cr=c*cc.r;
   FieldSpace fs=f(cv.x);
   ForTor  res;
   if (!SysControl::noInteractionMagneticField){
      FieldBH BHfield=f.c2();
      res=j.forceAndTorque(BHfield,cr);
      VecBH vbh=fs.c2();
      Vec LorentzForce=vbh.cross(get_v())*(-get_q());
      res.addForce(LorentzForce);
   }
   if (!SysControl::noInteractionElectricField){
      VecE E=fs.c1();
      res.addForce(q.get_q()*E);
      FieldE Efield=f.c1();
      res+=cv.qd.forceAndTorque(Efield,cr);
   }
   if (!SysControl::noInteractionGravitationField){
      VecG G=fs.c3();
      res.addForce(m*G);
   }
   return res;
}

ForTor ComPar::forces( const FieldSpace& fs)const
{
   ForTor  res;
   if (!SysControl::noInteractionMagneticField){
      // there are good reasons from my electrophotography model
      // that in a 2D model there are magnetic field lines
      // lying in the 2D system plane. So it is not allowed there
      // to model the magnetic fields as AxVec, what would be the
      // natural choice for a common representation of the Lorentz
      // force in 2D and 3D.
      VecBH fsBH=fs.c2();
      res.addTorque(j.torque(fsBH));
      Vec LorentzForce=fsBH.cross(get_v())*(-get_q());
      res.addForce(LorentzForce);
   }
   if (!SysControl::noInteractionElectricField){
```

```
      VecE fsE=fs.c1();
      res.addForce(q.force(fsE));
   }
   if (!SysControl::noInteractionGravitationField){
      VecG fsG=fs.c3();
      res.addForce(m*fsG);
   }
   return res;
}

ForTor ComPar::forces( const RigidObject& ro)const
{
   if (SysControl::eta==0. || SysControl::noFriction) return ForTor();
   R eta=SysControl::eta;
   using CpmRoot::Pi;
   Vec vRel=cv.v-ro.get_v();
   AxVec omegaRel=cv.omega-ro.getOmega();
      // is OK for angular velocities !!!
   R vRelAbs=vRel.absVal();
   R vRelRotAbs=cc.r*omegaRel.absVal();
   R pi_eta_r=Pi*eta*cc.r;
   Force force(-vRel*6*pi_eta_r);
   AxVec torque=-omegaRel*8*pi_eta_r*cc.r*cc.r;
   return ForTor(force,torque);
}

R ComPar::disVal(const ComPar& cl)const
   // not limited by 1 as most disVal functions. Roughly spoken, this
   // is actual distance in units of particle diameter.
{
   using CpmLinAlg::R2;
   R dTrans=(cv.x-cl.cv.x).absVal();
   R sr=(cc.r+cl.cc.r);
   R phi=(cv.ori*(-cl.cv.ori)).phi();
   R dRot=sr*phi;
      // we transform a rotation into a peripheral distance in order to
      // make spatial distances and rotational ones directly comparable.
   R2 dPos2(dTrans,dRot);
   R dPos=dPos2.absVal();
   return dPos/sr;
}

void ComPar::mark(Graph& g)const
{
   static const Color particleColor(GREEN);
   for (Z i=1;i<=n;i++) part[i].mark(g,particleColor);
}

void ComPar::mark(const Camera& sc, const Color& col)const
{
```

```
static bool minimalSkeleton=true;
static bool markBoundingSphere=false;
if (!markComParMec){
   const R reduct=0.4; // was 0.2
   if (col.transparent()) return;
   Z i;
   if (!SysControl::coloredParticles
      && SysControl::howToMarkParticles==0 ){
         // Then s system of orthogonal tripods is drawn
      Vec e1(1,0);
      e1*=cv.ori; // now this is a vector with a fixed
         //direction in the body frame
      Z k;
      for (i=1;i<=n;i++){
         R ri=part[i].cc.p_.r_;
         Spc xi=part[i].cv.x_;
         V<Vec> tp=xr[i].base(e1);
            // two vectors xr[i], e1 determine an orthonormal tripod
         for (k=1;k<=tp.dim();k++) tp[k]*=ri;
         sc.mark(cv.x,xi+tp[1],col);
         if (minimalSkeleton){ // most simple representation
            sc.mark(xi,xi+tp[2],col);
            if (tp.dim()==3) sc.mark(xi,xi+tp[3],col);
         }
         else{ // actual space fill well represented
            sc.mark(xi-tp[2],xi+tp[2],col);
            if (tp.dim()==3) sc.mark(xi-tp[3],xi+tp[3],col);
         }
      }
      if (SysControl::dtMult>0){
         R dt=SysControl::tStep*SysControl::dtMult;
         // and a velocity indicator for the center of mass
         Spc xcm=cv.x;
         Vec vs=cv.v;
         Spc xcm2=xcm+vs*dt;
         sc.mark(xcm,xcm2,col);
      }
   }
   else{ // independent representation of the components
      // each of them has now a velocity indicator so that
      // rotation of the particle becomes visible directly
      for (i=1;i<=n;i++) part[i].mark(sc,col);
      if (markBoundingSphere){
         PointPar ce=part[1]; // marking the bounding sphere too
         ce.set_x(cv.x);
         ce.set_r(cc.r);
         ce.mark(sc,col);
      }
   }
   if (isMag_){ // the direction of the magnetic moment is to be
```

```
      // indicated
      Color colJ(col);
         // color for marking the magnetic moment
      Vec jj=j.get_j().polDec().c2()*cc.r;
      Spc xSymbol=cv.x+jj;
      PointPar sp1;
      R rSymbol=reduct*cc.r/::sqrt((R)n);
      sp1.set_r(rSymbol);
      if (SysControl::coloredParticles && Spc::dimension()==2){
         colJ=Color(WHITE);
         // here a disk looks better as the center of the particle
         // than as the hat of the 'arrow'
         sp1.set_x(cv.x);
      }
      else{
         sp1.set_x(xSymbol);
      }
      sp1.mark(sc,colJ); // show symbol particle
      sc.mark(cv.x,xSymbol,colJ); // show connecting line
   }
}
else{ // markComParMec
   static bool firstrun=true;
   static R angMomFac=1;
   static R angVelFac=1;
   Word loc="ComPar::mark(Camera,Color)";
   R r=get_r();
   Vec e1=I.e1()*I.i1();
   Vec e2=I.e2()*I.i2();
   Vec e3=I.e3()*I.i3();
   R a1=e1.absVal();
   cpmassert(a1>0,loc);
   R renorm=r/a1;
   Spc x=get_x();
   e1*=renorm;
   e2*=renorm;
   e3*=renorm;
   sc.mark(x,x+e1,col);
   sc.mark(x,x+e2,col);
   sc.mark(x,x+e3,col);
   Vec om(cv.omega[1],cv.omega[2],cv.omega[3]);
   if (firstrun){
      R oma=om.absVal();
      if (oma>0) angVelFac=r/oma;
   }
   om*=angVelFac;
   Spc x1=x+om;
   PointPar sym1;
   R r1=r*0.02;
   sym1.set_r(r1);
```

```
      sym1.set_x(x1);
      sc.mark(x,x1,col);
      sym1.mark(sc);
      AxVec io=I*cv.omega;
      Vec iom(io[1],io[2],io[3]);
      if (firstrun){
         R ioma=iom.absVal();
         if (ioma>0) angMomFac=r/ioma;
         firstrun=false;
      }
      iom*=angMomFac;
      Spc x2=x+iom;
      PointPar sym2;
      R r2=r*0.04;
      sym2.set_r(r2);
      sym2.set_x(x2);
      sc.mark(x,x2,col);
      sym2.mark(sc);
   }
}

// now I/O for ComPar works through the division of data into a
// ComParC (constant data) and a ComParV (variable data) component.
// Till 02-03-13 writing on file and MPI-messaging used different
// functions, so that it was hard to see whether strange
// dynamical behavior of configurations read from file was based
// on a difference in available data

bool ComPar::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("ComPar",str);
   cpmp(cc);
   cpmp(cv);
   return true;
}

bool ComPar::scanFrom(istream& str)
{
   cpms(cc);
   cpms(cv);
   createAllDependentData();
   return true;
}

void ComPar::setDirJ(const Vec& dir)
{
   Vec v1=getDirJ();
   AxVec av=v1.rotTo(dir);
   operator*=(Group(cv.x,av));
}
```

```
// new version that expects E to be the consistent mean field
// (that is constant during a complete integration step (step
// as opposed to steplet) so that fRelax and dt need not to
// be used
void ComPar::setElcDipole(R dt, const VecE& E)
{
   static R Emax=3e6; // maximum electric field that
   // may exist in air: 3MV/m
   if (SysControl::noInteractionChargeToCharge ||
      SysControl::noInteractionElectricDipole) return;
   dt; // not used
   VecE Eeff=E;
   R ea=Eeff.absVal();
   if (ea>Emax) Eeff*=(Emax/ea);
   Vec qdInduced=Eeff*alpha;
   cv.qd.setMoment(qdInduced);
}

bool ComPar::multiplyParameter(Word const& w, R fac)
{
   Word loc("ComPar::multiplyParameter(...)");
   if (w=="charge"||w=="sigma"){
      if (fac==1.) return true;
      cc.prop.sigma*=fac;
      q*= fac;
      for (Z i=1;i<=n;i++) part[i].cc.p_.q_*=fac;
   }
   else if (w=="alphaAdhesion"){
      if (fac==1.) return true;
      cc.prop.alphaAdhesion_*=fac;
     // for (Z i=1;i<=n;i++) part[i].cp.alphaAdhesion*=fac;
      for (Z i=1;i<=n;i++) part[i].cc.p_.alphaAdhesion_*=fac;
   }
   else if (w=="relativeRangeAdhesion"){
      if (fac==1.) return true;
      cc.prop.relativeRangeAdhesion_*=fac;
      for (Z i=1;i<=n;i++) part[i].cc.p_.relativeRangeAdhesion_*=fac;
   }
   else if (w=="effectiveE"){
      if (fac==1.) return true;
      cc.prop.E_*=fac;
      for (Z i=1;i<=n;i++) part[i].cc.p_.E_*=fac;
   }
   else if (w=="scnr"){
      if (fac==1.) return true;
      cc.prop.scnr_*=fac;
      for (Z i=1;i<=n;i++) part[i].cc.p_.scnr_*=fac;
   }
   else if (w=="J"){
```

```
      if (fac==1.) return true;
      cc.prop.J*=fac;
      j*=fac;
      if (fac==0) isMag_=false;
   }
   else if (w=="rho"){
      if (fac==1.) return true;
      cpmassert(fac!=0,loc);
      R facInv=1./fac;
      cc.prop.rho*=fac;
      m*=fac;
      mInv*=facInv;
      I.scale(fac); // notice that the geometry is fixed
      Iinv.scale(facInv);
      cc.I0.scale(fac);
      Iinv0.scale(facInv);
      for (Z i=1;i<=n;i++) part[i].cc.p_.m_*=fac;
   }
   else {
        // even if fac==1 we return false if w does not match a defined
        // meaning
      cpmwarning("ComPar::multiplyParameter(...): "&w&
         " is invalid argument");
      return false;
   }
      // should not stop for a parameter which is not in the list
   return true;
}

 void ComPar::set_cp(const ContactProperties& cp)
  // setting the ContactProperties subset of the ParticleProperties
{
   cc.prop.set_cp(cp);
   for (Z i=1;i<=n;i++) part[i].set_cp(cp);
}

void ComPar::set_pp(const ParticleProperties& pp)
 // setting the ContactProperties subset of the ParticleProperties
{
   set_cp(pp);
   R fac,valOld,valNew;
   Word w;

   w="J";
   valOld=cc.prop.J;

   if (valOld!=0){
      valNew=pp.J;
      fac=valNew/valOld;
      multiplyParameter(w,fac);
```

```
         // if fac==0, this function takes care of updating isMag
   }
   else{
      cc.prop.J=pp.J;
      R jVal=cc.volume*cc.prop.J;
      j=MagDipole((cc.ej0*cv.ori)*jVal,cv.x);
      isMag_=(jVal!=0);
   }

   w="rho";
   valOld=cc.prop.rho;
   valNew=pp.rho;
   cpmassert(valOld!=0,"ComPar::set_pp(...)");
   fac=valNew/valOld;
   multiplyParameter(w,fac);

   cc.prop.eps=pp.eps;
}

void ComPar::set_E(R E)
{
   ContactProperties cp=cc.prop;
   cp.E_=E;
   set_cp(cp);
}

void ComPar::setCofOfNorRes(R cnr)
{
   ContactProperties cp=cc.prop;
   cp.scnr_=cnr*cnr;
   set_cp(cp);
}

R ComPar::gap(const RigidObject& ro, Z i)const
{
   Z k;
   Vo<R> rox(n);
   Vo<R> roxAbs(n);
   for (k=1;k<=n;k++) rox[k]=ro.relCor(part[k].get_x(),i);
   for (k=1;k<=n;k++) roxAbs[k]=CpmRoot::absVal(rox[k]);
   for (k=1;k<=n;k++){
      roxAbs[k]-=cc.ri[k];
   }
   pair<Z,Z> pzz=roxAbs.indInfSup();
   Z iMin=pzz.first;
   R d0=rox[iMin];
   R r0=cc.ri[iMin];
   return d0>0 ? d0-r0 : d0+r0;
}
```

```
R ComPar::gap(const ComPar& cl)const
{
   Vec xRel=cl.cv.x-cv.x;
   R dRel=xRel.absVal();
   R rSum=cc.r+cl.cc.r;
   R gapCoarse=dRel-rSum;
   if (gapCoarse>0) return gapCoarse;
   R res=rSum;
   for (Z i=1;i<=n;i++){
      PointPar pi=part[i];
      for (Z k=1;k<=cl.n;k++){ // notice that i and k refer
            //to different clusters, so that
            // i==k plays no special role
         PointPar pk=cl.part[k];
         R gik=pi.gap(pk);
         if (gik<res) res=gik;
      }
   }
   return res;
}

SubsetOfSpace ComPar::toSubOfSpc()const
{
   SubsetOfSpace res=Sphere(part[1].get_x(),part[1].get_r());
   for (Z i=2;i<=n;i++){
      res=res|Sphere(part[i].get_x(),part[i].get_r());
   }
   return res;
}

ComParC ComPar::getC()const
{
   return cc;
}

ComParV ComPar::getV()const
{
   return cv;
}

void ComPar::set(const ComParC& clc, const ComParV& clv)
{
   cc=clc;
   cv=clv;
   createAllDependentData();
}

ComPar::ComPar(const ComParC& clc, const ComParV& clv)
{
   set(clc,clv);
```

```
}

V<ComPar> ComPar::neutralize(const V<ComPar>& vc)
{
   Z mL=4;
   Word loc("ComPar::neutralize(V<ComPar>&)");
   cpmmessage(mL,loc&" started");
   Z i,nvc=vc.dim();
   if (nvc==1) return vc;
   R qP=0;
   R qN=0;
   R qi;
   for (i=1;i<=nvc;i++){
      qi=vc[i].get_q();
      if (qi>0) qP+=qi; else qN-=qi; //qN becomes positive too
   }
   if (cpmverbose>mL){
      cpmcerr<<endl<<" prior to neutralizing"<<endl;
      cpmdebug(qP);
      cpmdebug(qN);
      cpmdebug(qP-qN);
   }
   if (qP==0. || qN==0.){
      cpmwarning(loc&
      ": all charges have the same sign, input returned without change"
      );
      return vc;
   }
   R f=qP/qN;
   f=sqrt(f);
   R fInv=1./f;
   V<ComPar> res(vc);
   for (i=1;i<=nvc;i++){
      qi=vc[i].get_q();
      if (qi>=0){
         res[i].set_q(qi*fInv);
         res[i].cc.prop.sigma*=fInv;
            // this is needed since IO is such that
            // charge isn't written to file but reconstructed
            // from r and prop.sigma. This is more legacy than
            // a good idea
      }
      else{
         res[i].set_q(qi*f);
         res[i].cc.prop.sigma*=f;
      }
   }
   qP=0;
   qN=0;
   for (i=1;i<=nvc;i++){
```

```
      qi=res[i].get_q();
      if (qi>0) qP+=qi; else qN-=qi;
   }
   if (cpmverbose>mL){
      cpmcerr<<endl<<" after neutralizing"<<endl;
      cpmdebug(qP);
      cpmdebug(qN);
      cpmdebug(qP-qN);
   }
   cpmmessage(mL,loc&" done");
   return res;
}

SourceOfVectorField ComPar::source()const
{
   return SourceOfVectorField(q,cv.qd,j,getPointMass());
}

VectorField ComPar::farField(const V<ComPar>& par,
   const V<Spc>& pointList)
{
   Z i,np=par.dim();
   V<SourceOfVectorField> sl(np);
   for (i=1;i<=np;i++) sl[i]=par[i].source();
   sl=SourceOfVectorField::redistribute(sl,pointList);
   VectorField res;
   for (i=1;i<=sl.dim();i++) res+=sl[i].field();
   return res;
}

VectorField ComPar::meanField(const V<ComPar>& par,
   const V<Spc>& pointList)
{
   Z i,j,np=par.dim(),n=pointList.dim();
   V<SourceOfVectorField> sl(np);
   for (i=1;i<=np;i++) sl[i]=par[i].source();
   sl=SourceOfVectorField::redistribute(sl,pointList);
      // complexity proportional to np
   V<FieldSpace> val(n);
   for (i=1;i<=n;i++){ // complexity proportional to n*n not np*np !
      Spc xi=pointList[i];
      for (j=1;j<=n;j++){
         if (i==j) continue;
         val[i]+=sl[j].eval(xi);
      }
   }
   return VectorField(pointList,val);
}

B ComPar::isVal(const V<ComParV>& vv)
```

```
   // returns true if there are no reasons to
   // reject data vv
   // Present implementation based on comparing velocities with
   // SysControl::vCrit
{
   if (SysControl::vCrit<=0) return B(true);
   Z mL=2;
   const R lazyFactor=1.5;
      // since the velocity never exceeds 2*SysControl::vCrit
      // setting this factor >2 prevents any influence from it.
   static Word loc("ComPar::isVal(const V<ComParV>& vv)");
   CPM_MA
   R vStop=lazyFactor*SysControl::vCrit;
   Z i,j,d=Spc::dimension(),n=vv.dim();
   R_Matrix vel(d,n);
   Z count=0;
   for (j=1;j<=n;j++){
      Vec vj=vv[j].v;
      if (vj.absVal()>vStop) count++;
      for (i=1;i<=d;i++){
         vel[i][j]=cpmabs(vj[i]);
      }
   }
   V<R> vMax(d);
   for (i=1;i<=d;i++){
      vMax[i]=vel[i].sup();
   }
   ostringstream ost;
   ost<<endl<<"Output of ComPar::isVal(const V<ComParV>& vv)";
   ost<<endl<<"concerning maximum velocity in i-direction, i=1,2,(3)";

   for (i=1;i<=d;i++){
      ost<<endl<<"vMax["<<i<<"]="<<vMax[i];
   }
   ost<<endl<<"SysControl::aCrit="<<SysControl::aCrit;
   ost<<endl<<"SysControl::vCrit="<<SysControl::vCrit;
   ost<<endl<<"vStop="<<vStop;
   B res(true);
   if (count>0) res=B(false);
   if (!res){
      cpmwarning(loc&" "&cpm(count)&" particles move unexpectedly fast");
      cpmmessage("Details: "&Word(ost.str()));
   }
   CPM_MZ
   return res;
}

B ComPar::isVal(const V<ComPar>& vc)
{
   Z vd=vc.dim();
```

```
   V<ComParV> vv(vd);
   for (Z i=1;i<=vd;i++) vv[i]=vc[i].cv;
   return ComPar::isVal(vv);
}

R ComPar::sclVal(const Word& w)const
{
   if (w=="mass") return m;
   else if (w=="velocity") return cv.v.absVal();
   else if (w=="angular_velocity") return cv.omega.absVal();
   else if (w=="kinetic_energy") return kinEnr();
   else if (w=="charge") return get_q();
   else if (w=="electric_dipole") return get_qd().absVal();
   else if (w=="electric_polarization")
      return get_qd().absVal()/cc.volume;
   else if (w=="magnetic_dipole") return get_j().absVal();
   else if (w=="volume") return cc.volume;
   else if (w=="surface") return getSurface();
   else cpmerror("ComPar::sclVal("&w&") not defined");
   return 0;
}

void ComPar::showProperty(Word const& theme, V<ComPar> const& vp,
   Camera const& ca, R_Vector const& parameters)
// here 'Carrier' means particle with non-vanishing magnetic moment
// 'Toner' means particle which is not a carrier particle.
{
   Z mL=3;
   Word loc("ComPar::showProperty(...)");
   CPM_MA
   const Z nBinMax=200;
   const R electricMomentCompareFactor=4;
   const bool histogramCumulative=true;
   Z i,j,nPlanes,nBins,n;
   R_Matrix val;
   Iv iv;
   if (theme=="HistogramMagneticMoment"){
      nPlanes=3;
      n=vp.dim();
      V<bool> sel(n);
      for (i=1;i<=n;i++) sel[i]=vp[i].isMag();
      V<ComPar> vSel=vp.select(sel);
      n=vSel.dim();
      nBins=inf<Z>(n,nBinMax);
      val=R_Matrix(nPlanes,n);
      R_Vector scale(n);
      for (j=1;j<=n;j++)
            scale[j]=vSel[j].get_j().getMoment().absVal();
      R maxv=scale.sup();
      iv=Iv(-maxv,maxv);
```

```
      for (i=1;i<=nPlanes;i++)
         for (j=1;j<=n;j++)
            val[i][j]=vSel[j].get_j().getMoment()[i];

   }
   else if (theme=="HistogramTonerElectricMoment"){
      nPlanes=3;
      n=vp.dim();
      V<bool> sel(n);
      for (i=1;i<=n;i++) sel[i]=!vp[i].isMag();
      V<ComPar> vSel=vp.select(sel);
      n=vSel.dim();
      nBins=inf<Z>(n,nBinMax);
      val=R_Matrix(nPlanes,n);
      R E_bias=parameters[1];
      R_Vector alpha(n);
      for (j=1;j<=n;j++)
            alpha[j]=vSel[j].getAlpha();
      R maxv=electricMomentCompareFactor*E_bias*alpha.sup();
      iv=Iv(-maxv,maxv);
      for (i=1;i<=nPlanes;i++)
         for (j=1;j<=n;j++)
            val[i][j]=vSel[j].get_qd().getMoment()[i];
   }
   else if (theme=="HistogramCarrierElectricMoment"){
      nPlanes=3;
      n=vp.dim();
      V<bool> sel(n);
      for (i=1;i<=n;i++) sel[i]=vp[i].isMag();
      V<ComPar> vSel=vp.select(sel);
      n=vSel.dim();
      nBins=inf<Z>(n,nBinMax);
      val=R_Matrix(nPlanes,n);
      R E_bias=parameters[1];
      R_Vector alpha(n);
      for (j=1;j<=n;j++)
            alpha[j]=vSel[j].getAlpha();
      R maxv=electricMomentCompareFactor*E_bias*alpha.sup();
      iv=Iv(-maxv,maxv);
      for (i=1;i<=nPlanes;i++)
         for (j=1;j<=n;j++)
            val[i][j]=vSel[j].get_qd().getMoment()[i];
   }
   else if (theme=="HistogramTonerVelocity"){
      nPlanes=3;
      n=vp.dim();
      V<bool> sel(n);
      for (i=1;i<=n;i++) sel[i]=!vp[i].isMag();
      V<ComPar> vSel=vp.select(sel);
      n=vSel.dim();
```

```
   nBins=inf<Z>(n,nBinMax);
   val=R_Matrix(nPlanes,n);
   R maxv=parameters[1];
   iv=Iv(-maxv,maxv);
   for (i=1;i<=nPlanes;i++)
      for (j=1;j<=n;j++)
         val[i][j]=vSel[j].get_v()[i];
}
else if (theme=="HistogramCarrierVelocity"){
   nPlanes=3;
   n=vp.dim();
   V<bool> sel(n);
   for (i=1;i<=n;i++) sel[i]=vp[i].isMag();
   V<ComPar> vSel=vp.select(sel);
   n=vSel.dim();
   nBins=inf<Z>(n,nBinMax);
   val=R_Matrix(nPlanes,n);
   R maxv=parameters[1];
   iv=Iv(-maxv,maxv);
   for (i=1;i<=nPlanes;i++)
      for (j=1;j<=n;j++)
         val[i][j]=vSel[j].get_v()[i];
}
else if (theme=="Histogram_xPosition"){
   nPlanes=1;
   n=vp.dim();
   nBins=inf<Z>(n,nBinMax);
   val=R_Matrix(nPlanes,n);
   R maxv=parameters[1];
   iv=Iv(-maxv,maxv);
   for (i=1;i<=nPlanes;i++)
      for (j=1;j<=n;j++)
         val[i][j]=vp[j].get_x()[1];
}
else if (theme=="Histogram_yPosition"){
   nPlanes=1;
   n=vp.dim();
   nBins=inf<Z>(n,nBinMax);
   val=R_Matrix(nPlanes,n);
   R maxv=parameters[1];
   iv=Iv(-maxv,maxv);
   for (i=1;i<=nPlanes;i++)
      for (j=1;j<=n;j++)
         val[i][j]=vp[j].get_x()[2];
}
else if (theme=="Histogram_zPosition"){
   cpmassert(Spc::dimension()==3,loc);
   nPlanes=1;
   n=vp.dim();
   nBins=inf<Z>(n,nBinMax);
```

```
      val=R_Matrix(nPlanes,n);
      R maxv=parameters[1];
      iv=Iv(-maxv,maxv);
      for (i=1;i<=nPlanes;i++)
         for (j=1;j<=n;j++)
            val[i][j]=vp[j].get_x()[3];
   }
   else{
      cpmerror(loc&": unvalid theme "&theme);
   }
   Histogram his(nPlanes,nBins,iv[1],iv[2],theme);
   for (i=1;i<=nPlanes;i++)
      for (j=1;j<=n;j++)
         his.put(i,val[i][j]);
   his.show(ca.getGraph(),histogramCumulative);
   CPM_MZ
}

void ComPar::showVelocities(const V<ComPar>& vp, const Camera& ca)
{
   Z i,j,d=Spc::dimension(),n=vp.dim();
   Z nBins=64;
   R vMin=0;
   R vMax=2*SysControl::vCrit;
   Histogram his(d,nBins,vMin,vMax);
   for (i=1;i<=d;i++){
      for (j=1;j<=n;j++){
         his.put(i,cpmabs((vp[j].get_v())[i]));
      }
   }
   his.show(ca.getGraph());
}

Z ComPar::com(const ComPar& p)const{ return cv.x.com(p.cv.x);}
// comparison based solely on position
```

36 cpmcomparx.h

```
//? cpmcomparx.h
//? Status of work 2008-10-25.
//?
/*
   cpmcomparx.h
   Description: compar stands for composite particles. Declares a class of
   irregularly shaped particles as clusters of overlapping spherical
   particles. In close analogy to this, also simple spherical
   particles are treated.

   The treatment of rigid body motion is in full agreement with Huette p.
   E33 equ. (16). The time step according to the half-step linear advance
   scheme (explicit midpoint method) that arrises from the MPM-method
   (see
   Ulrich Mutze: Predicting Classical Motion Directly from the Action
   Principle II, http://www.ma.utexas.edu/mp_arc/index-99.html, entry
   99-271) after limit of velocity spread to zero is implemented here
   much more elegantly than before.

   Emphasis was put on defining all structures such that no formal
   changes are needed to switch from 2 space dimensions to three ones.
   This askes for a consequent distinction between axial and polar
   vectors, which brings out the physical structure very nicely. I had to
   work hard (more than 50 pages of calculations to see things clear
   enough) to find the present framework although it does not deviate in
   any point from best practicies.

*/

using namespace CpmRoot;

using CPM_DIM::Vec;
using CPM_DIM::AxVec;
using CPM_DIM::Spc;
using CPM_DIM::Group;
using CPM_DIM::RefSys;
using CPM_DIM::Plane;
using CPM_DIM::SubsetOfSpace;
using CPM_DIM::InTens;
using CPM_DIM::Rdim;
using CPM_DIM::Rdim_;
using CPM_DIM::Box;

using CPM_PHYSICS::Force;
using CPM_PHYSICS::ForTor;
using CPM_PHYSICS::PointMass;
using CPM_PHYSICS::PointCharge;
```

```
using CPM_PHYSICS::ElcDipole;
using CPM_PHYSICS::MagDipole;
using CPM_PHYSICS::PatchCharge;
using CPM_PHYSICS::FieldSpace;
using CPM_PHYSICS::VecE;
using CPM_PHYSICS::VecBH;
using CPM_PHYSICS::VecG;
using CPM_PHYSICS::ScalarField;
using CPM_PHYSICS::VectorField;
using CPM_PHYSICS::SourceOfVectorField;
using CPM_PHYSICS::FieldE;
using CPM_PHYSICS::FieldBH;
using CPM_PHYSICS::FieldG;

using CPM_CAMERA::Camera;

using CpmRootX::R1;
using CpmRootX::Z1;
using CpmRootX::B;
using CpmRootX::RecordHandler;
using CpmProperties::SysControl;
using CpmProperties::BodyProperties;
using CpmProperties::ContactProperties;
using CpmProperties::ParticleProperties;
using CpmProperties::DumbbellData;
using CpmProperties::NuggetData;
using CpmProperties::PairData;
using CpmLinAlg::R2;
using CpmGraphics::Graph;
using CpmImaging::Color;
using CpmImaging::Palette;
using CpmImaging::INVISIBLE;
using CpmImaging::RED;
using CpmImaging::GREEN;
using CpmArrays::V;
using CpmArrays::X2;
using CpmArrays::X4;
using CpmArrays::R_Vector;
using CpmArrays::R_Matrix;
using CpmFunctions::F;
using CpmGeo::Angle;
using CpmGeo::DEG;
using CpmGeo::Iv;

// notice that we are here in namespace CpmRigidBody2 or CpmRigidBody3
// so we are introducing here two variables for each name
// e.g. CpmRigidBody2::accExp and CpmRigidBody3::accExp

//doc namespace CpmRigidBodyx{
```

```
extern bool accExp;
// accurate exponential function (from the Lie algebra of the rotation
// group to this group). If this is false a fast and generally
// sufficient approximation is used. The free rotation of asymmetrical
// ridgid bodies behaves much more stable if accExp=true

extern bool markComParMec;
   // if this is true, the function ComPar::mark(Camera...) shows
   // the axes of inertia and the vectors of angular velocity and
   // angular momentum

extern bool balance;
   // if this is true, all tensors of inertia get changed
   // to the effect that their moments of inertia are equal
   // to the mean value of the 3 original values

///////////////// class RigidObject ////////////////////////////////////

   class RigidObject{ // describes the kinematical state of a rigid body
      // describes the kinematical state of a rigid body. The kinematical
      // state (sometimes also called dynamical state), by definition,
      // does not only carry full information on a configuration ('a
      // point in configuration space') but on any of the following
      // physically equivalent concepts:
      // (i) two configurations for two points in time which are 'close
      // together'.
      // (ii) an 'infinitesimal' path in configuration space
      // (iii) a point in configuration space and a vector from the
      // local vector space (tangent space) in the sense of differential
      // geometry.

      // The configuration of a RigidObject is described by the position
      // of a distinguished point and an element of the rotation group
      // SO(3) which generates the present 'orientation of the object in
      // space' from some reference orientation. Less formally the SO(3)
      // element will be called angular position (in German:'Winkellage';
      // the German word 'Lage' nicely combines partial meanings of lay,
      // location, position, constellation to a general notion of 'the
      // characteristics of placement in space').

      // If we fix an origin in space, we may describe position in space
      // by a position vector and combine the position vector (which
      // describes a translation) and the SO(3) element to an element of
      // the Euclidean group. So that the configuration space of a
      // RigidObject may be identified with the Euclidean group. This is
      // studies in some detail in my Mathematica file
      // euclideangroup2.nb.

   protected:
```

```
// independent quantities

      Z1 locked_;
         // if !=0, freeStep(dt) will act idle, except of replacing
         // t by t+dt

      R1 r_;
         // smallest value such that a sphere of radius r arround x
         // contains all matter forming the object

      Spc x_;
         // center-of-mass position or otherwise characteristic position

      Vec v_;
         // velocity of point x

      AxVec ori_;
         // 'orientation' in the meaning of 'angular position'.
         // Global rotation which creates the present
         // angular orientation of the object from a reference
         // configuration.
         // From the definition of AxVec one sees that the components of
         // this vector closely related to what normally is called Euler
         // parameters of the rotation ('rotation matrix') under
         // consideration. The agreement would be exact if in the
         // definition of the Euler parameters one would replace the
         // sin-function by the tan-function.

      AxVec omega_;
         // angular velocity of collective object motion arround x.
         // angular velocity gives the time trend in angular position.
         // The velocity vi at position xi then is given by
         // vi=v+omega.cross(xi-x)

      R1 t_;
         // new element, time to which the quantities refer
         // added 2000-12-3
         // Remember: R1 auto-initializes as 0

// dependent quantity

      RefSys rs_;
         // The 'body-fixed' system of reference determined by x and ori

// non-public functions:

      virtual void update(){ rs_=RefSys(x_,ori_);}
         // updating the dependent quantities according to the actual
         // values of the independent ones. Needs not to be called if
         // only velocities (v or omega) were changed
```

```
      void make(R r, const Spc& x, const Vec& v,
         const AxVec& ori, const AxVec& omega);
         // constructing a RigidObject out of the independent quantities

// Now we consider transformations under Group. We need only to consider
// independent quantities, since only those have to be transformed. The
// dependent ones will be obtained via update() from the transformed
// independent quantities. This procedure guarantees the constancy of
// internal geometry independent of the number of transformations acting
// on a system. Here we divide the independent quantities into streamable
// ones i.e. those that can be written on file and retreived from files
// with sufficient accuracy and acceptable effort, and non-streamable
// ones like instances of F<X,Y>.
// In the final classes to be used, non-streamable data will always
// be dependent on streamable ones and thus will get values from calling
// update().
// In intermediate classes like Body, one needs quantities like shape and
// field in order to formulate interaction laws at a point in time where
// a definition of field and shape in terms of streamable data cannot yet
// be available if one wants to hold things sufficiently general. In
// order to make these intermediate classes, like Body, selfconsistent
// one has to define the transformation law also for non-streamable data.

      virtual void transformStreamableData(const Group& g);

      virtual void transformNonStreamableData(const Group& g){g;}
         // there are no nonstreamable data yet
    public:
      typedef RigidObject Type;
      CPM_IO_V
      CPM_NAM_V(RigidObject)

      virtual RigidObject* clone()const{ return new RigidObject(*this);}
         // this definition from allows to make
         // RigidObjects and instances of derived classes into the
         // polymorphic array components of the polymorphic array Vp<>.
         // See cpmp.h. Actually we will need Vp<ExtSys>, with ExtSys
         // derived from RigidObject as template arguments.

      RigidObject(R r=0):r_(r){}
         // Object at rest in the master system of reference

      RigidObject(R r, const Spc& x, const Vec& v, const AxVec& ori,
         const AxVec& omega){ make(r,x,v,ori,omega);}
         // constructing a RigidObject out of the independent quantities

      R get_r()const{ return r_;}
      void set_r(R rf){ r_=rf;}
```

```
Spc get_x()const{ return x_;}
Vec get_v()const{ return v_;}

R get_t()const{ return t_;}
void set_t(R tf){ t_=R1(tf);}

AxVec getOri()const{ return ori_;}
AxVec getOmega()const{ return omega_;}
RefSys getRefSys()const{ return rs_;}

Group frmMasSys()const{ return rs_.frmMasSys();}
   // from master system
   // returns the transformation which creates rs from the master
   // system of reference

Group toMasSys()const{ return rs_.toMasSys();} // to master system
   // returns the transformation which creates the master system of
   // reference from system rs

R relCor(const Spc& p, Z i)const{ return rs_(p)[i];}
   // relative coordinate of p = coordinate of x in reference
   // system rs i=1,2,3 (3 gives always 0 in two dimensional case)

Vec get_v(const Spc& p)const{ return v_+omega_.cross(p-x_);}
   // velocity at an arbitrary point p of the object

virtual void set_v(const Vec& v){ v_=v;}
   // setting linear velocity.
   // virtual since adding data in derived classes askes for their
   // proper handling

virtual void setOmega(const AxVec& omega){ omega_=omega;}
   // setting angular velocity.

virtual void spin(const AxVec& omega){ omega_+=omega;}
   // adding to angular velocity

virtual void boost(const Vec& v){ v_+=v;}
   // adding to linear velocity

virtual void freeStep(R dt);
   // state change during time span dt under the assumption that v
   // and omega remain constant. Notice that in the absence of
   // external forces v is constant but that the absence of torques
   // does not imply that omega is constant unless the tensor of
   // inertia is a multiple of the unit tensor. Actually, we use
   // RigidObject never as dynamical objects for which inertia
   // plays a role. So ther is no tensor of inertia defined for
   // RigidObject. This will change with classes PointPar and
   // ComPar. Notice that
```

```
         //    freeStep(dt)
         // is identical (appart from t+=dt, which belongs to freeStep)
         // to applying operator *=(g) for the following instance of
         // Group: g=(omega.exp(dt),x-x*omega.exp(dt)+v*dt).
         // For the sake of efficiency we do not make use of this.

      virtual void freeMot(R tf){ freeStep(tf-t_);}
         // changes the present state into the state at time tf employing
         // a mode of motion in which velocity and angular velocity are
         // constant.

      void operator*=(const Group& g)
         // has always this form, building blocks are virtual and thus
         // adjust the function so that additional data of derived
         // classes get transformed properly
      {
         transformStreamableData(g);
         transformNonStreamableData(g);
         update();
      }

      void lock(){ locked_.set(1);}
      void unlock(){ locked_.set(0);}

      virtual void adjustOmega(const Spc& xp, const Vec& vp);
         // Adjusts the angular velocity omega such that after calling
         // the present function we have: get_v(xp)==vp. The quantities
         // x and v remain unchanged.

      R disVal(const RigidObject& ro)const;
         // returns the 'the distance' of *this and ro as a fraction to
         // 2r. Thus the value is not limited by 1. The distance is a sum
         // of the distance between the centers, and the distance between
         // body-fixed peripheral points in a configuration in which the
         // centers are shifted back to coincidence.
   };

////////////////////// struct PhyParPrp ////////////////////////////

struct PhyParPrp{ // physical particle properties
      R1 m_;
         // mass
      R1 r_;
         // an interaction radius, in collision of particles
         // with walls which are implemented by classes derived from
         // CpmPhysicsx::Body it acts as a radius of a rigid sphere
         // that starts to feel reaction forces as soon as this sphere
         // overlaps with the body which represents the wall.
         // In addition. r_ is the regularization radius of the
         // point charge and the point mass associated with the particle.
```

```
R1 q_;
   // charge

R1 alphaAdhesion_;
R1 relativeRangeAdhesion_;
R1 E_;
   // E/(1-sigma*sigma); effective Young's modulus of Hertz spheres
R1 mu_;
R1 v0_;
R1 scnr_;
   // Notice that the 'coefficient of normal restitution (cnr)
   // (1 for elastic collisions, <1 for inelastic) defined as a
   // reduction factor for normal velocity, not for kinetic energy
   // (although the latter, the square of  cnr is needed in the
   // computation of forces (our only usage)

R1 U0_,r0_,a_;
   // U0_,r0_,a_ are the three constants defining a
   // inter-particle potential (see attribute forceType)
   // for 'chemical' forces

Z1 forceType_;
   // 0 no chemical interaction
   // 1 Lennard Jones
   // 2 Morse
   // 3 UM
   // 4 Hertz type elastic repulsion

void set_cp(ContactProperties const& cp)
{
   alphaAdhesion_=cp.alphaAdhesion_;
   relativeRangeAdhesion_=cp.relativeRangeAdhesion_;
   E_=cp.E_;
   scnr_=cp.scnr_;
   mu_=cp.mu_;
   v0_=cp.v0_;
}

ContactProperties get_cp()const
{
   ContactProperties res;
   res.alphaAdhesion_=alphaAdhesion_;
   res.relativeRangeAdhesion_=relativeRangeAdhesion_;
   res.E_=E_;
   res.mu_=mu_;
   res.v0_=v0_;
   res.scnr_=scnr_;
   return res;
}
```

```
      typedef PhyParPrp Type;
      CPM_IO
      PhyParPrp(){}
      PhyParPrp(R d, ParticleProperties const& pp)
      {
         set_cp(pp);
            // sets E_,mu_,v0_,scnr_ and the adhesion values
         r_=0.5*d;
         m_=pp.rho*CpmDim3::Spc::vol(r_);
         q_=pp.sigma*CpmDim3::Spc::sur(r_);
         forceType_=4;
      }
      PhyParPrp(R m, R r, R q=0):m_(m),r_(r),q_(q){}
   };

//////////////////////// class PointParC ////////////////////////////////////
// PointPar stands for point particle
// constant quantities associated with an instance of PointPar

struct PointParC{ // constant quantities of point particles

   PhyParPrp p_;
      // a set of material properties which is shared by all particles
      // of family part

   Z1 id_;
      // identification number, allows identification. Usefull
      // especially in parallel programs where a single particle
      // may have to be handled by various processes.

   B idValid_;
      // initialized as false, device to avoid that id can be changed
      // despite the fact that set_id is public
   Z1 color_;
      // artificial attribute for visualization
      // Color(color_) is actually a color according to
      // my Palette. Since Color(0) = Color(INVISIBLE) the default
      // particle should get an anaglyph representation
   typedef PointParC Type;
   CPM_IO
   CPM_NAM(PointParC)
   PointParC(){}
   PointParC(R m, R r, R q=0):p_(m,r,q){}
   explicit PointParC(const PhyParPrp p):p_(p){}
   PointParC(R d, ParticleProperties const& p):p_(d,p){}
   Z get_id()const{ return id_;}
   bool get_idValid()const{ return idValid_;}
   void setCol(Palette  c){color_=Z1(Color::toZ(c));}
   Color getCol()const{ return Color(Z(color_));}
};
```

```
//////////////////////// class PointParV ////////////////////////////////////
// variable quantities associated with an instance of PointPar

struct PointParV{ // variable quantities of point particles
   // there is no angular velocity!
   Z1 owner_;
      // needed for implementing parallelism
   Spc x_;
      // position
   Vec v_;
      // velocity
   Vec vDot_;
      // acceleration, automatically initialized as zero
      // which is what is needed
   R1 t_;
      // time for which data are valid
   typedef PointParV Type;
   CPM_IO
   CPM_NAM(PointParV)
   PointParV(){}
   explicit PointParV(Spc x, Vec v=Vec(), R t=0):x_(x),v_(v),t_(t){}
   Spc get_x()const{ return x_;}
   Vec get_v()const{ return v_;}
   void mir_(Z i)
   {
      if (i>=1 && i<=3){
         v_.mir_(i);
         vDot_.mir_(i);
      }
   }
};

//////////////////////////// class PointPar ////////////////////////////////////

   class PointPar{ // point particles

      // Describes kinematical states (i.e. positions and velocities) of
      // point particles. Since 2006-02-02 there is the acceleration
      // added. This serves as a memory for easy implementation of my
      // new integrator which is second order even for velocity-dependent
      // forces. Not all comments may already reflect this new situation.
      // As far as force laws are concerned, these point particles have
      // a radius (action radius), but they show no inertia of rotation
      // arround their center; actually, ther is no degree of freedom
      // corresponding to such a rotation.

      PointParC cc;
      PointParV cv;
         // notice: there are no dependent data members
```

```
public:
   friend class ComPar;
   typedef PointPar Type;
   CPM_IO
   CPM_NAM(PointPar)

   PointPar(){}

  explicit PointPar(PhyParPrp const& ppp, Spc const& x=Spc(),
     Vec const& v=Vec()): cc(ppp),cv(x,v){}
   // work horse constructor

   PointPar(const PointParC& cc1, const PointParV& cv1):
      cc(cc1),cv(cv1){}
   // obvious

   PointPar(R d, const ParticleProperties& pp):
   cc(d,pp),cv(){}
      // For uniformity with ComPar; the data pp.J
      // pp.eps, pp.fRelax which deal with magnetic and
      // electric dipoles arepassive here, since missing
      // rotational inertia one can't handle the torques on
      // PointPar's that would result from such dipoles.

   PointPar(R d, const ParticleProperties& pp,
      const NuggetData& pd, Z j=1): cc(d,pp),cv(){pd;j;}
   PointPar(R d, const ParticleProperties& pp,
      const DumbbellData& pd, Z j=1): cc(d,pp),cv(){pd;j;}
   PointPar(R d, const ParticleProperties& pp,
      const PairData& pd, Z j=1): cc(d,pp),cv(){pd;j;}
      // these three only for uniformity with ComPar, last argument
      // has no influence

   PointPar(RecordHandler const& rch, Word const& sec, Z j=1);
      // Flexible input which allows to use a rho input if
      // no mass input is there; and correspondingly uses
      // a sigma input if no q input is there. Due to
      // the 'universality' of the data type RecordHandler,
      // it is easy to provide a constructor with the same
      // signature also for ComPar

   R get_m()const{ return cc.p_.m_;}
   R get_r()const{ return cc.p_.r_;}
   R get_q()const{ return cc.p_.q_;}

   Z get_id()const{ return cc.id_;}

   void set_id(Z id1);// a valid id gets never changed unless
     // the id is cancelled explicitely via cancel_id
```

```
void setOwner(Z owner){ cv.owner_=owner;}
Z getOwner()const{ return cv.owner_;}
bool isVal()const{ return cc.idValid_;}
void cancel_id(){ cc.idValid_=false;}
bool idn(PointPar const& pa)const{
   return (isVal()&&pa.isVal()&&(get_id()==pa.get_id())) ?
   true : false;}

Spc get_x()const{ return cv.x_;}
Vec get_v()const{ return cv.v_;}
PhyParPrp get_p()const{ return cc.p_;}

PointParC getC()const{ return cc;}
PointParV getV()const{ return cv;}
void set(const PointParC& cc1, const PointParV& cv1)
{cc=cc1; cv=cv1;}

void set_m(R m){cc.p_.m_=m;}
void set_r(R r){cc.p_.r_=r;}
void set_m_r(R m, R r, const NuggetData& pd=NuggetData())
   // last argument for uniformity with ComPar, here it is
   // passive
{pd;cc.p_.m_=m;cc.p_.r_=r;}

void set_q(R q){cc.p_.q_=q;}
void rev_q(){cc.p_.q_*=-1;}
void invertCharge(){cc.p_.q_*=-1; }
   // changing the sign of the charges
void set_E(R E){cc.p_.E_=E;}
void setCofOfNorRes(R cnr){cc.p_.scnr_=cnr*cnr;}

void set_cp(ContactProperties const& cp){ cc.p_.set_cp(cp);}
ContactProperties get_cp()const{ return cc.p_.get_cp();}

R getEeff()const{ return cc.p_.E_;}

void setPot(R U0, R r0, R a, R cnr, Z forceType)
{
   cc.p_.U0_=U0; cc.p_.r0_=r0; cc.p_.a_=a;
   cc.p_.scnr_=cnr*cnr; cc.p_.forceType_=forceType;
}

void setForType(Z forceType){cc.p_.forceType_=forceType;}

void set_x(const Spc& x){ cv.x_=x;}
void set_v(const Vec& v){ cv.v_=v;}
void set_p(const PhyParPrp& p){ cc.p_=p;}
void setCol(Palette c){cc.setCol(c);}
```

```
R get_t()const{ return cv.t_;}
Color getCol()const{return cc.getCol();}
void set_t(R tf){ cv.t_=tf;}

PointCharge getPointCharge()const
{ return PointCharge(cc.p_.q_,cv.x_,SysControl::singRange);}

PointMass getPointMass()const
  { return PointMass(cc.p_.m_,cv.x_,SysControl::singRange);}

void boost(const Vec& v){ cv.v_+=v;}

void ranBoost(Vec const& v, R dv, R angleRange, Z j)
{
   Vec vj=v.randomize(dv,Angle(angleRange,DEG),j);
   boost(vj);
}

void spin(const AxVec& omega_){omega_;}
   // does nothing; for uniformity with ComPar

void reflect(const Plane& pl)
   {cv.x_=pl.reflect(cv.x_); cv.v_=pl.reflect(cv.v_);}

void operator *=(const Group& a){cv.x_*=a; cv.v_*=a; cv.vDot_*=a;}
   // action of the basic symmetry group on a particle. Notice that
   // x and v belong to different classes (Spc and Vec,
   // respectively) the group element a does different things to
   // them.

void mir_(Z i); // mirror (reflection)

void ranRot(R angleRange, Z j){}
   // Random rotation arround the center of mass which is trivial
   // here.
   // Definition for uniformity with ComPar

void operator *=(R fac);
   // scaling a point particle such that all linear dimensions
   // get multiplied by fac and mass by fac^3 (charge remains
   // constant since quantized in nature)

R dist(const PointPar& pa)const
   { return (cv.x_-pa.cv.x_).absVal();}
   // returns the distance of centers

R disVal(const PointPar& pa)const
{ return dist(pa)/(cc.p_.r_+pa.cc.p_.r_);}
   // returns the 'the distance' of *this and pa divided by the sum
   // of the radii of *this and pa. Thus the value is not limited by
```

```
   // 1.

R gap(const PointPar& pa)const
{ return dist(pa)-(cc.p_.r_+pa.cc.p_.r_);}
   // if the spheres occupied by the particles don't overlap, this
   // is their distance in the sense of geometry (distance of
   // closest point pair) If they overlap, the return value is
   // negative and gives (appart from the sign) the overlap
   // distance. Even if a small particle occupies the interior
   // of a large particle it is not too unreasonable a measure
   // of which the repulsive force should be a increasing function.
   // Notice that the minimum of the function is -r-pa.r

R corGap(const PointPar& pa)const{ return gap(pa);}
   // coarse gap

X2<R,Vec> gapAndDirection(const PointPar& pa)const
   // returns gap and the unit vector pointing from *this to pa
   // (center to center)
{
   X2<R,Rdim> pp3=(pa.cv.x_-cv.x_).polDec();
   return X2<R,Vec>(pp3.c1()-cc.p_.r_-pa.cc.p_.r_,pp3.c2());
}

 Spc frcCen(const PointPar& pa)const;
   // returns a point in which contact forces can reasonably
   // assumed to act. This point equals the point p1.cen(p2)
   // mentioned in function clsApp.

void freeStep(R dt){ cv.x_+=dt*cv.v_; cv.t_+=dt;}
   // free motion

void freeMot(R tf){ freeStep(tf-cv.t_);}
   // free motion. t=tf after call

void phase1(R dt){ if (SysControl::accMem) cv.v_+=cv.vDot_*dt;}
   // named according to Section 7 in GranMat publication

void intAct(R dt, const Force& fs,
   const FieldSpace& fval=FieldSpace()); // interaction
// Last  argument not used

void reg(void);
   // regularization of the motion by reducing the velocity
   // if this is larger than SysControl::vCrit
   // no action for SysControl::vCrit

Force forces( const PointPar& pp )const;
   // returns the force acting on the particle *this
   // resulting from the presence of a particle pp.
```

```
R pot( const PointPar& pp )const;
   // returns the potential of forces above

Force forceContact(PointPar const& p,
   Vec const& v, Vec const& vp)const;
   // force which particle *this feels as a result of the presence
   // of p. Contact interaction forces are treated as originating
   // at the points of closest approach (see function clsApp) and
   // acting on those points. Friction forces depend on velocities,
   // for which we have to insert the velocities of the points
   // of closest approach. These velocities are given by the last
   // two arguments of the function. If among the data of PointPar
   // there would be an angular velocity, then we would express v
   // and vp in terms of PointPar-data; so we have to provide them
   // as arguments.
   // This  friction scheme also damps relative rotation
   // of spherical particles so that Clust's behave dynamically
   // normal if they are not assymmetrical and thus may be made of
   // a single constituent. This helps to improve speed.
   // Contact force laws all are coded in function
   // forceFrictionAdhesionElastic(...)

R potElastic(PointPar const& p)const;
   // potential of the elasic forces contributing to forceContact

Force forces( const VectorField& f)const;
   // returns the force acting on the particle *this
   // resulting from the presence of electric, magnetic, and
   // gravitational fields.

Force forces( const FieldSpace& f)const;
   // returns the force acting on the particle *this
   // resulting from the presence of homogeneous electric,
   // magnetic,
   // and gravitational fields.

R pot( const FieldSpace& f)const;
   // potential energy pf particle *this resulting from the
   // presence of homogeneous electric, magnetic, and gravitational
   // fields.

Force forces( const RigidObject& f)const;

R kinEnr()const{ return cc.p_.m_*0.5*(cv.v_|cv.v_);}
   // kinetic energy

ScalarField getU()const
   // potential associated with the types of interaction to be
   // considered
```

```
      {
         if (SysControl::potentialToShow==3)
            return getPointMass().fieldU();
         else if (SysControl::potentialToShow==2)
            return ScalarField();
         else
            return getPointCharge().fieldU();
      }

   FieldE getE()const{ return getPointCharge().field();}
   FieldBH getBH()const{ return getPointCharge().fieldBH(cv.v_);}
   FieldG getG()const{ return getPointMass().field();}

   VectorField getField(bool noGravity=true)const
   {
      if (noGravity) return VectorField(getE(),getBH(),FieldG());
      else return VectorField(getE(),getBH(),getG());
   }

   Vec mom()const{ return cv.v_*cc.p_.m_;}
      // momentum

   AxVec angMom(Spc const& p)const;
      // returns the total (mechanical) angular momentum
      // of *this relative to the point p in space

   void mark(Graph& g, const Color& color)const;
      // Draws a filled disk with particle's radius according to
      // the object space metric of g. Here color acts a an instance
      // of the base class rgb of Color, so that Color(INVISIBLE)
      // acts as rgb(0,0,0) which is Color(BLACK)

   void mark(Graph& g)const{ Color col=getCol(); mark(g,col);}
      // same as previous function but uses the own color for drawing

   void mark(const Camera& sc, const Color& color)const;
      // If SysControl::coloredParticles, the representation is by
      // a filled circular disk, without a velocity indicator.
      // if SysControl::howToMarkParticles==1 a wire frame
      // representation using color is drawn. A velocity indication is
      // also  given as a line startingfrom the center.

   void mark(const Camera& sc)const
      { Color col=getCol(); mark(sc,col) ;}
      // same as previous function but uses the own color for drawing

  R sclVal(const Word& w)const;
      // scalar value
      // returns a scalar value designated by w, where
      // the following cases for w are implemented
```

```
   // "mass", "velocity", "kinetic_energy", "charge".
   // In cases where w indicates a vector quantity,
   // the actual value from this function is the
   // absolute value of the vector quantity.

SourceOfVectorField source()const;
   // unified treatment of field sources
   // of particle *this

void stop(R red=0, bool selective=false )
   {
      if (selective){
         cv.v_[3]*=red;
         cv.vDot_[3]*=red;
      }
      else{
         cv.v_*=red;
         cv.vDot_*=red;
      }
   }
   // brings *this to rest (state with all velocities equal to
   // zero) or reduces all velocities of *this by the factor red.

bool isMag()const{ return false;}
   // ' is magnetic'
   // for uniformity with ComPar

void deElectrize(){}
   // setting the electric moment to zero
   // for uniformity with ComPar

void removeMagDipole(){}
   // setting the magnetic moment to zero
   // for uniformity with ComPar

static B isVal(const V<PointPar>& vp);
   // returns true if there are no reasons to
   // reject data vp
   // Implementation based on calling
   // static B isVal(const V<PointParV>& vv)

static B isVal(const V<PointParV>& vv);
   // returns true if there are no reasons to
   // reject data vv
   // Present implementation based on the velocity distribution
static void showProperty(const Word& theme, const V<PointPar>& vp,
   const Camera& ca, const R_Vector& parameters);

static Force forceFrictionAdhesionElastic( R gap, const Vec& n12,
   R r1, R r2, const Vec& v1, const Vec& v2,
```

```
   const ContactProperties& cp1, const ContactProperties& cp2 );
// a new implementation for which the following description of
// friction forces is not used. Hysteresis of normal force instead
// and conventional friction law for tangential force.
// The idea to use hysteresis of normal forces for modeling
// friction was told to me by Dr. Bernhard Eberhardt at the
// university of Tübingen in 1998.
// n12 is the unit vector directed from 1 to 2

static R elasticPotential( R gap, const Vec& n12,
   R r1, R r2, ContactProperties const& cp1,
   ContactProperties const& cp2 );
// returns the potential of the elastic contribution to force
// given by the previous function

static R collisionTime(R r1, R r2, R m1, R m2, R v,
   const ContactProperties& cp1, const ContactProperties& cp2);
   // This is the time during which two centrally colliding elastic
   // sheres are in contact as computed first by H. Hertz 1881
   // (Source: Landau, Lifschitz: Lehrbuch der Theoretischen
   // Physik,
   // Band 7, 5. Auflage, Akademie Verlag 1983 § 9 p. 39)
   // Here v is the abs-value of relative velocity of the
   // spheres.

   // We consider two objects, distinguished by index 1 and 2.
   // Each has a mass, a velocity, a radius, and an instance of
   // ContactProperties.
   // The location of the objects is characterized (not
   // completely!) by the geometry of their 'region of closest
   // approach' ('contact region'). This is a numerical value of
   // the 'gap' (positive if there is really a gap and no overlap;
   // so -gap is 'the overlap') and a unit vector n12 from object 1
   // to object 2.
   // These quantities obviously have a well defined meaning if
   // both objects are spherical particles. In this case v1 and v2
   // are not the velocities of the particle centers but the
   // velocities of the points of closest approach (cf. comment to
   // function
   //  Force forceContact(const SphPar& p,
   //     const Vec& v, const Vec& vp)const; )
   // One of the objects is allowed to grow to a
   // half-space (radius tends to infinity). (For two half-spaces
   // the situation can only be applied if their limiting planes
   // are parallel.)
   // The return value of the function is the force that object 1
   // feels from object 2. It is a sum of three contributions:
   //    1. relative friction
   //    2. adhesion
   //    3. elastic repulsion
```

```
         // which will be described now for the most important case that
         // the two objects are of type SphPar.
   };

//////////////////////// class ComParC ////////////////////////////////
// ComPar stands for composite particle
// constant quantities associated with an instance of ComPar

struct ComParC{ // constant quantities of composite particles
   typedef ComParC Type;
   ComParC(){}
   CPM_IO
   CPM_NAM(ComParC)
   R1 r;
      // smallest value such that a sphere of radius r arround x
      // contains all particles of the cluster
      // (thus r= sup{ |xr[i]|+ part[i].r : i=1,...n}
   R1 volume;
      // the volume of the body
      // This is assumed to define also the surface of the body
      // as the surface of the corresponding sphere
   V<R> ri;
      // list of the radii of the constituent particles
   V<Vec> xr0;
      // list of the particle centers relative to center of mass in
      // the 'body-fixed frame'. These are needed in order not to
      // allow the bodies geometry vary with accumulating numerical
      // errors
   InTens I0;
      // Tensor of inertia ( with respect to x and the body fixed
      // axes)
   Vec ej0;
      // a unit vector the components of which refer to the body fixed
      // axes, which gives the direction of the magnetic moment. If
      // the magnetic moment is zero, ej0 will neverthelss be
      // initialized in every constructor

   ParticleProperties prop;
      // a set of material properties which is shared by all particles
      // of family part

   Z1 id;
      // identification number, allows identification. Usefull
      // especially in parallel programs where a single particle
      // may have to be handled by various processes.

   B idValid;
      // initialized as false, device to avoid that id can be changed
      // despite the fact that set_id is public
```

```
   bool isToner()const{ return !(prop.J>0);}
   R getVolume()const{ return volume;}
   R get_m()const{ return volume*prop.rho;}
   R get_q()const{ return pow(36*Pi*volume*volume,R(1.)/3)*prop.sigma;}
      // correct even if looks strange
   Z get_id()const{ return id;}
   bool get_idValid()const{ return idValid;}
   ContactProperties get_cp()const{ return (ContactProperties)prop;}
   void mir_(Z i)
   {
      if (i>=1 && i<=3){
         I0.mir_(i);
         ej0.mir_(i);
         for (Z k=1;k<=xr0.dim();++k) xr0[k].mir_(i);
      }
   }

};

//////////////////////// class ComParV ////////////////////////////////
// variable quantities associated with an instance of ComPar

struct ComParV{// variable quantities of composite particles
   typedef ComParV Type;
   ComParV(){}
   CPM_IO
   CPM_NAM(ComParV)

   Z1 owner;

   Spc x;
      // center-of-mass position

   Vec v;
      // velocity of center-of-mass

   Vec vDot;
      // acceleration of center-of-mass

   AxVec ori;
      // ori stands for orientation
      // global rotation which creates the present orientation of the
      // cluster relative to a reference configuration (at creation or
      // for the initial state of a dynamical process)

   AxVec omega;
      // angular Velocity of collective cluster motion arround center
      // of mass. The velocities v[i] of the individual particles
      // then are given by v[i]=v+omega.cross(xr[i])
```

```
   AxVec omgDot;
      // Time derivative of omega, angular acceleration

   ElcDipole qd;
      // The dipole is assumed to be located at x, thus qd.x=x. This
      // has the efficiency advantage, that the torque on qd directly
      // translates into a torque on *this.
      // addition 2001-4-1

   R1 t;
      // time, see RigidObject::t (addition 2000-12-12 after having
      // re-united ComPar and ComPar2
   Spc get_x()const{ return x;}
   Vec get_v()const{ return v;}

   void mir_(Z i)
   {
      if (i>=1 && i<=3){
         v.mir_(i);
         vDot.mir_(i);
         ori.mir_(i);
         omega.mir_(i);
         omgDot.mir_(i);
         //qd.mir_(i); not yet defined
      }
   }
};

//////////////////////////// class ComPar ////////////////////////////
// ComPar stands for composite particle
/*
Design consideration concerning representation of irregularly shaped
bodies as clusters of (potentially overlapping) spherical particles
which are rigidly connected

It is clear that any shape can be modelled by composing spheres (of
various sizes) if we can use sufficiently many for this purpose. Since we
have to be concerned about computational efficiency, it is important to
design the scheme such that already a few spheres make up shapes that
behave reasonable (for instance which simulate rather smooth surfaces
without deep holes which would restrict relative mobility of the
clusters too much. For this purpose (pointed out by Thomas Dera) it is
reasonable to let the spheres overlap (as we do here!).

The parameters which describe the physical properties of the surface
(e.g. Hamaker constant) or bulk interaction properties like elastic
constants and constants of dielectricity, are chosen here to have the
same values for all particles belonging to one cluster. This is mainly
in order not to have too many data associated with a cluster.
If this concern should turn out to be only a minor one, one should
```

```
integrate all physical parameters in the description of PointPar.

Also for reasons of efficiency, I decided to represent the electrical and
magnetic structure of a ComPar by a single point charge and a single
magnetic dipole. It would be not difficult, however, to endow each
constituent particle with its own point charge and magnetic moment. This
would allow to model toner particles with non-uniform charge distribution
as discussed for instance in
   James Q. Feng and Dan A. Hays: Theory of Electric Field Detachment of
   Charged Toner Particles in Electrophotography
   Journal of Imaging Science and Technology, Vol. 44/#1 January/February
   2000 p. 19-25

2000-6-15: Now SysControl::singleCharge controls whether a single charge
for a ComPar is used or whether each constituent particle has its own
charge.
*/

   class ComPar{ // composite particles
      // could be derived from RigidObject. Actually it is not
      // in order to avoid any degradation of efficiency.
      // The class implements rigid body dynamic via a propriatory(?)
      // version of the explicit midpoint method. The new part is the
      // parallelism in treating 'spatial position' and 'angular
      // position'. We associate (x,v) with space position and
      // (ori,omega) with angular position.
      // The unifying point of view is that x and ori belong to Lie
      // groups and v and omega belong to the Lie algebras of the
      // corresponding Lie groups. (Clearly, we may unify the two groups
      // under consideration into a single group, the Euclidean group).
      // Operations, which in mechanics textbooks are done in real-valued
      // 'generalized coordinates' are done here directly in the
      // operations defined for Lie groups and Lie algebras. This adds
      // both to clarity of structure and efficiency of the numerical
      // implementation. Some of the concepts involved are explained in
      // class RigidObject.
      // Since 2006-02-02 there is the rates of change for velocity and
      // angular velocity  added. These serve as a memory for easy
      // implementation of my new integrator which is second order even
      // for velocity-dependent forces. Not all comments may already
      // reflect this new situation.

// independent data (only those have to be written on file,
// dependent ones will be reconstructed by the function updateInternals.

      ComParC cc;
      // constant data during program run

      ComParV cv;
      // data that change with state
```

```
// dependent data; notice that we do not use one of the dependent data
// members of RigidObject: RefSys rs. This is for saving memory space
// since we typically will have to handle many instances of type ComPar.

      Z1 n;
         // number of particles in the cluster

      R1 m;
         // total mass of the cluster

      R1 mInv;
         // inverse of m

      V<Vec> xr;
         // list of the particle positions relative to center of mass

      InTens I;
         // Tensor of inertia ( with respect to x and the system of
         // master axes)
      InTens Iinv;
         // inverse of I
      InTens Iinv0;
         // inverse of I0
      V<PointPar> part;
         // array of particles i with x[i], v[i], not referring to the
         // center of mass but to reference space (master space).
         // Data for masses come from here, values for position and
         // velocity are derived from x, v, omega, and xr[i]
      PointCharge q;
         // The charge is assumed to be located at x, thus q.x=x. Having
         // a position associated with q itself is redundant in the
         // present setting. If one wants to incorporate non-trivial
         // charge distributions, one would have to have an array of
         // charge points ( i.e. V<PointCharge> ) in close analogy
         // to array part.

      MagDipole j;
         // The dipole is assumed to be located at x, thus j.x=x. This
         // has the efficiency advantage, that the torque on j directly
         // translates into a torque on *this.

      B isMag_;
         // true if the magnetic moment is different from zero

      R1 alpha;
         // polarizability

// functions
```

```
void make(R r, const V<Vec>& xRel, const V<R>& radii,
   const ParticleProperties& p, const Vec& ej, const Vec& v_=Vec(),
   const AxVec& omega_=AxVec());
   // This function may be called for any instance of ComPar
   // (e.g. the default one) and will initialize explicitely all
   // data members.
   // The array of xRel gives the positions of the centers of solid
   // spheres relative to the origin of the master system of
   // reference.
   // The array radii gives the corresponding radii of these
   // spheres.
   // The cluster to be formed is the union of these spheres.
   // This quantities are understood as referring to an arbitrary
   // common unit of length. The geometry given such up to a
   // scaling transformation will be scaled such that the volume of
   // the set union of these spheres (which, by definition, is the
   // volume of the cluster, is the volume of a sphere of  radius r.
   // This volume will be filled with matter such that the density
   // is p.rho.
   // Volume, center-of-mass, tensor of inertia of this all will be
   // found by stochasical integration.
   // The surface of the cluster comes into play in order to allow
   // the prescription of its total charge in the form
   // surface*p.sigma. Since the surface charge built up by
   // tribo-charging is far from a uniform distribution, it is not
   // clear that the exact geometrical surface (which would not be
   // easy to calculate for the set union of overlapping spheres)
   // is the best quantity to enter the formula given above.
   // To avoid complications of doubtful return we simply define
   // the 'active surface' of the cluster as 4*pi*r^2, the
   // surface of a sphere that has the same volume as the cluster.
   // The (Coulomb) magnetic moment (SI unit is Tesla) of the
   // cluster is prescribed as volume*J as absolute value and the
   // direction of the magnetic moment is given by ej.
   // Finally we shift the whole object such that the center of
   // mass coincides with the origin and set the center-of-mass
   // velocity equal to v and the angular velocity equal to omega.

 void updateInternals();
   // x, v, ori, omega are assumed to have valid values.
   // All quanties (except those that are constant anyway) get
   // changed to their actual value. The non-constant dependent
   // quantities get re-built from body-frame data (name ending in
   // '0') so that the geometry remains unaffected by numerical
   // errors resulting from composing many rotations.
   // This turned out to be very important and was the motivation
   // for introducing the body-frame data in 99-10-2.
   // Since this operation is called extremely often, it should be
   // very efficient. Therefore, in arrays of quantities (which
   // always are fixed in number) we only change the components
```

```
        // and don't assign the whole array. This causes some
        // code doublication in the next function.

      void createAllDependentData();
        // This assigns to all dependent data the proper value
        // (assuming that all independent data have proper value) not
        // making any assumption on the present values of these
        // dependent data. In particular, vectors get assigned and not
        // only their components.

public:
   typedef ComPar Type;
   virtual ComPar* clone()const{ return new ComPar(*this);}
   CPM_IO_V
   CPM_ORDER
   CPM_NAM(ComPar)
   ComPar(){}
      // does nothing useful

   ComPar(RecordHandler const& rch, Word const& sec, Z j=1);
      // Flexible input which allows to define a ComPar
      // from the same rch and the same sec for which also
      // PointPar(rch,sec) makes sense.
      // Letting j vary produces different instances based on different
      // random decisions

   ComPar(R d, const ParticleProperties& pp, const NuggetData& pd,
      Z j=1);
   // stochastically shaped cluster of pd.np (or less in exceptional
   // cases) constituents, which are spherical particles of type
   // PointPar.
   // The characteristics of this cluster are as follows:
   // 1. All constituents have the same radius
   // 2. The volume v of the cluster, which by definition is the volume
   //    of the set-union of the constituents, equals (4/3)*Pi*r^3 where
   //    r:=d/2.
   // 3. The centers of all constituents can be represented by instances
   //    of Spc and thus lie on a plane in the two-dimensional version
   //    of the program (i.e. for using namespace CpmDim2).
   //    Moreover, the centers of the constituents lie on a circle in the
   //    2-dimensional version(=:2D) and on the surface of a sphere in
   //    the 3-dimensional version (=:3D).
   //    Let the radius of this sphere/circle be r0. For np points
   //    populating such a manifold 'rather uniformly' a natural distance
   //    to the next neighbor is 3*r0/np in 2D and 2*r0/sqrt(np) in 3D.
   //    Let us denote this value by disMean. The factors 3 and 2 in this
   //    definition are a bit arbitrary and also sub-optimal. I did not
   //    modify them for compatibility with old files.
   // 4. The distance between the centers of any two constituents is
   //    greater or equal to isotropyFac*disMean. If isotropyFac is
```

```
//    sufficiently large, it becomes impossible to place np points on
//    a sphere/circle in such a manner that this distance requirement
//    is met. If the trialMax positions from the random generator were
//    tested, there will no new ones and the cluster will formed out
//    of the valid points placed so far. These then will be less than
//    np. If isotropyFac is considerably lower than 1, the
// constituents
//    are allowed to populate their available space rather
//    non-uniformely, resulting in a strongly non-spherical shape of
//    the cluster.
// 5. The radius of each constituent is overlapFac*disMean.
//    overlapFac should be >=1 for getting a cluster that has no
//    holes it its surface.
//
// 6. The mass of the cluster is v*pp.rho, the magnetic moment v*pp.J
//    (see 2. for v).
// 7. The charge of the cluster is (4*Pi*r^2)*pp.sigma
//    (see 2. for r).
// 8. The center-of-mass position of the cluster coincides with the
//    origin of the master system of reference.
// 9. All constituent particles have the same ContactProperties
//    attribute cp=(ContactProperties)pp
//
// The volume and the tensor of inertia are calculated rather
// efficiently and accurately by stochastic integration ('Monte Carlo
// Method'). For the true surface of a set-union of spheres no
// efficient algorithm is at hand, so I use the surface of a sphere
// as a surrogate. See the comment to function ComPar::make(...) for
// details.

ComPar(R d, const ParticleProperties& pp, const DumbbellData& pd,
   Z j=1);
 // Regular elongated object made of three spheres.
 // d as in the previous function.
 // Here we have three spheres the centers of which lie on a straight
 // line in an equi-spaced manner. Let D0 be the distance from the
 // center of the central sphere to any of the two spheres in
 // end-position. Let Rc be the radius of the central sphere. The two
 // end-spheres have the same radius Re.
 // Then the meaning of the arguments rc and re is rc=Rc/D0 and
 // re=Re/D0. In order for the spheres to overlap we need rc+re>1.
 // For re>rc we have a dumbbell and for re<rc we have a kind of a
 // base ball.
 // If the particles are treated as magnetic, the magnetic moment
 // forms an angle phi (in rad) with the axis of rotational symmetry
 // of the dumbbell.
 // Dumbbells were proposed 2000-5-5 by Mike Regan as a model for
 // experimental Kodak carrier material.

ComPar(R d, const ParticleProperties& pp, const PairData& pd, Z j=1);
```

```
// Describes elongated objects made of two spheres.
// Rather trivial spin-off from Dumbbell, might be useful as the
// simplest non-spherical particle

static X2<V<ComPar>,R> makePair(R dC, R dT, Z npC, Z npT,
        ParticleProperties ppC, ParticleProperties ppT,
        R overlapFacC=2, R overlapFacT=2,
        R isotropyFacC=1, R isotropyFacT=1,
        R facSafty=1.05, const Vec& dir=Vec(0,1));
     // This function does nearly the same than the previous one.
     // It creates only one carrier c and one toner t, determines the
     // charges from the surface of the particles and ppC.sigma and
     // ppT.sigma, and allows to set the direction of the vector
     // which points from the carrier center to the toner center
     // (center = center-of-mass)

  V<PointPar> getConstituents()const{ return part;}
     // getting the family of constituent particles in order to let
     // external systems have all the data they need for interaction
     // Notice that the constituents carry the same ContactProperties
     // as the whole object.

  ForTor forces( const ComPar& cl )const;
     // returns the force and torque acting on the cluster *this
     // resulting from the presence of a cluster cl. Notice that
     // forces on the constituent particles are the primary mediators
     // of interaction and that the combination to a net force and a
     // torque is completely within the responsibility of class
     // ComPar.
     // If get_id()==cl.get_id() x  the returned value is ForTor().
     // So in summing over particle pairs it is not necessary to
     // exclude the case i=j in most contexts.
     // In the present version, only the contact forces are mediated
     // by the constituent particles, wheras magnetic and electric
     // forces originate from and are felt by pointcharges,
     // electric moments, and magnetic moments that are located in
     // the center-of-mass of the cluster. The electric moments were
     // added in 2001-4-1 and are set proportional to a polarizabilty
     // and an external electric field, where the polarizability
     // is proportional to the volume of the ComPar.(see
     // void ComPar::createAllDependentData()):
     //    R epsr=prop.eps;
     //    alpha=volume*3*CpmPhysics::eps0_PHYS*(epsr-1)/(epsr+2);
     // In the time stepping algorithm, we separate the formation
     // of an induced electric moment and the generation of forces
     // (see ForTor ComPar::forces( const ComPar& cl )const) in a
     // natural manner. Notice also that
     //
     //    FieldE getE()const{ return q.field()+ej.field();}
     //
```

```
   // lets the electric moment contribute to field generation.
   // Since May 2000 there is a option (SysControl::singleCharge=0)
   // to replace the single point charge mentioned above by point
   // charges in the centers of the constituent particles.
   // The contact part of the force (i.e. friction, adhesion,
   // elastic repulsion) is implemented such that contact
   // forces originate from and act on the points of closest
   // approach between constituent particles and not the centers
   // of the constituent particles (as was the case in early
   // versions). This makes clusters consisting of a single
   // particle behave similar to larger clusters.
   // If also the friction forces are treated as acting directly
   // on the particle center, they are not able to prevent
   // densely packed one-particle clusters from spinning with
   // in arbitrary angular velocity (varying from particle to
   // particle). If the forces act on the periphery of the
   // particles only isolateted pairs can practice 'contact
   // spinning' and only if they arrange their angular velocities
   // opposit to each other. This more complex mode of motion is
   // much less likely to become excited by numerical noise.

R pot( const ComPar& cl )const;
   // returns the potential of the electric and gravitational
   // forces for the simple model that all charge and all mass is
   // collected in the point cv.x (center-of-mass). Since March
   // 2005 there is added also the potential energy of the elastic
   // forces.  Notice that the definition of the dynamics makes no
   // use of pot. The only reason for having pot is to demonstrate
   // conservation of energy.

R potElastic( const ComPar& cl )const;
   // returns potential energy of the elastic forces.

ForTor forces( const VectorField& f)const;
   // returns the force and torque acting on the cluster *this
   // resulting from the presence of electric, magnetic, and
   // gravitational fields.

ForTor forces( const FieldSpace& f)const;
   // returns the force and torque acting on the cluster *this
   // resulting from the presence of homogeneous electric,
   // magnetic, and gravitational fields.

R pot( const FieldSpace& f)const;
   // potential energy pf particle *this resulting from the
   // presence of homogeneous electric, magnetic, and gravitational
   // fields.

ForTor forces( const RigidObject& ro)const;
   // returns the force and torque acting on the cluster *this
```

```
         // resulting from the presence of a resistive medium such as air
         // which is at rest relative to ro. Such a resistive force may
         // either be used to model damping by motion in air or it may be
         // used as an artificial means to avoid numerical problems in
         // case that too large time steps led to too violent motion.
         // The present implementation is as follows: The damping medium
         // has dynamical viscosity  SysControl::eta and exerts damping
         // force and torque according to Stokes damping force formula
         // for a sphere of radius r: -6*pi*eta*v*r and damping torque
         // -8*pi*eta*r^3*omega. See L.D.Landau, E.M.Lifschitz: Lehrbuch
         // der Theoretischen Physik, Band VI, Hydrodynamik, S. 75
         // (20,17) and S. 77.

// time stepping and components for the explicit midpoint method

      void freeStep(R dt);
         // Integrator for free motion. Does not assume constant
         // angular velocity as RigidObject did

      void phase1(R dt)
      {
         if (SysControl::accMem)
         cv.v+=cv.vDot*dt; cv.omega+=cv.omgDot*dt;
      }

      void intAct(R dt, ForTor const& fs,
         FieldSpace const& fval=FieldSpace());
         // change of v and omega and electric moment during dt.
         // fs is the whole force-torque on the particle, including
         // contributions from electric moments. fval should be
         // chosen as the present value of the outside field at point x.
         // It is needed to update the value of the electric dipole
         // moment qd. Presently only the electric component of fval
         // matters since the magnetic moment is treated as constant.
         // (ideal permanent magnets).

      R get_t()const{ return cv.t;}
      void set_t(R tf){ cv.t=tf;}

      void freeMot(R tf){ freeStep(tf-cv.t);}
      // t=tf after call

// transformations

      void operator *=(const Group& g);
         // active transformation acting on a ComPar

      void mir_(Z i); // mirror (reflection)

      void ranRot(R angleRange, Z j);
```

```
   // random rotation
   // Rotates *this arround the center of mass by a rotation angle
   // in [0,angleRange) (angleRange in degrees) and arbitrary
   // rotation axis. Angle and axis depend randomly on j.

void operator *=(R fac);
   // scaling a cluster such that rho, sigma, J (see constructor)
   // have constant values and linear dimensions are magnified by
   // fac;

void boost(const Vec& v)
{ cv.v+=v; for (Z i=1;i<=n;i++) part[i].boost(v);}
   // active transformation which has on data the same influence as
   // changing the sytem of reference into a uniformely moving one

void ranBoost(Vec const& v, R dv, R angleRange, Z j)
{
   Vec vj=v.randomize(dv,Angle(angleRange,DEG),j);
   boost(vj);
}

void spin(const AxVec& omega_);
   // add opmega_ to the angular velocity omega and adjust particle
   // velocities accordingly

void stop(R red=0, bool selective=false )
   {
      if (selective){
         cv.v[3]*=red;
         cv.vDot[3]*=red;
      }
      else{ cv.v*=red;cv.omega*=red;cv.vDot*=red;cv.omgDot*=red;}
      updateInternals();
   }
   // brings *this to rest (state with all velocities equal to
   // zero) or reduces all velocities of *this by the factor red.

void reg(void);
   // see PointPar::reg()

bool isMag()const{ return isMag_;}

void deElectrize(){ cv.qd.setMoment(Vec());}
   // setting the electric moment to zero

void removeMagDipole(){ cc.prop.J=0.; isMag_=false; j=MagDipole();}
   // setting the magnetic moment to zero. It is important to make
   // the change in the 'genes' (cc in this case) too, so that
   // the function createAllDependentData() will not set isMag_
   // back to true.
```

```
void reflect(const Plane& pl)
{cv.x=pl.reflect(cv.x); cv.v=pl.reflect(cv.v);updateInternals();}
   // reflection at plane. Notice that reflection acts trivially on
   // axial vectors ori and omega

void invertCharge(){ q.invertCharge();}
   // changing the sign of the charges

void set_cp(const ContactProperties&);
   // setting the ContactProperties subset of the
   // ParticleProperties

void set_pp(const ParticleProperties&);
   // setting the ContactProperties subset of the
   // ParticleProperties

void set_m_r(R m, R r, const NuggetData& pd=NuggetData());
   // Changes *this into a ComPar with given mass, geometrical
   // radius (not radius of a sphere of equal volume!), and shape.
   // for uniformity with PointPar, this allows to set also the
   // shape via the fourth argument (in class PointPar this
   // argument is passive).

void set_E(R E);
   // setting effective E

void setCofOfNorRes(R cnr);//{ cc.prop.scnr_=cnr*cnr;}
   // setting the coefficient of normal restitution

void setCol(Palette c){cc.prop.color=c;}
   // setting the visualization color

void setForType(Z i){i;}
   // presently there are no different force types for class ComPar
   // so this is trivially implemented

Color getCol()const{return Color(cc.prop.color);}
   // getting the visualization color

void set_x(const Spc& sp){ operator*=(Group(sp-get_x()));}

void setDirJ(const Vec& v);
   // rotates the particle *this such that the magnetic
   // dipole moment and v are parallel

Vec getDirJ()const{ return j.get_j();}
   // returns the magnetic dipole moment as a vector

ParticleProperties get_pp()const{ return cc.prop;}
```

```
   // since ParticleProperties can be get and set, the internal
   // structure of ParticleProperties needs not tho be known
   // to the class ComPar

bool multiplyParameter(Word const& w, R fac);
   // multiplies a parameter indicated by argument w by factor fac.
   // More precisely:
   // If w does not belong to the list:
   // "charge", "sigma, "alphaAdhesion", "relativeRangeAdhesion",
   // "effectiveE", "scnr", "J", "rho"
   // we do nothing except of returning 'false'.
   // If w belongs to the previous list, we multipliy the parameter
   // denoted w by factor fac and adjust all dependent parameters
   // accordingly.
   // For fac==1, this is done very efficiently by doing nothing.

R kinEnr()const
   { return 0.5*((cv.omega|I*cv.omega)+m*(cv.v|cv.v));}
   // kinetic energy of *this
R totEnr(const VecBH& H)const{ return kinEnr()+j.potEnr(H);}
   // total energy of *this in a homogeneous magnetic field of
   // strength H
Vec elcMom(const Spc& x0=Spc())const{ return (cv.x-x0)*get_q();}
   // returns the moment of the electric charge relative to point
   // x0; Summing up these values of a set of ComPars of zero net
   // charge gives a result which is independent of x0 and is the
   // electric dipol moment of the charges

MagDipole get_j()const{ return j;}
ElcDipole get_qd()const{ return cv.qd;}
void setElcDipole(R dt, const VecE& E);
   // assuming an externally created electric field strength
   // at x, we set the electric dipole moment qd essentially as
   // qdSat=polarizability of the cluster times E.
   // Actually the present value of qd is shifted towards qdSat to
   // a degree which is complete only if dt is large compared
   // to the dielectric relaxation time tRelax=1./(prop.fRelax)
PointCharge getPointCharge()const{ return q;}
PointMass getPointMass()const
   // the corresponding mass point, regularization radius taken
   // from point charge
   { return PointMass(m,cv.x,q.get_r());}

SourceOfVectorField source()const;
   // unified treatment of electromagnetic field sources
   // of particle *this

R get_q()const{return q.get_q();}
   // getting the value of the charge
void set_q(R q_){ q.set_q(q_);}
```

```
   // setting the value of the charge
const Spc& get_x()const{ return cv.x;}
Vec get_v()const{ return cv.v;}

Vec mom()const{ return cv.v*m;}
   // linear momentum

AxVec angMom(Spc const& p)const;
   // returns the total (mechanical) angular momentum
   // of *this relative to the point p in space

Vec get_v(Spc const& p)const{ return cv.v+cv.omega.cross(p-cv.x);}
   // velocity at an arbitrary point p of the object

R get_r()const{ return cc.r;}
R getVolume()const{ return cc.volume;}
R getSurface()const
   { return pow(cc.volume*cc.volume*36*CpmRoot::Pi,R(1.)/3);}
   // this is the surface of a sphere that has the same volume
   // as *this
R get_m()const{ return m;}
R getAlpha()const{ return alpha;}
R get_sigma()const{ return get_q()/getSurface(); }
R getEeff()const{ return cc.prop.E_;}
ContactProperties get_cp()const{ return cc.get_cp();}
R sclVal(const Word& w)const;
   // scalar value
   // returns a scalar value designated by w, where
   // the following cases for w are implemented
   // "mass", "velocity", "angular_velocity",
   // "kinetic_energy", "charge", "electric_dipole",
   // "magnetic_dipole", "volume", "surface"
   // In cases where w indicates a vector quantity,
   // the actual value from this function is the
   // absolute value of the vector quantity.

Z get_id()const{ return cc.id;}

void set_id(Z id1);// a valid id gets never changed unless
  // the id is cancelled explicitely via cancel_id

void setOwner(Z owner1){ cv.owner=owner1;}
Z getOwner()const{ return cv.owner;}
bool isVal()const{ return cc.idValid;}
void cancel_id(){ cc.idValid=false;}

bool idn(ComPar const& cp)const{ // 'identical'
   return (isVal()&&cp.isVal()&&(get_id()==cp.get_id())) ?
   true : false;}
```

```
ScalarField getU()const
   // potential associated with the types of interaction to be
   // considered
   {
      if (SysControl::potentialToShow==3)
         return PointMass(m,cv.x).fieldU();
      else if (SysControl::potentialToShow==2)
         return j.fieldU();
      else
         return q.fieldU()+cv.qd.fieldU();
   }
FieldE getE()const{ return q.field()+cv.qd.field();}
FieldBH getBH()const{ return j.field();}
FieldG getG()const{ return PointMass(m,cv.x).field();}
   // gravitational field generated by the cluster.
VectorField getField(bool noGravity=true)const
{
   if (noGravity) return VectorField(getE(),getBH(),FieldG());
   else return VectorField(getE(),getBH(),getG());
}
   // returns the electric, magnetic, gravitational field as a
   // whole. If the noGravity argument is set to true, the
   // gravitational field is not calculated and thus is left in its
   // default value zero. So no superfluous computational burden
   // results for applications on electromechanics of small
   // particles

R disVal(const ComPar& cl)const;
   // returns the 'the distance' of *this and cl as a fraction of
   // r+cl.r. Thus the value is not limited by 1.

R gap(const RigidObject& ro, Z i)const;
   // returns the clear gap between *this and the j-k-plane
   // of ro.rs where {j,k}={1,2,3}\{i}

R gap(ComPar const& cl)const;
   // returns the clear gap between *this and cl.
   // Is negative in the case of overlap. If *this and cl
   // are sufficiently far away that even the bounding spheres
   // don't overlap, we simply return the clear gap between
   // the bounding sheres (very efficient and sufficiently
   // accurate).

R corGap(ComPar const& cp)const // coarse gap
{ return
   (get_x()-cp.get_x()).absVal()-(get_r()+cp.get_r());
}

void mark(Graph& )const;
   // marks the position (and orientation) on screen.
```

```
         // Invokes SphPar::mark(Graph& ) for all constituent particles.
         // Not useful in 3 dimensions

      void mark(const Camera& ca, const Color& col)const;
         // Visualization via a Camera. Presently the 2-dimensional
         // version is created from the 3-dimensional one by simple
         // translation without exploiting the obvious simpifications
         // occurring in this case. In the three-dimensional case
         // only lines (as opposed to solid areas) give a good
         // representation in anaglyphic mode, thus the representation is
         // by a system of lines.
         // Lines crossing at right angle mark the center of a
         // constituent spherical particle. The point where several lines
         // meet is the center-of-mass.
         // A line ending in a 'blob' gives the direction of a magnetic
         // moment. All lines not ending in a 'blob' end on the surface
         // of the ComPar to be represented.
         // If col is not invisible, it will be used instead the
         // color of the particle
         // If the name-space global boolean variable markComParMec is
         // true not the shape is indicated but the axes of inertia and
         // the vectors of angular velocity and angular momentum.
         // If dt is not 0, the rectilinear path that the particle
         // would go during dt based its present velocity will also be
         // indicated by a line starting at the center of mass

      void mark(const Camera& sc)const
         { Color col=getCol(); mark(sc,col);}
         // same as previous function but uses the own color for drawing

      SubsetOfSpace toSubOfSpc()const;
         // to subset of space
         // This helps visualisation since for objects of type
         // SubsetOfSpace a Camera can create a good anaglyphic
         // representation via its function mark

// new economic storage method
      ComParC getC()const;
      ComParV getV()const;
      void set(const ComParC&, const ComParV&);
      ComPar(const ComParC&, const ComParV&);

// methods that work on arrays of ComPars
      static V<ComPar> neutralize(const V<ComPar>& vc);
         // creates from an array of particles one that hat total charge
         // 0 by multiplying all positive charges by one factor and all
         // negative charges by the inverse of this factor.

      static VectorField farField(const V<ComPar>& vc, const V<Spc>& vx);
         // returns a vector field that is made by replacing the
```

```
      // field created by particles vc by a a sum of simple multipole
      // fields with poles located at vx[i], i=1,...vx.dim().
      // Naturally all particles located in the Voronoy neighborhood
      // of vx[i] are used to built the multipole at vx[i].

   static VectorField meanField(const V<ComPar>& vc, const V<Spc>& vx);
      // The previous field, as the name says, is not intended to be
      // represent something useful when evaluated at points which
      // are close to any of the points vx. The present construction
      // aims at filling this gap and to define the field generated by
      // the particles in the part of space where the particles are
      // located. A verbal description of the method tends to be less
      // readable than the transparent code in
      // cpmextendedparticlescpp.h

   static B isVal(const V<ComParV>& vv);
      // returns true if there are no reasons to reject data vv
      // Present implementation based on the velocity distribution

   static B isVal(const V<ComPar>& vc);
      // returns true if there are no reasons to reject data vc
      // Implementation based on calling
      // static B isVal(const V<ComParV>& vv)

   static void showVelocities(const V<ComPar>& vp, const Camera& ca);
   static void showProperty(const Word& theme, const V<ComPar>& vp,
      const Camera& ca, const R_Vector& parameters);
};
```

37 cpmconstphys.h

```
//? cpmconstphys.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_CONSTPHYS_H_
#define CPM_CONSTPHYS_H_

// Values of physical constants in SI units. Comments indicate the SI
// unit
// Source: Huette: Grundlagen ... 29 Auflage p. B2, source there is
// CODATA Bull 63 Nov 1986
// All quantities end in _PHYS, names of greek letters are abreviated
// to three letters.
// 2007-02-24 comment added to Boltzmann's constant

#include <cpmnumbers.h>

namespace CpmPhysics{

   using CpmRoot::R;
   using CpmRoot::Pi;

const R c_PHYS=299792458.;
   //  m/s  speed of light in vacuum

const R my0_PHYS=((4e-7)*Pi);
   // (V*s)/(A*m) permeability of vaccum

const R eps0_PHYS=1./(c_PHYS*c_PHYS*my0_PHYS);
   // (A*s)/(V*m) dielectric constant of vaccum

const R influence_PHYS=1./(4*Pi*eps0_PHYS);
   // (V*m)/(A*s) 'influence constant'

const R mag_influence_PHYS=1./(4*Pi*my0_PHYS);
   // (A*m)/(V*s) 'magnetic influence constant'

const R e_PHYS=1.60217733e-19;
   // A*s electrical charge of proton

const R h_PHYS=6.6260755e-34;
   // J*s = A*V*s Planck's constant

const R h__PHYS=h_PHYS/(2*Pi);
   // J*s = A*V*s 'h quer'

const R NA_PHYS=6.0221367e23;
   // 1 Avogadro's constant
```

```
const R k_PHYS=1.380658e-23;
   // J/K Boltzmann's constant
   // Laut Physik Journal Dezember 2006 p. 37
   // ist geplant den Wert 1.38065e-23 als exakt
   // festzulegen und zur Neudefinition der Temperatur zu
   // verwenden.

const R R_PHYS=k_PHYS*NA_PHYS;
   // J/K universal gas constant

const R sig_PHYS=(Pi*Pi/60.)*(k_PHYS*k_PHYS/(h__PHYS*c_PHYS))*
(k_PHYS*k_PHYS/(h__PHYS*c_PHYS))/h__PHYS;
   // W/(m^2 * K^4) Stefan-Boltzmann constant

const R me_PHYS=9.1093897e-31;
   // kg mass of electron

const R mp_PHYS=1.6726231e-27;
   // kg mass of proton

const R mu_PHYS=1.6605402e-27;
   // kg mass unit mu = mass(C12)/12

const R muBohr_PHYS=e_PHYS*h__PHYS/(2*me_PHYS*c_PHYS);
   // A*m^2 Bohr magneton. The magnetic moment (Ampere's magnetic Moment)
   // of the electron is 1.00115966 times muBohr_PHYS.
   // Coulomb's magnetic moment is mu0 times this and has dimension T*m^3

const R G_PHYS=6.67259e-11;
   // N*m^2/kg^2 gravitational constant G , sometimes gamma

// more astronomical additions (from Kuchling, not particulary accurate)

const R mEarth_PHYS=5.97e24;
   // kg mass of Earth

const R mSun_PHYS=3.334e5*mEarth_PHYS;
   // kg mass of Sun

const R AU_PHYS=1.49598e11;
   // m Astronomical Unit (of length)

const R gEarth_PHYS=9.81;
   // acceleration of free fall as a conventional constant

extern R gEarth;
   // acceleration of free fall at the place under consideration
   // can meanigfully get changed, so should not be treated as constant
```

```
} // namespace

#endif
```

38 cpmconstphys.cpp

```
//? cpmconstphys.cpp
//? Status of work 2008-10-25.
//?
#include <cpmconstphys.h>

CpmRoot::R CpmPhysics::gEarth=CpmPhysics::gEarth_PHYS;
   // acceleration of free fall
```

39 cpmcvector.h

```
//? cpmcvector.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_CVECTOR_H_
#define CPM_CVECTOR_H_
/*
   Purpose: Defining vectors wirh complex components in
      close analogy to R_Vector

*/

#include <cpmrvector.h>
#include <cpmc.h>

namespace CpmArrays{

   using std::vector;
   using CpmRoot::C;
   using CpmArrays::Perm;
   using CpmLinAlg::R2;
   using CpmLinAlg::R3;
   using CpmLinAlg::R4;
   using CpmGeo::Iv;

   typedef Vr<Z> I_Vector;

   class C_Vector{ // class of vectors with C-valued components.
      // arbitrary length, dynamic allocation

   Vr<C> rep;
      // we choose composition instead of inhertance, so the
      // interface of Vr<C> needs to be known only to the implementer
      // of this class, the user finds the interface of C_Vector
      // in the present file together with the macro definitions
      // in cpminterfaces.h

   C_Vector(const Vr<C>& x):rep(x){}

   static C prodFuncReal(const C& z1, const C& z2)
      { return z1*z2.real();}

   static C addFunc(const C& z1, const C& z2, const C& t)
      { return z1+z2*t.real();}

public:
   typedef C_Vector Type;
   typedef C ScalarType;
```

```
   CPM_IO
   CPM_ORDER
   CPM_TEST
   CPM_DESCRIPTORS
   CPM_CONJUGATION
   CPM_DOT_PRODUCT
   CPM_SUM_M
   CPM_PRODUCT_M
   CPM_SCALAR_M
   CPM_DIFFERENCE
   CPM_DIVISION
   Z dim()const{ return rep.dim();}

   C_Vector& operator *=(R r);
   C_Vector operator *(R r)const;
   C_Vector& add(const C_Vector& v, R t);

   const C& cyc(Z i)const{ return rep.c().cyc(--i);}
   C& cyc(Z i){ return rep.c().cyc(--i);}

//   const C& cyc(Z i)const{ return rep.cyc(i);}
//   C& cyc(Z i){ return rep.cyc(i);}

   C_Vector():rep(1,C()){}
      // default vector has one (not zero!) components
      // same with Vl<C> and V<C>
   explicit C_Vector(Z n):rep(n,C()){}
   explicit C_Vector(const Vl<C>& v):rep(V<C>(v)){}
   explicit C_Vector(const V<C>& v):rep(v){}
   explicit C_Vector(const vector<C>& v);

   C_Vector(Z n, C x):rep(n,x){}
      // constructor for a vector with n comonents equal to x

   operator V<C>()const{ return rep;}

   R_Vector real()const;

   R_Vector imag()const;

   C& operator[](Z i){ return rep.c()[--i];}
   const C& operator[](Z i)const{ return rep.c()[--i];}

   C in_(const C& t){ return rep.in_(t);}

   V<Z> permutationForIncreasingOrder()const
      {return rep.permutationForIncreasingOrder();}

   C_Vector permute(const V<Z>& p)
      { return C_Vector(rep.permute(p));}
```

```
   C_Vector operator*(const Perm& p)const
      { return C_Vector(rep.permute(p.getRep()));}

   C_Vector increasingCopy()const
      { return C_Vector(rep.increasingCopy());}

   C_Vector decreasingCopy()const
      { return C_Vector(rep.decreasingCopy());}

   void order(){ rep.order();}

   void set_(const C& t){ rep.set_(t);}
      // changes all components to the value t

    C_Vector scaleTo(const R& r)const;
// returns a vector of absolute value |r| and the direction of
// *this for positive r and antidirection for negative r.
// same function is available for complex n

   C_Vector der(R h=1.) const;
// The vector a.der() results from the vector a by numerical
// differentiation of a function, the equi-distantly sampled image
// (sampling step = h) of which the vector a is considered to be.
// Formulas: Bronstein Semendiajew, Taschenbuch der Mathematik S. 768
// Tabelle 7.13 for generic points and Abramowitz Stegun p. 914 Table
// 25.2 for the boundary points.

   C_Vector operator & (const C_Vector& x1)const
   { return C_Vector(rep&x1.rep);}
   // concatenation

   C_Vector operator & (const C& z)const
   { return C_Vector(rep&z);}

   C_Vector prepend(const C& z)const
   { return C_Vector(rep.prepend(z));}


   void printLineOn(ostream& out , const Word& separator="  " )const
      // prints the components of *this, separated by the character
      // string separator. In default mode all numbers are in a line which
      // is usefull for creating data that later have to be given to a
      // plot program such as gnuplot.
   { for (Z i=1;i<=rep.dim();i++) out<<separator<<rep[i];}

};

// inline reduction of most functions to those of rep in Vr<C>
// in order not to loose speed
```

```
inline bool C_Vector::prnOn(ostream& out)const
{ return rep.prnOn(out);}

inline bool C_Vector::scanFrom(istream& in)
{ return rep.scanFrom(in);}

inline Z C_Vector::com(const C_Vector& x)const
{ return rep.com(x.rep);}

inline C_Vector& C_Vector::operator +=(const C_Vector& x)
{ rep+=x.rep; return *this;}

inline C_Vector& C_Vector::operator *=(const C& r)
{ rep*=r; return *this;}

inline C_Vector& C_Vector::operator *=(const C_Vector& x)
{ rep*=x.rep; return *this;}

inline C_Vector C_Vector::neg()const
{ return C_Vector(-rep);}

inline C_Vector C_Vector::ranVal(Z j)const
{ return C_Vector(rep.ranVal(j));}

inline Z C_Vector::hashVal()const
{ return rep.hashVal();}

inline C_Vector C_Vector::testVal(Z j)const
{ return C_Vector(rep.testVal(j));}

inline Word C_Vector::nameOf()const
{ return Word("C_Vector");}

inline Word C_Vector::toWord()const
{ return rep.toWord();}

inline C_Vector C_Vector::net(Z i)const
{ return C_Vector(rep.net(i));}

inline C_Vector C_Vector::inv()const
{ return C_Vector(rep.inv());}

inline C_Vector C_Vector::con()const
{ return *this;}

inline C C_Vector::operator |(const C_Vector& x)const
{ return (rep|(x.rep));}

} // namespace
```

```
#endif
```

40 cpmcvector.cpp

```
//? cpmcvector.cpp
//? Status of work 2008-10-25.
//?
#include <cpmcvector.h>

using namespace CpmRoot;
using namespace CpmSystem;
using namespace CpmArrays;

C_Vector::C_Vector(const vector<C>& v):rep((Z)v.size())
{
   vector<C>::const_iterator i;
   Z j=0;
   for (i=v.begin();i!=v.end();++i) rep[++j]=*i;
}

C_Vector& C_Vector::operator *=(R r)
{
   rep.c().fc_(prodFuncReal,C(r,0));
   return *this;
}

C_Vector& C_Vector::add(const C_Vector& v, R t)
{
   rep.c().fg_(addFunc,v.rep.c(),C(t,0));
   return *this;
}

C_Vector C_Vector::operator *(R r)const
{
   C_Vector res=*this;
   return res*=r;
}

R_Vector C_Vector::real()const
{
   Z n=dim();
   R_Vector res(n);
   for (Z i=1;i<=n;i++) res[i]=rep[i].real();
   return res;
}

R_Vector C_Vector::imag()const
{
   Z n=dim();
   R_Vector res(n);
   for (Z i=1;i<=n;i++) res[i]=rep[i].imag();
```

```
   return res;
}

C_Vector C_Vector::scaleTo(const R& r)const
      // returns a vector of absolute value |r| and the direction of
      // *this for positive r and antidirection for negative r.
      // same function is available for complex numbers
{
   R rOrig=absVal();
   C_Vector res=*this;
   if (rOrig!=0.){
      R fac=r/rOrig;
      res*=C(r);
   }
   return res;
}

C_Vector C_Vector::der(R h) const
// The vector a.der() results from the vector a by numerical
// differentiation of a function, the equi-distantly sampled image
// (sampling step = h) of which the vector a is considered to be.
// Formulas: Bronstein Semendiajew, Taschenbuch der Mathematik S. 768
// Tabelle 7.13 for generic points and Abramowitz Stegun p. 914 Table
// 25.2 for the boundary points.
{
   R s;
   Z i,n=dim();
   C_Vector a(n);
   Word loc("C_Vector C_Vector::der(R h) const");
   cpmassert(h!=0,loc);
   if (n==1){
      a[1]=C();
   }
   else if (n==2){
   s=1./h;
      a[1]=s*((*this)[2]-(*this)[1]);
      a[2]=a[1];
   }
   else{  // n>=3
      s=1./(12*h);
      R s1=1./(2*h);
      a[1]=s1*(-3*(*this)[1]+4*(*this)[2]-(*this)[3]);
      a[2]=s1*(-(*this)[1]+(*this)[3]);
      a[n-1]=s1*(-(*this)[n-2]+(*this)[n]);
      a[n]=s1*((*this)[n-2]-4*(*this)[n-1]+3*(*this)[n]);
      for (i=3;i<=n-2;i++)
      a[i]=s*((*this)[i-2]-8*(*this)[i-1]+8*(*this)[i+1]-(*this)[i+2]);
    }
   return a;
}
```

41 cpmdefinitions.h

```
//? cpmdefinitions.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_DEFINITIONS_H_
#define CPM_DEFINITIONS_H_
/*********************************************************
*   Author: Ulrich Mutze                                 *
*   Copyright (c) 1999 Heidelberg Digital Finishing GmbH  *
*   All rights reserved                                  *
*                                                        *
*********************************************************
   cpmdefinitions.h
   Description: In this file one can place defines that are valid in all
   translation units working with Cpm classes
   Cpm = Classes for Physics and Mathematics || C+- (C plus minus)
         -           -           -                   - -    -
*********************************************************************/

// Definitions for a console-based program without graphics

//#define NDEBUG

//#define CPM_LONG

#define CPM_NAMEOF

#define CPM_USECOUNT

#define CPM_RANGE_CHECK

#define CPM_NOGRAPHICS

//#define CPM_USE_MPI
   // MPI = message passing interface

#define CPM_TEMPLATE_TESTS
   // having this defined increases code size
   // it helps to detect inconsistencies in interfaces of template
   // classes
   // should be disabled for release compilation

//#define CPM_NOLOG
   // means to disable creation and usage of file streams cpmcerr and
   // cpmdata
```

```
#endif
```

42 cpmdigraph.h

```
//? cpmdigraph.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_DIGRAPH_H_
#define CPM_DIGRAPH_H_
/*
    Description: directed graphs for organizing the
      communication between parallel processes. Later
      also used as base class for the graphics class
      Skeleton

*/
#include <cpms.h>
#include <cpmlinalg.h>

#include <cpmgraph.h>
   // probably graphs without visualization capabilitirs are too poor

namespace CpmGeo{

   using namespace CpmStd;

   using CpmRoot::Z;
   using CpmRoot::Word;
   using CpmRootX::Z1;
   using CpmRootX::R1;
   using CpmArrays::S;
   using CpmArrays::Outside;
   using CpmArrays::V;
   using CpmArrays::X2;
   using CpmLinAlg::Z2;
   using CpmLinAlg::R2;
   using CpmFunctions::F;
   using CpmFunctions::F1;

   using CpmGraphics::Graph;

   template<class X>
   class DiGraph{ // directed graph
      // directed graph. Has nothing to do with
      // CpmGraphics::Graph

      Z1 n_;
         // number of vertices;

      mutable V<bool> act_;
         // an array which holds a boolean variable for each
```

```
      // vertex. Will be used for book-keeping during algorithm
      // runs. Also functions declared as const will be
      // allowed to change the values of act_ during run;
      // they will restore finally to correct initial values
      // so that constancy holds effectively.
      // This tool helped to increase speed of function steps
      // by a factor of 100 !

   void refresh()const{ for (Z i=1;i<=n_;i++) act_[i]=true;}
      // tool for un-doing the changes in act_

   S<Z> grow(const S<Z>& sv)const;
      // adds to a set sv of vertices all vertices which can be
      // reached by one step along an edge. If the argument sv is not
      // a subset of {1,...n_}, an error will occur during run.
      // Therefore the function is private and will only be used
      // under controlled conditions for implementing function steps.

   S<Z> growRes(const S<Z>& sv)const;
      // returns a a set of active particles which can be reached
      // by a single step from sv. The vertices j returned
      // by the functions get marked as in-active by
      // setting act_[j]=false.

   static Z cutNeg(Z n){ return n>0 ? n : 0;}
      // trivial tool for safely defining the constructor
   mutable Z pc_;

protected:

   V<X> x_;
      // Array of objects associated with the vertices
      // x.dim()>=n_ has to be assured, so that for each i
      // with 1<=i<=n_, an x[i] is defined.

   V<C> pp_;
      // planar positions, needed for visualization, added 2007-07

   V< S<Z> > neg_; // neighbors
      // neg_.dim() equals the number of vertices n_
      // neg_[i] is the set of vertices which can be reached via an
      // (directed) edge starting at i. For this to make sense,
      // neg_[i].card() has to be <= neg_.dim() for all admissible
      // i's.
      // Further, S<Z> has to be a subset (not necessarily a proper
      // one) of {1,2,...,n_}

public:
   typedef DiGraph Type;
   typedef DiGraph CloneBase;
```

```
      virtual DiGraph* clone()const{ return new DiGraph(*this);}
      CPM_IO_V
         // IO should be virtual in order to allow
         // additional attributes in derived classes
// all functions test the validity of their arguments and
// stop via cpmerror for unvalid ones.

      DiGraph(const V<X>& obj, Z connect=0):
         n_(obj.dim()),neg_(n_),act_(n_,true),x_(obj),pp_(n_)
         // construction from a list of vertices.
         // Notice that is no problem if two vertices are equal as
         // objects in X. They are different vertices due to their
         // differing index.
         {
            if (connect>0){
               for (Z i=1;i<n_;i++) addEdge(i,i+1);
               if (connect>1) addEdge(n_,1); // was done always till
               // 2003-07-23
            }
         }

      DiGraph(Z n=1):n_(cutNeg(n)),neg_(n_),act_(n_,true),
         x_(n_),pp_(n_){}
         // a graph with n vertices and no edges

      DiGraph(Z m, Z n);
         // creates a graph with m*n vertices with the edges
         // which are natural for my particle modeling application

      DiGraph<X>& operator&=(const DiGraph<X>& g);
         // adding DiGraph g to *this in a way that
         // no edges connecting these two objects are added.
         // Also it is irrelevant whether some x[i] equals
         // a g.x[j]. As vetexes of the new graph these
         // are considered different.

      void set_neg(V< S<Z> > const& negNew);
         //: set neighbors
         // setting the neighbors list from outside
         // only matching data from negNew get used.
         // data out of range are ignored

      V< S<Z> > const& get_neg()const{ return neg_;}
         //: get neighbors

      S<Z> const& get_neg(Z i)const{ return neg_.read(i);}
         //: get neighbors
         // if i is not the index of a valid vertex, the return value
         // is the void set
```

```
V<X> const& get_x()const{ return x_;}
   //: get x

X const& operator[](Z i)const{ return x_[i];}

void addVertex(const X&);
   //: add vertex
   // adds a new vertex, no change of edges

void addEdge(Z i, Z j);
   //: add edge
   // makes (i->j) an edge
   // does nothing if i,j are not admissible ,
   // stopped till 2003-07-23
   // also does nothing for i==j
void addEdgeSym(Z i, Z j){ addEdge(i,j); addEdge(j,i);}
   //: add edge symetrically
   // makes (i->j) and (j->i) edges
void remEdge(Z i, Z j);
   //: remove edge
   // 'removes' an edge (nothing is done if no edge is there
   // or i isn't a vertex, no stop in this case)
void remEdge(Z i, const S<Z>& s);
   //: remove edge
   // remove a set of edges from vertex i
void remEdgeSym(Z i, Z j){ remEdge(i,j); remEdge(j,i);}
   //: remove edge symmetrically
   // self-explanatory
DiGraph<X> select(const V<bool>& s)const;
   //: select
   // returns the sub-DiGraph of *this which has as vertices
   // just those x[i] for which s[i]==true. The edges
   // of the new objects are just those
   // x[i]-->x[j] for s[i]==true and s[j]==true
   // j \in neg[i]. An application of this operation
   // is to select the part of *this that is within the
   // frame of some visualization tool.
S<Z2> edges(Z i)const;
   //: edges
   // returns the set of edges starting at i in a natural
   // representation as pairs of integers
S<Z2> edges()const;
   //: edges
   // returns the set of all edges of *this in a natural
   // representation as a set of Z2's
Z steps(Z i, Z j)const;
   //: steps
   // returns the length (as number of steps) of the shortest path
   // from i to j. If there is no such path we return -1
V< S<Z> > shells(Z i)const;
```

```
         //: shells
         // retuns a vector res such that res[1] is the set of all
         // vertices which can be reached from i by one step
         // res[2] that can be reached by 2 steps and not by 1 step
         // res[3] those that can be reached by 3 steps but not by less
         // ...
      Z dim()const{ return n_;}
         //: dimension
      Z card()const{ return n_;}
         //: cardinality
// functions added 2007-07
      void setPlnPos_(Word meth="ordered");
         //: set planar positions
         // "random" and "ordered" are the present options
      void setPlnPos_(V<C> const& vc)
      { if (pp_.dim()==vc.dim()) pp_=vc;}
         //: set planar positions
         // from a list of positions
      void mark(Graph& g)const;
         //: mark
      void rev_();
         //: revert
         // changes each edge i->j into an edge j->i
      void removeAllEdges_(){ neg_=V< S<Z> >(n_,S<Z>());}
         //: remove all edges
      X2< S<Z>, S<Z> > sorSink()const;
         //: source sink
         // Let res be the return value.
         // Then res.c1()
         // is the set of all indexes of vertices which are
         // source points (i.e. have out-going edges only)
         // And res.c2()
         // is the set of all indexes of vertices which are
         // sink points (i.e. have in-coming edges only).
      Word toWord(Z i)const{ return x_(i).toWord();}
         //: to word
         // returns a Word characterizing x_[i]. Valid for all i
         //
      bool wrtFor(ostream& str , bool writeComments)const;
         //: write formatted
   };

   bool close(Z2 a, Z2 b);
   bool rimRelated(Z2 a, Z2 b, Z rim);
      // defined in cpmdigraph.cpp
   S<Z> shiftNegFunc(S<Z> const& s, Z1 const& n);
      // means for implementing operator &=

template<class X>
DiGraph<X>::DiGraph(Z m, Z n):
```

```
   n_(cutNeg(m*n)),neg_(n_),act_(n_,true),x_(n_),pp_(n_)
{
   if (m<1 || n<1) cpmerror("DiGraph(Z,Z): unvalid arguments");
   V<Z2> ij(n_);
   for (Z k=0;k<n_;k++){
      Z i=k/n;
      Z j=k%n;
      ij[k+1]=Z2(i,j);
      x_[k+1]=R2(i,j);
   }
   for (Z k1=1;k1<=n_;k1++){
      for (Z k2=1;k2<=n_;k2++){
         if (close(ij[k1],ij[k2])||rimRelated(ij[k1],ij[k2],n))
            addEdge(k1,k2);
      }
   }
}

template<class X>
DiGraph<X>& DiGraph<X>::operator&=(const DiGraph<X>& g)
{
   F< S<Z>, S<Z> > tr=F1<S<Z>,Z1,S<Z> >(n_)(shiftNegFunc);
   V< S<Z> > gn=g.neg_(tr);
   neg_&=gn;
   x_&=g.x_;
   pp_&=g.pp_;
   n_+=g.n_;
   return *this;
}

template <class X>
void DiGraph<X>::set_neg(V< S<Z> > const& negNew)
{
   const Z mL=5;
   cpmmessage(mL,"DiGraph<X>::set_neg(V< S<Z> >) started");
   Z d1=n_;
   Z d2=negNew.dim();
   Z d=cpminf<Z>(d1,d2);
    for (Z i=1;i<=d;i++){
      S<Z> negni=negNew[i];
      if (negni.card()>0){
         while (negni.last()>d1){
            negni.remove(negni.last());
         }
      }
      neg_[i]=negni;
   }
   cpmmessage(mL,"DiGraph<X>::set_neg(V< S<Z> >) done");
}
```

```
template <class X>
void DiGraph<X>::addVertex(const X& v)
{
   x_&=v;
   n_+=1;
   bool actAdd=true;
   act_&=actAdd;
   S<Z> negAdd;
   neg_&=negAdd;
}

template<class X>
void DiGraph<X>::addEdge(Z i, Z j)
{
   if(i<1) return;
   else if (j<1) return;
   else if (i> n_) return;
   else if (j> n_ ) return;
   else if (i==j) return;
   else neg_[i].add(j);
}

template<class X>
S<Z> DiGraph<X>::grow(const S<Z>& sv)const
{
   S<Z> res(sv);
   for (Z i=1;i<=sv.card();i++){
      Z vi=sv[i];
      Z dvi=neg_[vi].card();
      for (Z j=1;j<=dvi;j++){
         res.add(neg_[vi][j]);
      }
   }
   return res;
}

template<class X>
S<Z> DiGraph<X>::growRes(const S<Z>& sv)const
{
   S<Z> res;
   for (Z i=1;i<=sv.card();i++){
      pc_++;
      Z vi=sv[i];
      Z dvi=neg_[vi].card();
      for (Z j=1;j<=dvi;j++){
         Z vj=neg_[vi][j];
         if (act_[vj]){
            res.add(vj);
            act_[vj]=false;
         }
```

```
      }
   }
   return res;
}

template<class X>
void DiGraph<X>::remEdge(Z i, Z j)
{
   if(i<1 || j<1 || i> n_ || j> n_) return;
      // nothing to do
   S<Z> remj;
   remj.add(j); // direct construction interferes with the protected
      // constructor S<T>(Z) which here causes trouble due to T=Z
   neg_[i]=neg_[i]-remj;
}

template<class X>
void DiGraph<X>::remEdge(Z i, const S<Z>& s)
{
   if(i<1 || i> n_) return;
      // nothing to do
   neg_[i]=neg_[i]-s;
}

template<class X>
DiGraph<X> DiGraph<X>::select(const V<bool>& s)const
{
   pair<V<X>,V<Z> > pvv=x_.select(s);
   V<X> xNew=pvv.first;
   V<Z> ind=pvv.second;
   DiGraph<X> res(xNew);
   Z n=res.dim(),i;
   for (i=1;i<=n;i++){
      S<Z> nii=neg_[ind[i]];
      nii.select(s);
      res.neg_[i]=nii;
   }
   return res;
}

template<class X>
S<Z2> DiGraph<X>::edges(Z i)const
{
   if(i<1 || i> n_)
      cpmerror("DiGraph::::edges(Z)const: unvalid argument");
   S<Z2> res;
   S<Z> v(neg_[i]);
   for (Z j=1;j<=v.card();j++){
      res.add(Z2(i,v[j]));
   }
```

```
   return res;
}

template<class X>
S<Z2> DiGraph<X>::edges()const
{
   S<Z2> res;
   for (Z i=1;i<=n_;i++) res=(res|edges(i));
   return res;
}

template<class X>
V< S<Z> > DiGraph<X>::shells(Z i)const
{
   if(i<1 || i> n_)
      cpmerror("DiGraph::::shells(Z)const: unvalid argument");
   S<Z> s=neg_[i];
   V<S<Z> > res(1);
   res[1]=s;
Label:
   s=growRes(s);
   if (s.card()==0){
      refresh();
      return res;
   }
   else{
      res&=s;
      goto Label;
   }
}

template<class X>
Z DiGraph<X>::steps(Z i, Z j)const
{
   if(i<1 || i> n_ || j<1 || j> n_)
      cpmerror("DiGraph::steps(Z,Z)const: unvalid argument");
   S<Z> s=neg_[i];
   Z count=1;
Label:
   if (s.hasElm(j)){
      refresh(); // this compensates our
         // breaking strict constancy
      return count;
   }
   s=growRes(s);
   if (s.card()==0){
      refresh();
      return -1;
   }
   else{
```

```
      count++;
      goto Label;
   }
}

template<class X>
bool DiGraph<X>::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("DiGraph",str);
   cpmp(n_);
   cpmp(neg_);
   cpmp(x_);
   return true;
}

template<class X>
bool DiGraph<X>::scanFrom(istream& str)
{
   cpms(n_);
   cpms(neg_);
   cpms(x_);
   return true;
}

template<class X>
void DiGraph<X>::rev_()
{
   S<Z2> ed=edges();
   removeAllEdges_();
   Z n=ed.card();
   for (Z i=1;i<=n;++i){
      Z2 z2=ed[i];
      addEdge(z2[2],z2[1]);
   }
}

template<class X>
void DiGraph<X>::setPlnPos_(Word meth)
{
   Z i,j;
   if (meth=="random"){
      R eps=0.05;
      Iv ivx(eps,1-eps);
      Iv ivy=ivx;
      Z ir=7;
      for (i=1;i<=n_;++i){
         R xi=ivx.ran(ir++);
         R yi=ivy.ran(ir++);
         pp_[i]=C(xi,yi);
      }
```

```
   }
   else if (meth=="ordered"){
      Z i0=1;
      Word w0("cpmbasictypes.h");
      for (i=1;i<=n_;++i){
         if (x_[i]==w0){
            i0=i;
            break;
         }
      }
      pp_[i0]=C(0,0.5);
      R fracAmpl=0.25;
      V< S<Z> > vsz=shells(i0);
      Z nSh=vsz.dim();
      Iv iv(0,1);
      V<R> vr=iv.div(nSh+1);
      R xAmpl=fracAmpl/nSh;
      Z ir=13;
      for (i=1;i<=nSh;++i){
         S<Z> si=vsz[i];
         Z sic=si.card();
         R xi=vr[i+1];
         V<R> ivd=iv.div(sic);
         R yAmpl=fracAmpl/sic;
         yAmpl=CpmRootX::inf<R>(yAmpl,xAmpl);
         for (j=1;j<=sic;++j){
            Z nij=si[j];
            pp_[nij]=C(xi+Iv(xAmpl).ran(ir++),ivd[j]+Iv(yAmpl).ran(ir++))\
            ;
         }
      }
      C z0(0.5,0.5);
      R fac=0.8;
      for (j=1;j<=n_;++j){
         pp_[j]=z0+(pp_[j]-z0)*fac;
      }
   }
   else ;
}

namespace{
   Word transForm(Word const& fn)
   {
      return fn.cut(2).cut(-3);
   }
}

template<class X>
void DiGraph<X>::mark(Graph& g)const
{
```

```
   using namespace CpmGraphics;
   using namespace CpmImaging;
   Z i,j;
   for (i=1;i<=n_;++i){
      S<Z2> edgi=edges(i);
      Z ci=edgi.card();
      for (j=1;j<=ci;++j){
         Z2 a=edgi[j];
         C c1=pp_[a[1]];
         C c2=pp_[a[2]];
         C cm=(c1+c2)*0.5;
         g.setThcLin_(true);
         g.draw(c1,cm,GREEN);
         g.setThcLin_(false);
         g.draw(cm,c2,GREEN);
      }
   }
   for (i=1;i<=n_;++i) g.mark(pp_[i],16,RED);
   R eps=0.01;

   for (i=1;i<=n_;++i){
      C ci=pp_[i];
      Word xi=transForm(x_[i]);
      g.addText(xi,ci[1]+eps,ci[2],BLACK,0);
   }
}

template<class X>
X2< S<Z>, S<Z> > DiGraph<X>::sorSink()const
{
   S<Z2> ed=edges();
   V<bool> isSink(n_,true);
   V<bool> isSource(n_,true);
   Z i;
   for (i=ed.b();i<=ed.e();++i){
      Z2 z2=ed[i];
      Z i=z2[1];
      Z j=z2[2];
      // now we have an edge (i-->j)
      // if isSink should finally contain all indexes from which
      // no edge arises, it has to be disqualified now for index i:
      // (since there is a edge arising from i)
      isSink[i]=false;
      // if isSource should finally contain all indexes to which
      // no edge enters, it has to be disqualified now for index j:
      // (since there is an edge entering j)
      isSource[j]=false;
   }
   V<Z> list(n_);
   for (i=1;i<=n_;++i) list[i]=i;
```

```
   return X2< S<Z>, S<Z> >(S<Z>(list,isSource),S<Z>(list,isSink));
}

template<class X>
bool DiGraph<X>::wrtFor(ostream& str , bool writeComments)const
{
   X2< S<Z>, S<Z> > x2ss=sorSink();
   S<Z> source=x2ss.c1();
   S<Z> sink=x2ss.c2();
   Z i,j,k;
   Word tab0="";
   Word incr="  ";
   for (i=source.b();i<=source.e();++i){
      Z vi=source[i];
      Word tab=tab0;
      str<<tab.str()<<toWord(vi).str()<<endl;
      S<Z> vsi=get_neg(vi);
      tab&=incr;
      for (j=vsi.b();j<=vsi.e();++j){
         Z sj=vsi[j];
         str<<tab.str()<<toWord(sj).str()<<endl;
         S<Z> vsj=get_neg(sj);
         tab&=incr;
         for (k=vsj.b();k<=vsj.e();++k){
            Z sk=vsj[k];
            str<<tab.str()<<toWord(sk).str()<<endl;
         }
      }
   }
   return true;
}

} // namespace

#endif
```

43 cpmdigraph.cpp

```
//? cpmdigraph.cpp
//? Status of work 2008-10-25.
//?
#include <cpmdigraph.h>

namespace CpmGeo{
   using namespace CpmRoot;
   using namespace CpmLinAlg;
   using CpmArrays::Vo;

   bool close(Z2 a, Z2 b)
   {
      R r=(a-b).absVal();
      return( r>0.5 && r<1.5);
   }

   bool rimRelated(Z2 a, Z2 b, Z rim)
   {
      return (a.x1==0 && b.x1==(rim-1))||(a.x1==(rim-1) && b.x1==0);
   }


   S<Z> shiftNegFunc(S<Z> const& s, Z1 const& n)
   {
      Vo<Z> a=s.toV();
      for (Z i=1;i<=a.dim();++i) a[i]+=n;
      return S<Z>(a);
   }


} // namespace
```

44 cpmdim.h

```
//? cpmdim.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_DIM_H_
#define CPM_DIM_H_
/*
Description: Common basis for 2D and 3D geometry
*/

#include <cpmlinalg.h>
#include <cpmangle.h>
#include <cpminterval.h>
#include <cpmvo.h>
#include <cpmx.h>
#include <cpmr2func.h>
#include <cpmfield.h>
#include <cpmvectorgraphics.h>

namespace CpmColCam{ // color camera
   // lean namespace for declarations in the range of geometry-related
   // classes
   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::C;
   using CpmRoot::Word;
   using CpmRoot::Pi;

   using CpmGeo::Angle;
   using CpmGeo::Iv; // Interval
   using CpmGeo::DiGraph; // directed graph
   using CpmGeo::RAD;
   using CpmGeo::DEG;

   using CpmRootX::R1;
   using CpmRootX::Z1;

   using CpmLinAlg::Z2;
   using CpmLinAlg::Z3;
   using CpmLinAlg::R2;
   using CpmLinAlg::R3;
   using CpmLinAlg::RR2;
   using CpmLinAlg::RR3;

   using CpmFunctions::F;
   using CpmFunctions::F1;

   using CpmArrays::V;
```

```
   using CpmArrays::X2;
   using CpmArrays::X4;
   using CpmArrays::Vo;
}

namespace CpmDim2{

   using namespace CpmStd;
   using namespace CpmColCam;

   typedef R2 Rdim;
      // space of components of polar vectors
   typedef Z2 Zdim;
      // related to lattice points in space
   typedef RR2 RRdim;
      // space of corresponding matrices
   typedef R1 Rdim_;
      // hyperplane in Rdim 'R(dim-1)'
   const Z dimVec=2;
   const Z dimAxVec=1;
   #include <cpmdim2.h>
   #include <cpmdimx.h>
}
namespace CpmDim3{

   using namespace CpmStd;
   using namespace CpmColCam;

   typedef R3 Rdim;
      // space of components of axial vectors
   typedef Z3 Zdim;
      // related to lattice points in space
   typedef RR3 RRdim;
      // space of corresponding matrices
   typedef R2 Rdim_;
      // hyperplane in Rdim 'R(dim-1)'
   const Z dimVec=3;
   const Z dimAxVec=3;
   #include <cpmdim3.h>
   #include <cpmdimx.h>
}

#endif
```

45 cpmdim.cpp

```
//? cpmdim.cpp
//? Status of work 2008-10-25.
//?
#include <cpmrmateig.h>
#include <cpmalgorithms.h>
#include <cpmsmoothlocalinterpol.h>
#include <cpmdim.h>

using CpmRoot::Z;

namespace CpmDim2{

const Z Vec::dimension=2;
const Z AxVec::dimension=1;

#include <cpmdim2cpp.h>
#include <cpmdimxcpp.h> // common part of implementation

}

namespace CpmDim3{

const Z Vec::dimension=3;
const Z AxVec::dimension=3;

#include <cpmdim3cpp.h>
#include <cpmdimxcpp.h> // common part of implementation

}
```

46 cpmdim2.h

```
//? cpmdim2.h
//? Status of work 2008-10-25.
//?
/*
    Description: see cpmdim3.h. The present file handles the
      2-dimensional case:
*/

//doc namespace CpmDim2{

   class Group;
   class RefSys;
   class Vec;

//////////////////// class AxVec ////////////////////////////////////////

   class AxVec{
// axial vectors, rotation descriptors, 1-dimensional in this case
      // space of 'axial vectors' i.e. rotations in
      // 2-dimensional space and their infinitesimal generators.
      // Interestingly, this is the autonomous class
      // (i.e. needs no support from other classes)
      // Vector space structure over R is in this case not inherited from
      // base class.
      // If we interprete an axial vector e.g. as an angular velocity the
      // the absolute value of the AxVec is dphi/dt. If it is a finite
      // rotation, the length is phi.
      // The finite rotation belonging to *this with data element x1
      // is the one (x,y)|--> (x',y') such that
      // x'=cos(x1)*x-sin(x1)*y
      // y'=sin(x1)*x+cos(x1)*y
      // see function actionOfRotationVectors(...) in cpmdim2cpp.h

      friend class Group;
      friend class Vec;

      R x3;
         // a data element of this name would result from
         // class AxVec:public R1{ ...};. Since meanwhile I defined R1 in
         // a way that it allows automatic conversion to R and misses
         // many functions which R2 and R3 define, I can't use R1 any
         // longer in this way.

   public:
      typedef AxVec Type;
      CPM_IO
      CPM_ORDER
```

```
Word nameOf()const{ return Word("CpmDim2::AxVec");}

static const Z dimension;
   // in our case 1

AxVec():x3(0){};
   // vector with all components equal to zero

AxVec(const R& a, const RefSys& rs);

explicit AxVec(R r):x3(r){}
   // constructor from components

 AxVec(Angle const& alpha, Z i=3):x3((i==3 ? alpha.toR() : 0.)){}
   // constructor from rotation angle and master frame axis
   //
   // Rotation arround 3-axis with angle phi. The skrew sense of
   // the rotation is the standart one: Looking against the axis
   // vector, a positive phi indicates counter-clock-wise rotation.
   // Notice that there are also negative Angles a
   // (a.idPos()==false)
   // These thus describe clock-wise rotations. This implies that
   // a rotation that is close to 360 degrees, all
   // counter-clock-wise is a kind of being not describable by an
   // AxVec. If Argument i is different from 3 we return the
   // same element as for phi=0 and i==3. This is most likely
   // to make programs written with 3D in mind also valid for 2D.
   //
   // Note on problems encountered:
   // If there is a 0 instead of 0., both the MS compiler and
   // gnu-compiler compile without warning but in running I found x3
   // initialized as 1 in a case where alpha=Angle(90,DEG).
   // Reason is unclear. Consequence: avoid expressions that rely
   // on type conversions in a constructor initialization list.
   // The second pair of brackets, also helps and than allows to
   // write  0 instead of 0. ! curious!
   // On the other hand: to be able to do some basic computation
   // inside initializers is extremely useful.

AxVec(Angle const& phi, Vec const& axis);
   // same as previous function; the second argument has no
   // influence; it is there only for reasons of uniformity to the
   // 3D case. Defined later in the presnt file as inline

AxVec(Angle const& phi, AxVec const& axis):x3(phi){axis;}
   // same as previous function; the second argument has no
   // influence.

 AxVec(const V<Vec>& b1, const V<Vec>& b2, bool careful=true);
   // constructs the finite rotation which transforms the
```

```
         // orthonormal basis b1 into the orthonormal basis b2.
         // If the third parameter has value true, one actually works
         // with potentially modified versions of b1 and b2 which are
         // guaranteed to be orthonormal bases.

      R& operator[](Z i);
      R const& operator[](Z i)const;
// simple access
      R operator()(void)const{ return x3;}

// group structure, composition of rotations.
      friend AxVec operator *(const AxVec& a1, const AxVec& a2 );
         // see  Vec operator *(const Vec& x, const AxVec& d);
         // for a conceptual definition of this product.
      friend AxVec operator !(const AxVec& a){ return AxVec(-a.x3);}

      AxVec& operator *=(const AxVec& a){ return *this=(*this)*a;}

      AxVec& operator *=(Group const& g){ return *this;}
         // AxVec is perpendicular to the plane which g transforms.
         // The action thus is trivial. (was wrong till 2006-03-28)

      AxVec& mir_(Z i) // mirror (reflection), clear from code.
      {
         if (i==3) x3=-x3;
         return *this;
      }

      AxVec pow(R p)const;

// operations for infinitesimal rotations and angular velocities

   AxVec exp(const R& t, bool accurate=true)const;
         // the standard exponential function from the Lie algebra
         // (to which *this belongs here) and the Lie group (return
         // value).
         // If omega is a constant angular velocity, then omega.exp(t) is
         // the finite rotation which results from having this angular
         // velocity be effective over the time span t.
         // Since in 2D exp is the identity, there is no need for an
         // fast approximate representation. Thus the second argument
         // has no influence in this case (3D is different!)

   AxVec adjointAction(const AxVec& d)const;
         // returns *this since rotating the 1-dimensional AxVecs is
         // the trivial operation

// vector product
   AxVec cross(const AxVec& a)const{ a; return AxVec();}
      // cross product or vector product of axial vectors. Since all
```

```
      // AxVecs are parallel, all cross products vanish
   Vec cross(Vec const& v)const;

   Angle phi()const{ return Angle(x3,RAD);}
      // returns the rotation angle belonging to *this.

   AxVec& phi_(Angle const& phi)
   { if (x3>=0) x3=phi; else x3=-phi; return *this;}
   //: phi
   // Changes *this into a rotation vector with a rotation angle
   // phi*sign(x3). So, in a sense, the function takes the sense of
   // rotation from *this and the rotation angle from the argument.
   // See the implementation of Path::quarterEllipse for a beneficial
   // usage.

   R normalize(){R res=cpmabs(x3); x3*=cpminv(res); return res;}

   AxVec ranVal(Z j=0)const;
      // absVal of the result is <= absVal of *this
   R disVal(const AxVec& a)const;
// linear algebra
   AxVec& operator +=(const AxVec& a){ x3+=a.x3; return *this;}
   AxVec& operator -=(const AxVec& a){ x3-=a.x3; return *this;}
   AxVec& operator *=(const R& r){ x3*=r; return *this;}
   AxVec& neg(void){ x3=-x3; return *this; }
   Z dim()const{ return 1;}
   R absVal()const{ return CpmRoot::absVal(x3);}
   R absSqr()const{return x3*x3;}
   friend AxVec operator +(const AxVec& a, const AxVec& b)
      { return AxVec(a.x3+b.x3);}
   friend AxVec operator -(const AxVec& a, const AxVec& b)
      { return AxVec(a.x3-b.x3);}
   friend AxVec operator -(const AxVec& a){ return AxVec(-a.x3);}
   friend AxVec operator *(const AxVec& a, const R& r)
      { return AxVec(a.x3*r);}
   friend AxVec operator *(const R& r, const AxVec& a)
      { return AxVec(r*a.x3);}
   friend R operator |(const AxVec& a, const AxVec& b)
      { return  a.x3*b.x3;}
   pair<R,R1> polDec()const;
      // polar decomposition: First component (res.first if res is the
      // return value) is the absolute value r and the second component
      // is sign(x3)

   pair<Angle,Vec> toAngAxs()const;
      // For uniformity with the corresponding function in cpmdim3.h
      // The returned vector is useless in this case and is chosen
      // as the zero vector
   bool isVal()const{ return CpmRoot::isVal(x3);}
```

```
   static V<AxVec> canBas()
         // canonical basis
      { return V<AxVec>("",AxVec(1.));}

   };

   RR2 toMat(const AxVec& d);
      // rotation matrix belonging to the axial vector. I.e.
      // x*toMat(d)=x*d, where on the left-hand side we have R2*RR2
      // (see cpmlinalg.h).

//////////////////////// class Vec ////////////////////////////////////////

   class Vec: public Rdim{ // space of translations and velocities
      friend class Group;
   public:
      static const Z dimension;
      // CPM_IO and CPM_ORDER from R2 !!!
      // also function R normalize()
      Word nameOf()const{ return Word("CpmDim2::Vec");}
      Vec():Rdim(){}
      Vec(const Rdim& a):Rdim(a){}
         // can't be explicit if the arithmetics definded for R2
         // is to be used without explicit conversions in Vec
      Vec(const R2& a, const RefSys& rs);
      Vec(R r1, R r2, R r3=0.):R2(r1,r2){r3;}
         // to enable code unification for 2 and 3 dimensions
      explicit Vec(const C& a):R2(a[1],a[2]){}
      explicit Vec(const V<R>& v):Rdim(v){}
      explicit Vec(AxVec const& ax){x1=0;x2=0;}
         // no meaningful initialization from AxVec in 2D
      R2 toRdim()const{ return R2(x1,x2);}
      C toC()const{ return C(x1,x2);}
      friend Vec operator *(Vec const& x, AxVec const& d);
         // x*d is the image of x under the active transformation
         // represented by d. This is the 'master representation' which
         // defines the geometrical meaning of d. The group
         // multiplication in AxVec is defined just such that
         // x*(d1*d2) = (x*d1)*d2.

// R& operator[](Z i) and the corresponding const version were defined
// till 2000-12-15 although these are defined from R2 as non-virtual.
// This seems to have created mysterious run-time errors but no compiler
// warning.

      Vec& operator *=(const AxVec& d){ return *this=(*this)*d;}

      AxVec cross(const Vec& a)const
         // vector product of vectors is an axial vector
      { return AxVec(crsPrd(a));}
```

```
Vec cross(const AxVec& a)const
   // vector product of a vector with an axial vector is a vector
{ return Vec(x2*a.x3,-x1*a.x3);}

Vec rot90()const{ return Vec(x2,-x1);}
   // rotation 90 (degrees)
   // Returns a vector which results from *this by a rotation
   // of 90 degrees in positive direction. I came to consider this
   // an important operation by studying lecture notes of Professor
   // Engelbert Schlipf (Fachhochschule Ulm) on technical mechanics.
   // January 2008

R spat(const Vec& a, const Vec& b)const{a; b; return 0;}
   // for uniformity with 3D

friend Vec operator -(const Vec& a){ return Vec(-a.x1,-a.x2);}

Vec& operator *=(const Group&);

Vec& mir_(Z i) // mirror (reflection), clear from code.
{
   if (i==1) x1=-x1;
   else if (i==2) x2=-x2;
   else ;
   return *this;
}

Vec& operator *=(const R& r){ R2::operator *=(r); return *this;}
   // see AxVec::operator*=(const R&)

X2<Vec,Vec> tanNormDec(const Vec& b)const;
   // the first component of the return pair is the component of b
   // in the direction of *this, and the second component is the
   // normal component

V<Vec> base(const Vec& b=Vec(0,1) )const;
   // returns a positively oriented pair of orthogonal unit vectors
   // e[1],e[2].
   // The argument b has no effect and is there only to have the
   // same interface as in the 3-dimensional case, where we need
   // two vectors for determining a ortho-normal base.
   // If *this is not zero, we have e[1]=Vec(1,0), e[2]=Vec(0,1).
   // Else, e[1] is the unit vector (*this)/|*this| and
   // e[2]=e[1]*AxVec(Angle(90)), which is the uniquely determined
   // supplement, for forming a positively oriented ortho-normal
   // base.

void setDir(const Vec& v){*this=v.polDec().second()*absVal();}
```

```
      AxVec rotTo(const Vec& v)const;
         // if the return value is interpreted as rotation, it is the
         // most direct one carrying *this into v.

      Angle angTo(const Vec& v)const{ return rotTo(v).phi();}
         // angle 'from *this to v'

      AxVec rotParOrAntiPar(const Vec& v);
         // name: rotate parallel or anti-parallel
         // The smallest rotation (i.e. the one with the absolute value
         // of the rotation angle) which makes *this parallel to the
         // the lines determined by {v,-v}

      Vec randomize(R dv, Angle const& a, Z j)const;
         // Returns a vector which is a stochastic
         // modification of a copy of *this. The length of the
         // result is uniformly distributed (as a function of j)
         // in the interval [max(0,|*this|-|dv|),|*this|+|dv|], and
         // the direction is obtained from the direction of *this
         // by rotating arround a random axis by an angle which is
         // uniformely distributed between -a and a.

      static Vec ranUnitVec(Z j=0);
         // random unit vector

      static V<Vec> canBas()
         // canonical basis
      { return V<Vec>("",Vec(1,0),Vec(0,1));}

      static V<Vec> homUnitVec(Z n, Z nSteps=10, Z stopPeriod=10,
         R dtFac=0.07, R eps=0.1);
         //: homogeneous unit vectors
         // generates a system of n unit vectors which are
         // first selected at random and then distributed more
         // homogeneously by applying nSteps steps of a
         // repulsive dynamical process. This is more important in
         // 3D than here, but for uniformity of methods its done
         // in 2D in the same manner. Implementation is identical
         // for 2D and 3D and is in cpmhomunitvecxpp.h.
     };

   inline AxVec::AxVec(const Angle& phi, const Vec& axis):x3(phi){axis;}
      // in order to avoid warnings from unreferenced formal parameters
      // one has to add the effect-less statement axis; this is not
      // allowed by the MS compiler to be placed directly in class AxVec,
      // since there Vec is not yet defined

//////////////////// class Group ////////////////////////////////////////
// see cpmdim3.h for explanation of most functions
```

```
   class Spc;
   class InTens;

   class Group{
// group of active Euclidian transformations (translations and rotations)

      friend class AxVec;
      friend class Vec;
      friend class Spc;
      friend class InTens;
      friend Spc operator *(const Spc& x, const Group& g);

      AxVec A;
      Vec a;

   public:
      typedef Group Type;
      CPM_INVERSION
      CPM_PRODUCT_M
      CPM_IO
      CPM_ORDER
      Word nameOf()const{ return Word("CpmDim2::Group");}
      Group(){}
         // unit element
      Group(const Spc& x0, const AxVec& A1);
         // constructor from fixed point and axial vector for
         // rotation arround that point. See cpmdim3.h for more.
      Group(const Spc& x0, const Angle& phi);
         // see cpmdim3.h

      Group(const AxVec& A1, const Vec& a1):A(A1),a(a1){}

      explicit Group(const AxVec& A1):A(A1),a(){}

      explicit Group(Angle const& alpha):A(AxVec(alpha)),a(){}
         // constructor from angle; this is usefull in context where
         // Group is available and AxVector not (as in my ParDynSys
         // template)

      explicit Group(const Vec& a1):A(),a(a1){}

      Vec shift(void)const{ return a;}

      AxVec rot(void)const{ return A;}

      friend Vec operator * (const Vec& x, const Group& g)
      { return x*g.A;}

      static Rdim act(const Rdim& x, const Group& g)
         // transforming a point via the fundamental transformation
```

```
         // group. Here the point is represented by its coordinate-
         // tuple with respect to the master system of reference
      { return Vec(x)*g.A+g.a;}

      F<Rdim,Rdim> actRdim()const
      { return F1<Rdim,Group,Rdim>(*this)(act);}

      R disVal(const Group& g)const
         { return CpmRootX::sup(A.disVal(g.A),a.disVal(g.a));}

      Group ranVal(Z j=0)const{ return Group(A.ranVal(j));}
         // see cpmdim3.h for explanation

      Group ranVal(Spc const& x0, Z j=0)const
      { return Group(x0, A.ranVal(j));}
         // random rotations arround x0 with rotation angle up to that of
         // *this. Useful in context where AxVec (which defines the
         // corresponding random sequence) is not available
   };

   inline Group& Group::operator *=(const Group& g)
   {
      A*=g.A; (a*=g.A)+=g.a; return *this;
   }

   inline Group Group::operator !()const
   {
      return Group(!A,(-a)*(!A));
   }

//////////////////////// class Spc ////////////////////////////////////////
// see cpmdim3.h for explanation of most functions

   class Spc{
// affine Euclidian space, instances are points and not vectors
      friend class Group;
      friend class RefSys;
      Vec p;
      explicit Spc(const Vec& p1):p(p1){}
      static R refLen;

   public:
      typedef Spc Type;
      CPM_IO
      CPM_ORDER
      Spc():p(){}
      Spc(R x1, R x2, R x3=0.):p(x1,x2){x3;}
         // x3 not active, for uniformity with cpmdim3.h

      explicit Spc(const Rdim& x):p(x[1],x[2]){}
```

```
explicit Spc(const V<R>& x):p(x[1],x[2]){}

Spc(const Rdim& x, const RefSys& rs);

explicit Spc(const C& a):p(a[1],a[2]){}

Spc(const V<R>& x, const RefSys& rs);
   // construction from components with respect
   // to the reference system rs, where the components
   // are represented by a dimension-independent type

C toC()const{ return p.toC();}

Rdim toRdim()const{ return p.toRdim();}

bool isVal()const{ return p.isVal();}

static Rdim toRdimStatic(const Spc& x)
   { return x.p.toRdim();}

static F<Spc,Rdim> FSR;

R& operator[](Z i){return p[i];}
const R& operator[](Z i)const{return p[i];}

Z dim()const{ return p.dim();}
Word nameOf(void)const{ return "CpmDim2::Spc";}

void setRefLen(R rl)
{ cpmassert(rl>0,"Spc::setRefLen()");refLen=rl;}
R getRefLen(){ return refLen;}

friend Spc operator * (const Spc& x, const Group& g)
{ return Spc(x.p*g.A+g.a);}


Spc& operator *=(const Group& g)
   // action of Group as a mutating operation
{ p=p*g.A+g.a; return *this;}

Spc& operator /=(const Group& g)
   // inverse action of Group as a mutating operation
   // for 'utmost efficiency' not written as *=!g
 { p-=g.a; p*=!g.A; return *this;}

Spc& rot_(Spc const& x0, AxVec const& a)
   //: rotation
   // rotation by a, which leaves the point x0 fixed
{ Vec pr=p-x0.p; p=x0.p+pr*a; return *this;}
```

```
Spc& rot(Spc const& x0, AxVec const& a){ return rot(x0,a);}
   // old name

friend Spc operator + (const Spc& x, const Vec& a)
   // shifting position by adding a vector
 { return Spc(x.p+a);}

friend Spc operator - (const Spc& x, const Vec& a)
   // shifting position by adding the vector -a
{ return Spc(x.p-a);}

Spc& operator +=(const Vec& a)
   // adding a vector as a mutating operation
{ p+=a; return *this;}

Spc& operator -=(const Vec& a)
   // subtracting a vector as a mutating operation
{ p-=a; return *this;}

friend Vec operator - (const Spc& x, const Spc& y)
   // getting a vector from subtracting two points
{ return Vec(x.p-y.p);}

Vec vecTo(const Spc& goal)const{ return Vec(goal.p-p);}
   // Vector which shifts *this into goal

Spc scl(Spc const& fix, R fac)const{ return fix+(*this-fix)*fac;}
   //: scale
   // Returns the point resulting from *this by
   // multiplying the position vector relative to
   // the point 'fix' by factor fac.

Spc& scl_(Spc const& fix, R fac){ return *this=scl(fix,fac);}

Spc cen(Spc const& x)const{ return Spc((p+x.p)*0.5);}
   // center (midpoint) between *this and x

static Spc convex(const V<Spc>& x, V<R>& w, bool careful=true);

static X2<Spc,R> envSph(V<Spc> const& vs);
   //: enveloping sphere
   // center and radius of an envelopping sphere of an array
   // of points

void convexAsMember(const V<Spc>& x, V<R>& w, bool careful=true)
{ *this=convex(x,w,careful);}
   // changes *this into the convex linear combination of the x

Spc versus(const Spc& a, R frac)const
   { return Spc(p+(a.p-p)*frac);}
```

```
   // returns a point which lies on the line from *this to a.
   // frac==0 gives *this, frac==1 gives a;  for values
   // outside [0,1] the meaning is an obvious generalization of
   // this

R disVal(const Spc& a)const;

R distance(const Spc& a)const // p1.distance(p2):=|p1-p2|
{  return (p-a.p).absVal();}

R distSquare(const Spc& a)const // p1.distSquare(p2):=|p1-p2|^2
{  return (p-a.p).absSqr();}


static R vol(R r){ return Pi*r*r;}
   // 'volume' of the 'sphere' (radius=r) in the space under
   // consideration. If CpmDim2::Spc is considered together with
   // CpmDim3::Spc, it is more  natural to speak of area of a
   // circular disk

static R sur(R r){ return 2*Pi*r;}
   // 'surface area' of the 'sphere' (radius=r) in the space under
   // consideration. If CpmDim2::Spc is considered together with
   // CpmDim3::Spc, it is more natural to speak of the length of
   // the circumference of a circular disk

static R radFrmVol(R ar){ return ar<=0 ? 0. : pow(ar/Pi,R(0.5));}
   // radius from 'volume' (of a 'sphere')

static R radFrmSur(R len ){ return len<=0 ? 0. : len/(2*Pi);}
   // radius from 'surface' (of a 'sphere')

static Z dimension(void){ return 2;}
   // dimension of the space. Avoids to have to say:
   // Spc x; Z d=x.dim()

static X4<Spc,R,R,R> meet(const Spc& p1, const Vec& v1,
   const Spc& p2, const Vec& v2);
   // Returns a point p such that there are real numbers x1,x2
   // such that p=p1+v1*x1=p2+v2*x2
   // Actually (p,0,x[1],x[2]) are returned as a quadruple.
   // The second less informative argument is
   // |p1+v1*x1-(p2+v2*x2)|/2 and is only numerical noise. In
   // the 3D case it is more interesting.

static pair<Spc,R> circle(const V<Spc>& x);
   // let x be a list consisting of 3 mutually different points.
   // We return center and radius of a circle on which
   // x[1], x[2], and x[3] lie.
   // The radius (=inverse of curvature) is given as positive
```

```
         // if the direction of x[3]-x[2] results from x[2]-x[1] by
         // a positve rotation (< 180 degrees absolute) and negative if
         // one has to rotate negatively (also <180 degrees in absolute
         // measure)

      static V<Spc> tet();
         //: tetraeder
         // flat tetraeder, tetraeder vertices projected to the plane
   };

   inline Group::Group(const Spc& x0, const AxVec& A1):A(A1)
   { Vec x0p=x0.p; a=x0p; a-=(x0p*=A1);}

   inline Group::Group(const Spc& x0, const Angle& phi):A(AxVec(phi))
   { Vec x0p=x0.p; a=x0p; a-=(x0p*=A);}

//////////////////////// class RefSys ////////////////////////////////////
// orthonormal, positively oriented coordinate system in Euclidean affine
// 2-space

class RefSys{
//orthonormal, positively oriented reference system in space

      friend class AxVec;
      friend class Vec;
      friend class Spc;

// inependent data
      Group g0;

// dependent data for efficiency
      Group g0Inv;
      Vec v1,v2;
      Spc x0;

      void update();

      static Vec u1,u2;
      static Spc orig;

   public:
      typedef RefSys Type;
      CPM_IO
      CPM_ORDER
      Word nameOf()const{ return Word("CpmDim2::RefSys");}
      RefSys();

      explicit RefSys(const Group& g0_):g0(g0_){update();}

      RefSys(const Spc& x0_, const AxVec& av);
```

```
      RefSys(const V<Spc>& vs);
         // assumption: vs.dim()>=Rdim
         // only the first Rdim components of vs get used.
         // vs[1] is the origin
         // e1:=(vs[2]-vs[1]), normalized, is the 1-axis
         // e2:=e1*AxVec(Angle(90,DEG)) is the 2-axis
         // If vs is too special for this to work, cpmassert will signal
         // the situation. In case that cpmassert is set not to stop but
         // only/ to warn (so that the function does not terminate) the
         // default RefSys gets constructed

      RefSys& operator *=(const Group g)
         { g0*=g; update(); return *this;}

      RefSys& operator +=(const Vec& a)
         { g0*=Group(a); update(); return *this;}

      RefSys& operator *=(const AxVec& A)
         { g0*=Group(A); update(); return *this;}

      RefSys& rotByEulAng(const Angle& phi, const Angle& theta,
         const Angle& psi);
         // rotate by Euler angles. Theta acts as if it were 0.
         // Defined only for uniformity with the 3D case

      RefSys per(const Word& p)const{p; return *this;}
      // permutation not useful in 2-dim, to be defined in order to have
      // the same interface as in the 3-dimensional case.
      // Therefore we return *this without any change.

// getting the components of objects with respect to the reference system
// *this
      R2 operator()(const Spc&)const;
      R2 operator()(const Vec&)const;
      R operator()(const AxVec&)const;
      Spc operator()(Rdim x)const{ return x0+x[1]*v1+x[2]*v2; }
      Vec operator[](Rdim x)const{ return x[1]*v1+x[2]*v2;}

      Group frmMasSys()const{ return g0;}  // from master system
      AxVec rot()const{ return g0.rot();}
      Vec shift()const{ return g0.shift();}

      Group toMasSys()const{ return g0Inv;} // to master system

      Group operator /(const RefSys&)const;

      Group toSys(const RefSys& rs)const{ return rs/(*this);}
         // More suggestive (at least at a first glance) notation
         // for the / operator.
```

```
         // RefSys rs1=...,rs2=...;
         // Group g=rs1.toSys(rs2);
         // bool b=(rs2==(rs1*=g)); // b==true always

// getting axes and origin associated with the RefSys
      Vec e1()const{ return v1;}
         // e1 and e2 are orthogonal and lie in the RefSys; they
         // determine the orientation of the RefSys
      Vec e2()const{ return v2;}
      Vec n()const{ return v2;}

      Spc o()const{ return x0;}
      V<Vec> getVectors()const{ return V<Vec>("",v1,v2);}
      CpmFunctions::F<Spc,Rdim> toFSR()const;
         // creates a CPM-function which transforms from Spc to
         // coordinates with respect to system *this
      CpmFunctions::F<Rdim,Spc> toFRS()const;
         // creates a CPM-function which transforms from coordinates
         // (with respect to system *this) to Spc
      };

//////////////////////// class Plane ////////////////////////////////////
// oriented affine hyperplane, which in this case is a straight line

   class Plane{
// oriented affine hyperplane, which in this case is a line

      RefSys rs;
         // the reference system in which *this is the 1-plane
   public:
      typedef Plane Type;
      CPM_IO
      Word nameOf()const{ return Word("CpmDim2::Plane");}
      Plane();
         // creates the 1-plane of the master system of reference
      explicit Plane(const RefSys& rs_):rs(rs_){}
         // creates a plane, which is the 1-plane of RefSys rs_

      Plane(const Vec& n, const Spc& x0);
         // creates a plane containing point x0 and having a normal unit
         // vector parallel to n (irrespective of normalization of n)
      R dist(const Spc& x)const{ return rs(x)[2];}
         // returns the signed distance of y from the plane. Positive
         // distance means positive 2-component with respect to a
         // positively oriented 'Zweibein' the 1-direction which is in
         // the plane
      Spc reflect(const Spc& x)const
         { R2 y=rs(x); y[2]=-y[2]; return Spc(y,rs);}
         // returns the mirror point of x with respect to the plane *this
      Vec reflect(const Vec& x)const
```

```
         { R2 y=rs(x); y[2]=-y[2]; return Vec(y,rs);}
         // returns the mirror image of vector x with respect to the
         // plane *this
      AxVec reflect(const AxVec& x)const{ return x;}
         // returns the mirror image of axial vector x with
         // respect to the plane *this
      Spc project(const Spc& x)const{ R2 y=rs(x); y[2]=0;
         return Spc(y,rs);}
         // returns the projected image of x on the plane *this
      Vec project(const Vec& x)const{ R2 y=rs(x); y[2]=0;
         return Vec(y,rs);}
         // returns the projected image of x on the plane *this
      Plane& operator *=(const Group g){ rs*=g; return *this;}
      Plane& operator +=(const Vec& a){ rs*=Group(a); return *this;}
      Plane& operator *=(const AxVec& A){ rs*=Group(A); return *this;}
      R2 operator()(const Spc& x){ return rs(x);}
      R2 operator()(const Vec& v){ return rs(v);}
      R1 operator()(const AxVec& av){ return (R1)rs(av);}

// getting directions associated with the plane
      Vec e1()const{ return rs.e1();}
      Vec e2()const{ return rs.e2();}
      Vec n()const{ return rs.e2();}
         // normal direction
      RefSys getRefSys()const{ return rs;}
      Word des()const;
         // description as a Word for diagnosis
   };

   class SubsetOfPlane{
// subset of a Plane  defined via an indicator function
         // stream I/O makes no sense for this class due to f being not
         // serializable
      RefSys rs;
         // the plane under consideration is the 12-plane of rs
      F<Rdim_,bool> f;
         // f(x1)==true says that a point x with rs-coordinates (x1,0)
         // belongs to the subset to be defined here.

   public:
      Word nameOf()const{ return Word("CpmDim2::SubsetOfPlane");}
      SubsetOfPlane(const RefSys& rs_, const F<Rdim_,bool>& f_):
         rs(rs_),f(f_){}
         // the default constructor thus creates the whole 12-plane of
         // the master system of reference

      SubsetOfPlane():rs(),f(true){}
      explicit SubsetOfPlane(const RefSys& rs_):rs(rs_),f(true){}

      pair<R,bool> ind(const Spc& x)const{ R2 y=rs(x);
```

```
         return pair<R,bool>(y[2],f(Rdim_(y[1])));}
         // returns the distance of x from the plane and the truth value
         // of the 'belonging to set'-relation for the projection of x
         // onto the plane

      bool indLean(const Spc& x)const
         { R2 y=rs(x); return f(Rdim_(y[1]));}
         // lean indicator
         // returns the truth value of the 'belonging to set'-relation
         // for the projection of x onto the plane

      SubsetOfPlane& operator *=(const Group g)
         { rs*=g; return *this;}
      SubsetOfPlane& operator +=(const Vec& a)
         { rs*=Group(a); return *this;}
      SubsetOfPlane& operator *=(const AxVec& A)
         { rs*=Group(A); return *this;}
   };

//////////////////// class InTens//////////////////////////////////////////
// inertial tensor, tensor of inertia

   class InTens{
// tensor of inertia, reduces in 2D to a scalar, the moment of inertia

      R I; // moment of inertia

   public:
      typedef InTens Type;
      CPM_IO
      Word nameOf()const{ return Word("CpmDim2::InTens");}

      InTens():I(0){}

      explicit InTens(R I_):I(I_){}

      explicit InTens(const RR3& tens):I(tens[3][3]){}
         // construction from a matrix representation, only the
         // 3,3-component counts

      AxVec operator * (const AxVec& a )const
         // inertia tensors transform axial vectors (eg. applied to
         // angular velocity to give angular momentum.
          {  return AxVec(a[3]*I); }

      void balance();
         // changes the inertia tensor *this into one which has equal
         // eigenvalues and the euclidean length of eigVal remains the
         // same. This is useful for plausibility controls.
```

```
   InTens& operator *=(const AxVec& a );
      // The definition is such that (I*a)*(x*a)=(I*x)*a

   InTens& operator *=(const Group& );

   InTens& mir_(Z i) // mirror (reflection), clear from code.
   {
       return *this;
   }

   InTens adjointAction(AxVec const& a)const;

   void scale(R fac){I*=fac;}
      // multiplying the eigenvalues by a common factor fac. No
      // dimension reflecting power of fac is taken.

   friend InTens operator !(const InTens& x);
      // inverse matrix, here implemented by inverting the eigen
      // values

   Vec e1()const{ return Vec();}
   Vec e2()const{ return Vec();}
   Vec e3()const{ return Vec();}
      // should not be relevant in 2D

   R i1()const{ return I;}
   R i2()const{ return I;}
   R i3()const{ return I;}
      // should not be relevant in 2D
};

InTens rotate(const InTens&, const AxVec& );
```

47 cpmdim2cpp.h

```
//? cpmdim2cpp.h
//? Status of work 2008-10-25.
//?
using namespace CpmStd;
//////////////////////////// class AxVec ////////////////////////////////

R& AxVec::operator[](Z i)
{
   if (i==3) return x3;
   else {
      cpmerror("CpmDim2::AxVec: index i="&cpmwrite(i)&
      " out of range");
      return x3;
   }
}

const R& AxVec::operator[](Z i)const
{
   if (i==3) return x3;
   else {
      cpmerror("CpmDim2::AxVec: index i="&cpmwrite(i)&
      " out of range");
      return x3;
   }
}

pair<Angle,Vec> AxVec::toAngAxs()const
{
   return pair<Angle,Vec>(Angle(x3,RAD),Vec());
}

AxVec operator *(const AxVec& a1, const AxVec& a2)
// addition of angles
{
   Angle phi1=a1.phi();
   Angle phi2=a2.phi();
   return AxVec(phi1+phi2);
}

namespace{
   R2 actionOfRotationVectors(R2 const& x, R const& a)
   {
      R co=::cos(a);
      R si=::sin(a);
      return R2(co*x.x1-si*x.x2,si*x.x1+co*x.x2);
   }
}
```

```
AxVec AxVec::adjointAction(const AxVec& d)const
{
   d; return *this; // AxVec is Abelian, thus adjoint action trivial
}

AxVec AxVec::exp(const R& t, bool accurate)const
{
   accurate;
   return AxVec(x3*t);
}

AxVec AxVec::pow(R p)const
{
   return AxVec(x3*p);
}

R AxVec::disVal(const AxVec& a)const
// method free of potential singularities
{
   Angle phi1=phi();
   Angle phi2=a.phi();
   return phi1.disVal(phi2);
}

RR2 toMat(const AxVec& d)
{
   R arc=d[3];
   R co=cos(arc);
   R si=sin(arc);
   RR2 res;
   res.x1.x1=co;
   res.x1.x2=-si;
   res.x2.x1=si;
   res.x2.x2=co;
   return res;
}

Z AxVec::com(const AxVec& a)const
{ return R1(x3).com(R1(a.x3));}

bool AxVec::scanFrom(istream& in)
{
   return CpmRoot::read(x3,in);
}

bool AxVec::prnOn(ostream& out)const
{
   if (!CpmRoot::writeTitle("AxVec",out)) return false;
   if (!CpmRoot::write(x3,out)) return false;
```

```
   return true;
}

AxVec::AxVec(const V<Vec>& b1, const V<Vec>& b2, bool careful)
{
   V<Vec> e1=b1;
   V<Vec> e2=b2;
   if (careful){
      e1=e1[1].base(e1[2]);
      e2=e2[1].base(e2[2]);
   }
   Vec vn=b2[1];
   C z(vn|b1[1],vn|b1[2]);
      // components of the new direction in the old basis
   *this=AxVec(Angle(z));
}

AxVec AxVec::ranVal(Z j)const
{
   R av=CpmRoot::ranVal(x3,j);
   return AxVec(av);
}

Vec AxVec::cross(Vec const& v)const
{
   return Vec(-x3*v[2],x3*v[1]);
}

//////////////////// class Vec ////////////////////////////////////////

Vec& Vec::operator *=(const Group& g )
   // The definition is such that (I*a)*(x*a)=(I*x)*a
{
   return *this*=g.A;
}

Vec operator *(const Vec& x, const AxVec& d)
// see actionOfRotationVectors()
{
   return Vec(actionOfRotationVectors(x,d()));
}

X2<Vec,Vec> Vec::tanNormDec(const Vec& a)const
{
   X2<R,R2> pdt=polDec();
   R2 t=pdt.second();
   Vec tanComp=t*(t|a);
   Vec normComp=a-tanComp;
   return X2<Vec,Vec>(tanComp,normComp);
}
```

```
V<Vec> Vec::base(const Vec& a)const
{
   // a not needed in 2 dimensions
   a;
   X2<R,R2> p=polDec();
   V<Vec> res(2);
   if (p.first()==0.){
      res[1]=Vec(1,0);
      res[2]=Vec(0,1);
   }
   else{
      Vec v1(p.second());
      res[1]=v1;
      res[2]=v1;
      AxVec rot90(Angle(90.));
      res[2]*=rot90;
   }
   return res;
}

AxVec Vec::rotTo(const Vec& v)const
{
   X2<R,R2> p1=polDec();
   if (p1.first()==0.) return AxVec();
   Vec e1=p1.second();
   AxVec rot90(Angle(90.));
   Vec e2=e1;
   e2*=rot90;
   C z((v|e1),(v|e2));
   return AxVec(Angle(z));
}

AxVec Vec::rotParOrAntiPar(const Vec& v)
{
  AxVec res1=rotTo(v);
  AxVec res2=rotTo(-v);
  return (res1.absVal()<=res2.absVal() ? res1 : res2);
}

Vec Vec::ranUnitVec(Z j)
{
   R phi=CpmRoot::randomR(j)*CpmRoot::Pi2;
   return Vec(cos(phi),sin(phi));
}

#include <cpmhomunitvecxcpp.h>
   // a piece of implementation which is common to 2D and 3D

//////////////////////////// class Spc ////////////////////////////////////
```

```
R Spc::refLen=1e-3;
   // a mm, this suitable for small particle applications

static R f(R x)
//f: [0,Infinity]-->[0,1], f'(0)=1
{
   static R PiHalf=0.5*CpmRoot::Pi;
   static R PiHalfInv=1/PiHalf;

   return PiHalfInv*atan(PiHalf*x);
}

F<Spc,Rdim> Spc::FSR=F<Spc,Rdim>(toRdimStatic);

R Spc::disVal(const Spc& a)const
   // p1.sistValue(p2)=(2/Pi)arctan((Pi/2)*|p1-p2|/refLen)
   // translation invariant function
   // Was a bad construct having a hidden reference to the
   // conventional length 1
   // (corresponding to 1 m in my applications till 2000-01-27)
{
   R x=(p-a.p).absVal()/refLen;
   return f(x);
}

Spc Spc::convex(const V<Spc>& x, V<R>& w, bool careful)
{
   Z n=x.dim(),i;
   cpmassert(n==w.dim(),"CpmDim2::convex");
   if (careful){
      R sum=0,val;
      for (i=1;i<=n;i++){
         val=w[i];
         if (val<0) val=-val;
         sum+=val;
         w[i]=val;
      }
      if (sum>0){
         sum=1./sum;
         for (i=1;i<=n;i++) w[i]*=sum;
      }
      else{
         cpmwarning("CpmDim2::convex: sum=0");
         R wi=1./n;
         for (i=1;i<=n;i++) w[i]=wi;
      }
   }
   Vec pSum;
   for (i=1;i<=n;i++) pSum+=x[i].p*w[i];
```

```
   return Spc(pSum);
}

X2<Spc,R> Spc::envSph(V<Spc> const& vs)
{
   Z n=vs.dim();
   if (n==0){
      Spc o;
      return X2<Spc,R>(o,0);
   }
   R nInv=1./n;
   Vec pSum;
   Z i;
   for (i=1;i<=n;i++) pSum+=vs.cui(i).p*nInv;
   Spc cen(pSum);
   Vo<R> rs(n);
   for (i=1;i<=n;i++) rs.cui(i)=(vs.cui(i)-cen).absVal();
   R r=rs.sup();
   return X2<Spc,R>(cen,r);
}

pair<Spc,R> Spc::circle(const V<Spc>& x)
{
   using namespace CpmArrays;
   Z mL=3;
   static Word loc("Spc::circle(const V<Spc>& x)");
   cpmmessage(mL,loc&" started");
   Z n=x.dim();
   cpmassert(n==3,loc);
   Angle w90(90.);
   AxVec rot90(w90);
   Spc p1=x[1];
   Spc p2=x[2];
   Spc p3=x[3];
   Vec v21=p2-p1;
   Vec v32=p3-p2;
   bool pos=v21.rotTo(v32).phi().isPos();
   Spc m21=p1+v21*0.5;
   Spc m32=p2+v32*0.5;
   v21*=rot90;
   v32*=rot90;
   X4<Spc,R,R,R> meet=Spc::meet(m21,v21,m32,v32);
   R_Vector d(3);
   Spc pRes=meet.c1();
   d[1]=(p1-pRes).absVal();
   d[2]=(p2-pRes).absVal();
   d[3]=(p3-pRes).absVal();
   R r= ( pos==true ? d.mean() : -d.mean() );
   cpmmessage(mL,loc&" done");
   return pair<Spc,R>(pRes,r);
```

```
}

V<Spc> Spc::tet()
// as in 3 dimensions but ignoring all z-coordinates
{
   const R h=::sqrt(3.)*0.5;
   const R hs=h/3;
   const R hl=hs*2;
   Rdim p1(hl,0);
   Rdim p2(-hs,0.5);
   Rdim p3(-hs,-0.5);
   Rdim p4(0,0);
   return V<Spc>("",Spc(p1),Spc(p2),Spc(p3),Spc(p4));
}

//////////////////////////// class Group ////////////////////////////////

////////////////////////// class RefSys ////////////////////////////////////

Vec RefSys::u1(1,0);
Vec RefSys::u2(0,1);

Spc RefSys::orig(0,0);

RefSys::RefSys() // g0,g0Inv initialized as unity, x0 as (0,0)
{
   v1=u1; // calling update() here would be unnecessarily complicated
   v2=u2;
}

RefSys::RefSys(const Spc& x0_, const AxVec& av)
// construct the RefSys which has x0_ as origin and the axes of
//which result from the master axes by applying the rotation
// described by the axial vector av
{
   Vec shift=x0_-orig;
   g0=Group(av,shift);
   update();
}

void RefSys::update()
{
   v1=u1*g0;
   v2=u2*g0;
   x0=orig*g0;
   g0Inv=!g0;
}

R2 RefSys::operator()(const Spc& x)const
{
```

```
   Vec v=x-x0;
   return R2(v|v1,v|v2);
}

R2 RefSys::operator()(const Vec& v)const
{
   return R2(v|v1,v|v2);
}

R RefSys::operator()(const AxVec& v)const
{
   return v[1];
}

Group RefSys::operator /(const RefSys& rs)const
{
   return rs.g0Inv*g0; // notice that g0 = rs.g0 * ( rs.g0Inv*g0) is the
   // relation to be satisfied by the expression in brackets
}

AxVec::AxVec(const R& a, const RefSys& rs)
{
   rs;
   x3=a;
}

Vec::Vec(const R2& a, const RefSys& rs)
{
   *this=a[1]*rs.v1+a[2]*rs.v2;
}

Spc::Spc(const R2& x, const RefSys& rs)
{
   *this=rs.x0+x[1]*rs.v1+x[2]*rs.v2;
}

Spc::Spc(const V<R>& x, const RefSys& rs)
{
   R2 a(x);
   *this=rs.x0+a[1]*rs.v1+a[2]*rs.v2;
}

RefSys& RefSys::rotByEulAng(const Angle& phi, const Angle& theta,
         const Angle& psi)
{
   theta; // no influence in 2D
   AxVec av(phi+psi);
   return operator *=(av);
}
```

```
//////////////////////// class Plane ////////////////////////////////////

Plane::Plane(){}

Plane::Plane(const Vec& n, const Spc& x)
{
   Spc x0;
   Vec a=x-x0;
   Vec e2(0,1);
   AxVec A=e2.rotTo(n);
   Group g(A,a);
   rs=RefSys(g);
}

Word Plane::des()const
{
   ostringstream ost;
   ost<<endl<<"Description of a instance of Plane:";
   ost<<endl<<"Origin is"<<rs.o();
   ost<<endl<<"e1() is"<<e1();
   ost<<endl<<"n() is"<<n();
   string res=ost.str();
   return Word(res);
}

/////////////////////////// class InTens ////////////////////////////////

InTens& InTens::operator *=(const AxVec& a )
   // The definition is such that (I*a)*(x*a)=(I*x)*a
{
   a; // no influence
   return *this;
}

InTens& InTens::operator *=(const Group& g )
   // The definition is such that (I*a)*(x*a)=(I*x)*a
{
   g; // no influence
   return *this;
}

InTens operator !(const InTens& x)
{
   R xI=x.I;
   cpmassert(xI>0.,"CpmDim2::operator !(InTens)");
   return InTens(1./xI);
}

InTens rotate(const InTens& I, const AxVec& a )
// far from optimum at least for symmetric I's
```

```
{
   InTens res=I;
   res*=a;
   return res;
}

//////////////// I/O for all classes ////////////////////////////////////

bool Spc::prnOn(ostream& out)const
{
   CpmRoot::writeTitle("Spc",out);
   return p.prnOn(out);
}

bool Spc::scanFrom(istream& in)
{
   return p.scanFrom(in);
}

bool Group::prnOn(ostream& out)const
{
   CpmRoot::writeTitle("Group",out);
   return (A.prnOn(out)&&a.prnOn(out));
}

bool Group::scanFrom(istream& in)
{
   return (A.scanFrom(in)&&a.scanFrom(in));
}

bool RefSys::prnOn(ostream& out)const
{
   CpmRoot::writeTitle("RefSys",out);
   return g0.prnOn(out);
}

bool RefSys::scanFrom(istream& in)
{
   bool res=g0.scanFrom(in);
   update();
   return res;
}

bool Plane::prnOn(ostream& out)const
{
   CpmRoot::writeTitle("Plane",out);
   return rs.prnOn(out);
}

bool Plane::scanFrom(istream& in)
```

```
{
   return rs.scanFrom(in);
}

bool InTens::prnOn(ostream& out)const
{
   CpmRoot::writeTitle("InTens",out);
   return CpmRoot::write(I,out);
}

bool InTens::scanFrom(istream& in)
{
   return CpmRoot::read(I,in);
}

void InTens::balance(){}
   // changes the inertia tensor *this into one which has equal
   // eigenvalues and the euclidean length of eigVal remains the same.
   // This is useful for plausibility controls

InTens InTens::adjointAction(AxVec const& a)const
{
   InTens res=*this;
   res*=a;
   return res;
}
```

48 cpmdim3.h

```
//? cpmdim3.h
//? Status of work 2008-10-25.
//?
/*

    Description:
      declares and partly defines classes describing geometrical objects
      which depend on the space dimension in a non-trivial manner.
      The idea is to select the actual version for use by a namespace
      using declaration:
      using namespace CpmDim3;
      will give the classes Vec, AxVec, InTens the form of classes
      describing polar vectors, axial vectors, tensors of inertia in
      3-dimensional space. Wheras
      using namespace CpmDim2;
      selects the proper implementation for 2-dimensional geometry.

      Contains some very efficient concepts of computational geometry
         which seem not to be widely (if ever) known.
*/

   class Group;
   class RefSys;
   class Vec;

//doc namespace CpmDim3{

//////////////////// class AxVec ////////////////////////////////////////

   class AxVec: public R3{
// axial vectors, rotation descriptors
      // space of 'axial vectors' i.e. rotations in
      // space and their infinitesimal generators. Interestingly, this is
      // an autonomous class (i.e. needs no support from other classes).
      // The vector space structure over R is inherited from base class.
      // If we interprete an axial vector e.g. as an angular velocity,
      // the the absolute value of the AxVec is dphi/dt.
      //
      // The natural combination of angular velocites corresponds to
      // relative rotations of frames: consider a rigid body rb_1
      // rotating with respect to the master system of reference with
      // angular velocity omega_1. Then consider a second rigid body rb_2
      // rotating with respect to rb_1 with an angular velocity omega_2.
      // Then rb_2 rotates with respect to the master system with an
      // angular velocity omega_1 + omega_2.
      //
      // If it is a finite rotation, the length is tan(phi/2).
```

```
   // This description of finite rotations is closely related to the
   // description by four-component Euler vectors (quaternions). I
   // never found this representation in the literature. Especially
   // the usefulness of this representation (as opposed to the
   // standard representation by orthogonal matrices) in computational
   // geometry seems to be not generally recognized so far.
   // Notice that AxVec defines all operations of a field (in the
   // sense of algebra) without being one: a*(x+y)!=a*x+a*b (i.e.
   // multiplication and addition are not distributive). Both
   // structures come from the Lie group structure of SO(3) endowed
   // with a canonical system of coordinates which approximately
   // canonical in the neighborhood of the unit element. As a Lie
   // group, SO(3) has a meaningful linear space structure near unity
   // and this is (artificially as far as the group is concerned)
   // extended over the whole space. Other axial vectors (e.g. angular
   // momentum, angular velocity) directly correspond to infinitesimal
   // rotations and for those, the linear structure is the primary
   // one.

   friend class Group;
   void make(R phi, const R3& n);
      // building block for two constructors of AxVec's corresponding
      // to finite rotations
public:
   // CPM_IO and CPM_ORDER from R3 !!!
   static const Z dimension;
      // in our case 3
   Word nameOf()const{ return Word("CpmDim3::AxVec");}

   AxVec(){};
      // AxVec with all components equal to zero

   AxVec(const R3& a, const RefSys& rs);
      // AxVec with given components with respect to the master system
      // of reference

   AxVec(const R3& a):R3(a){}
      // constructor from components respect to the master system of
      // reference

   AxVec(R r1, R r2, R r3):R3(r1,r2,r3){}
      // constructor from components respect to the master system of
      // reference

   explicit AxVec(R r3):R3(0,0,r3){}
      // uniformity with 2-dimensional case

   AxVec(Angle const& phi, R3 const& n);
      // constructor from rotation angle and rotation axis
      // n is a descriptor of the rotation axis
```

```
         // It has not to be a unit vector, but the corresonding unit
         // vector will be taken for describing the axis. The components
         // of n refer to the master system of reference

      AxVec(Angle const& phi, Vec const& axis);
         // constructor from rotation angle and rotation axis.
         // The Vec axis is a descriptor of the rotation axis
         // It has not to be a unit vector, but the corresonding unit
         // vector will be taken for describing the axis.

      AxVec(Angle const& phi, AxVec const& axis);
         // same as previous function but axis given more reasonably
         // as an axial vector

      explicit AxVec(Angle const& phi, Z i=3);
         // i=1,2,3 rotation arround i-axis (master system !) with angle
         // phi.
         // The skrew sense of the rotation is the standart one:
         // Looking against the axis vector, a positive phi indicates
         // counter-clock-wise rotation. The default is set such that
         // AxVec(phi) in 2-dimennsional geometry and in
         // three-dimensional geometry are consistent

      AxVec( V<Vec> const& b1, V<Vec> const& b2, bool careful=true);
         // constructs the finite rotation which transforms the
         // orthonormal basis b1 into the orthonormal basis b2.
         // If the third parameter has value true, one actually works
         // with potentially modified versions of b1 and b2 which are
         // guaranteed to be orthonormal triplets. Notice that this is
         // (appart from the default) the first constructor which does
         // not refer to a reference system.

// group structure, composition of rotations.
      AxVec operator *(const AxVec& a)const;
         // 'product of ternions'
      AxVec operator !()const{ return AxVec(-x1,-x2,-x3);}
         // !a is the inverse of a with respect to the multiplication
         // given above.
         // The default AxVec is the unit element of the group structure
         // of AxVec.
         // Notice that the inversion for the additive structure and the
         // multiplicative structures are the same and, consequently, the
         // neutral elements are the same too.
      AxVec& operator *=(const AxVec& a){ return *this=(*this)*a;}
         // group multiplication as a mutating operation. This hides all
         // R3::operator*=(...) irrespective of argument type ...
         // (which is const R& in our case).
         // Thus we have to define multiplication by R also in the scope
         // of AxVec if we don't want the user of this class to have to
         // be aware of its relation to R3 and make the suitable casts
```

```
         // for accessing this operation.

      AxVec& operator *=(const Group&);
         // here (*this) is considered a pure rotation and, hence, a
         // element of a subgroup of Group. The '*' referes to this
         // group.

      AxVec& mir_(Z i) // mirror (reflection), clear from code.
      {
         if (i==1) x1=-x1;
         else if (i==2) x2=-x2;
         else if (i==3) x3=-x3;
         else ;
         return *this;
      }

// operations for infinitesimal rotations and angular velocities

      AxVec& operator *=(const R& r){ R3::operator*=(r); return *this;}
         // Vector space operation. See comment on previous function

      AxVec exp(const R& t, bool accurate=true)const;
         // the standard exponential function from the Lie algebra
         // (to which *this belongs here) and the Lie group (return
         // value).
         // If omega is a constant angular velocity, then omega.exp(t) is
         // the finite rotation which results from having this angular
         // velocity be effective over the time span t.
         // If the last argument is false, we return the approximation
         // for small angles which is a.exp(t)=a*t*0.5

      AxVec pow(R p)const;
         // AxVec x=..; x.pow(3)=x*x*x etc. with the normal
         // generalization to rational and, final, real exponents.

      AxVec adjointAction(const AxVec& d)const;
         // In mathematical terminology this is the natural action of the
         // Lie group (to which argument d belongs) on its Lie algebra
         // (to which *this belongs). This natural action sometimes is
         // called the adjoint representation of the Lie group.
         // In Physics the most elementary appearence of this operation
         // is the natural transformation of angular velocities under
         // rotations

// vector product
      AxVec cross(const AxVec& a)const
         { return crsPrd(a);}
         // cross product or vector product of axial vectors. This is
         // related to the commutator of infinitesimal rotations:
         // Generally we have
```

```
      // d1*d2-d2*d1 = 2*(d1.cross(d2))/(1- (d1|d2))
      // And for infinitesimal transformations the nominator can be
      // neglected.

   Vec cross(Vec const& v)const;

   friend AxVec operator -(const AxVec& a)
      { return AxVec(-a.x1,-a.x2,-a.x3);}

   Angle phi()const;
      //: phi
      // returns the rotation angle belonging to *this: Range is [0,
      // Angle(piAppr)] piAppr=2*atan(hugeVal), where hugeVal is a
      // singularity management constant defined in cpmdim3.cpp,
      // presently set to 1e12

   AxVec& phi_(Angle const& phi);
      //: phi
      // Changes *this into a rotation vector which has as rotation
      // axis the unit vector given by *this and phi as the rotation
      // angle.
      // This is a useful operation in situations in which the axis
      // of rotation can be defined as a vector product of two vectors,
      // and the rotation angle has to be set from requirements which
      // can't be expressed in terms of these vectors.

   pair<Angle,Vec> toAngAxs()const;
      // returns a pair made out of rotation angle and rotation
      // axis (as unit vector)

   R disVal(const AxVec& a)const;
      // Here the function inherited from the base class is not
      // adequate

   AxVec ranVal(Z j=0)const;
      // returns a random value such that the rotation value of
      // the result is in [-|a|,|a|], where a is the rotation
      // angle of *this. The rotation axis of the result is
      // random and independent on the axis of *this.
      // (It employes Vec::ranUnitVec.)
   static V<AxVec> canBas()
   { return V<AxVec>("",AxVec(1,0,0),AxVec(0,1,0),AxVec(0,0,1));}
};

RR3 toMat(const AxVec& d);
   // rotation matrix belonging to the axial vector. I.e.
   // x*toMat(d)=x*d, where on the left-hand side we have R3*RR3
   // (see linalgtum.h). Although the matrix is less efficient in
   // acting on vectors compared to the right-hand side x*d, the
   // matrix is the only thing to describe easily the transformation
```

```
      // properties of tensors of rank 2 and higher.

//////////////////////// class Vec ////////////////////////////////////////

   class Vec: public Rdim{
// vectors, space of point differences and velocities
      friend class Group;
      friend class AxVec;
   public:
      static const Z dimension;
      // CPM_IO and CPM_ORDER from R3 !!!
      // also function R normalize()
      Word nameOf()const{ return Word("CpmDim3::Vec");}
      Vec():Rdim(){}

      Vec(const Rdim& a):Rdim(a){}
         // can't be explicit if the arithmetics definded for R3
         // is to be used without explicit conversions in Vec

      explicit Vec(AxVec const& ax){x1=ax.x1;x2=ax.x2;x3=ax.x3;}

      Vec(const R3& a, const RefSys& rs);

      Vec(R r1, R r2, R r3=0):R3(r1,r2,r3){}

      explicit Vec(const C& a):R3(a[1],a[2],0){}
      explicit Vec(const V<R>& v):R3(v){}
         // dimension independent input (now very useful since
         // V<R> is now a class that can be directly inputted
         // from class RecordHandler (added 2004-05-03)

      C toC()const{ return C(x1,x2);}

      R3 toRdim()const{ return R3(x1,x2,x3);}

      friend Vec operator *(const Vec& x, const AxVec& d);
         // x*d is the image of x under the active transformation
         // represented by d. This is the 'master representation' which
         // defines the geometrical meaning of d. The group
         // multiplication in AxVec is defined just such that
         // x*(d1*d2) = (x*d1)*d2.
         // Notice that for mappings R(d): Vec --> Vec, x |--> x*d
         // this implies
         // R(d1*d2)=R(d2)°R(d1), if ° denotes the conventional
         // composition of mappings:
         // (f°g)(x)=f(g(x))
         // For the 'contravariant composition' f&g, which is the natural
         // one for mappings written as acting 'from the right'
         // x(f&g)=(xf)g
         // we have
```

```
      // R(d1*d2)=R(d1)&R(d2)
      // Notice that we have x*(d1*d2) = (x*d1)*d2 only 'up to
      // roundoff errors'.
      // So x*(d1*d2*....dn)=(...((x*d1)*d2)*....*dn) with an
      // error which increases with n. The vector
      // yn:=(...((x*d1)*d2)*....*dn) will finally change its length
      // considerably with increasing n. On the other hand
      // xn:=x*(d1*d2*....dn) will always result from a single AxVec
      // on x and will thus have the same length as x up to roundoff
      // errors of a single function evaluation. Here we clearly see
      // the benefit from representing rotations by a set of
      // independent data. If we would use matrices (9 real
      // parameters, so 6 more than needed), these matrices have to be
      // orthogonal and each matrix multiplication would spoil the
      // orthogonality of the result by roundoff errors.
      // So we would have to implement a mechanism that restitutes
      // orthogonaliy if deviations have grown to some size.
      // Since enforcing orthogonality is computationally expensive,
      // the only staightforward strategy, namely to do the adjustment
      // after each multiplication, is hardly affordable.

   Vec& operator *=(const AxVec& d){ return *this=(*this)*d;}
      // x*=d resuls from x by applying rotation d

   AxVec cross(const Vec& a)const{ return crsPrd(a);}
      // vector product of vectors is an axial vector
      //(uses R3 CpmLinAlg::crsPrd(const R3&, const R3&))

  Vec cross(const AxVec& a)const{ return crsPrd(a);}
     // vector product of a vector with an axial vector is a vector

   R spat(const Vec& a, const Vec& b)const
      { return operator|(a.cross(b));}

   friend Vec operator -(const Vec& a)
      { return Vec(-a.x1,-a.x2,-a.x3);}

   Vec& operator *=(const Group&);

   Vec& mir_(Z i) // mirror (reflection), clear from code.
   {
      if (i==1) x1=-x1;
      else if (i==2) x2=-x2;
      else if (i==3) x3=-x3;
      else ;
      return *this;
   }

   Vec& operator *=(const R& r){ R3::operator *=(r); return *this;}
      // see AxVec::operator*=(const R&)
```

```
Vec rot90()const{ return Vec(x2,-x1,x3);}
   // rotation 90 (degrees)
   // Returns a vector which results from *this by a rotation
   // of 90 degrees arround the 3 axis in positive direction.
   // I came to consider this an important operation by studying
   // lecture notes of Professor Engelbert Schlipf
   // (Fachhochschule Ulm) on technical mechanics.

X2<Vec,Vec> tanNormDec(Vec const& b)const;
   // the first component of the return pair is the component of b
   // in the direction of *this, and the second component is the
   // normal component

V<Vec> base(const Vec& b=Vec(0,1,0) )const;
   // returns a positively oriented triplet of orthogonal unit
   // vectors e[1],e[2],e[3].  If *this is not zero, e[1] is
   // parallel to *this else e[1]=Vec(1,0,0).
   // If e[1] and b determine a plane, then e[1],b and e[1],e[2]
   // determine the same oriented plane. Otherwise, e[2] is the
   // first vector from the list Vec(1,0,0), Vec(0,1,0), Vec(0,0,1)
   // which is not proportional to e[1].

void setDir(const Vec& v){*this=v.polDec().second()*absVal();}
   //: set direction
   // gives *this the direction of v, without changing length

AxVec rotTo(const Vec& v)const;
   //: rotate to
   // if the return value is interpreted as rotation, it is the
   // most direct one carrying *this into v.

Angle angTo(const Vec& v)const{ return rotTo(v).phi();}
   //: angle to
   // angle 'from *this to v'

AxVec rotParOrAntiPar(const Vec& v);
   //: rotate parallel or anti-parallel
   // The smallest rotation (i.e. the one with the absolute value
   // of the rotation angle) which makes *this parallel to the
   // the lines determined by {v,-v}.

Vec randomize(R dv, Angle const& a, Z j)const;
   // Returns a vector which is a stochastic
   // modification of a copy of *this. The length of the
   // result is uniformly distributed (as a function of j)
   // in the interval [max(0,|*this|-|dv|),|*this|+|dv|], and
   // the direction is obtained from the direction of *this
   // by rotating arround a random axis by an angle which is
   // uniformely distributed between -a and a.
```

```
      static Vec ranUnitVec(Z j=0);
         //: random unit vector

      static V<Vec> canBas()
      { return V<Vec>("",Vec(1,0,0),Vec(0,1,0),Vec(0,0,1));}
         //: canonical basis
         // 'Base' seems not to be idiomatic here so I discontinue
         // writing base.

      static V<Vec> homUnitVec(Z n, Z nSteps=10, Z stoppingPeriod=10,
         R dtFac=0.07, R eps=1e-2);
         //: homogeneous unit vectors
         // generates a system of n unit vectors which are
         // first selected at random and then distributed more
         // homogeneously by applying nSteps steps of a
         // repulsive dynamical process. Implementation is identical
         // for 2D and 3D and is in cpmhomunitvecxpp.h.

     };

//////////////////// class Group ////////////////////////////////////////

   class Spc;
   class InTens;

class Group{
// group of active Euclidean transformations (translations and rotations)
      // Group of active transformations which defines
      // the 'physical geometry' under consideration.
      // All classes X that cooperate with this geometry
      // have to define have to define an action of G on X.
      // In physics, the action of groups on system states is normally
      // written as x |--> D(g)x for which a natural C++ formulation
      // would be x |--> g*x which would be implemented as
      // X operator *(const Group& g, const X& x).
      //
      // For classes from which one may wish to derive new classes
      // with aditional geometry-related data (e.g. particles with spin
      // derived from a class of scalar particle) one would like to
      // realize this as a virtual function so that 'incremental
      // programming' for the additional degrees of freedom becomes
      // possible and is supported by the programing language.
      // So, only a member function is the canonical implementation
      // of the group action. This means that Group has to act from
      // the right on X. I choose
      // X& X::operator *=(const Group& g);
      // (or also
      //    void X::operator *=(const Group& g); )
      // for the implementation of the group action. In order to let
```

```
      // such an operator access the private members of Group
      // all geometrical classes should be friends of Group. So the
      // following friend list has to be extended on occasion.
      friend class AxVec;
      friend class Vec;
      friend class InTens;
      friend class Spc;
      friend Spc operator * (const Spc& x, const Group& g);
// private data: same structure aso for 2 dimensions. Discrete symmetries
// (inversion) not present in Group (could be added in a derived group)

      AxVec A;
      Vec a;
         // we write (A,a) for a Group instance with these data
   public:
      typedef Group Type;
      CPM_INVERSION
      CPM_PRODUCT_M
      CPM_IO
      CPM_ORDER
      Word nameOf()const{ return Word("CpmDim3::Group");}

      Group(){}
         // unit element

      Group(const Spc& x0, const AxVec& A1);
         // constructor from fixed point and axial vector for
         // rotation arround that point.
         // inline definition in the Spc group, since Spc has to be
         // defined for implementation of this function.
         // x|--> x0 + (x-x0)*A1, thus x|--> (A1,x0-x0*A1)x
         // Group(x0,A1)=(0,-x0)*(A1,0)*(0,x0)
         // Here we wrote x0 also for the position vector of x0 in the
         // master frame.

      Group(const Spc& x0, const Angle& phi);
         // =Group(x0,AxVec(phi))

      Group(const AxVec& A1, const Vec& a1):A(A1),a(a1){}
         // constructor from axial vector and vector

      explicit Group(const AxVec& A1):A(A1),a(){}
         // constructor from axial vector

      explicit Group(Angle const& alpha):A(AxVec(alpha)),a(){}
         // constructor from angle; this is usefull in context where
         // Group is available and AxVector not (as in my ParDynSys
         // template)

      explicit Group(const Vec& a1):A(),a(a1){}
```

```
         // constructor from vector

      Vec shift(void)const{ return a;}
         // returns the shift (translation) part

      AxVec rot(void)const{ return A;}
         // returns the rotation part (the homogeneous transformation)

      friend Vec operator * (const Vec& x, const Group& g)
         // transforming a vector via the fundamental transformation
         // group
      { return x*g.A;}

      static Rdim act(const Rdim& x, const Group& g)
         // transforming a point via the fundamental transformation
         // group. Here the point is represented by its coordinate-
         // tuple with respect to the master system of reference
      { return Vec(x)*g.A+g.a;}

      F<Rdim,Rdim> actRdim()const
      { return F1<Rdim,Group,Rdim>(*this)(act);}

      R disVal(const Group& g)const
         { return CpmRootX::sup(A.disVal(g.A),a.disVal(g.a));}

      Group ranVal(Z j=0)const{ return Group(A.ranVal(j));}
         // random rotations arround ori with rotation angle up to that of
         // *this. Useful in context where AxVec (which defines the
         // corresponding random sequence) is not available.

      Group ranVal(Spc const& x0, Z j=0)const
      { return Group(x0, A.ranVal(j));}
         // random rotations arround x0 with rotation angle up to that
         // of *this. Useful in context where AxVec (which defines the
         // corresponding random sequence) is not available.
   };

   inline Group& Group::operator *=(const Group& g)
   {
      A*=g.A; (a*=g.A)+=g.a; return *this;
   }

   inline Group Group::operator !()const
   {
      return Group(!A,(-a)*(!A));
   }

//////////////////////// class Spc ////////////////////////////////////////

   class Spc{
```

```
// affine Euclidian space, instances are points and not vectors
      friend class Group;
      friend class RefSys;
      Vec p;
      explicit Spc(const Vec& p1):p(p1){}
      static R refLen;
         // reference length, a length to which the function disVal
         // referes in the sense that p1.disVal(p2) is |p1-p2|/refLen
         // for points the Euclidian distance of which is much smaller
         // than refLen.
   public:
      typedef Spc Type;
      CPM_IO
      CPM_ORDER

      Word nameOf()const{ return Word("CpmDim3::Spc");}

      Spc():p(){}

      Spc(R x1, R x2, R x3=0):p(x1,x2,x3){}
         // construction from components with respect
         // to the master system of reference, giving two components is
         // sufficient so there is uniformity with 2-dimensional case

      explicit Spc(const Rdim& x):p(x[1],x[2],x[3]){}
         // construction from components with respect
         // to the master system of reference

      explicit Spc(const V<R>& x):p(x[1],x[2],(x.dim()<3 ? 0 : x[3]) ){}

      Spc(const R3& x, const RefSys& rs);
         // construction from components with respect
         // to the reference system rs

      explicit Spc(const V<R>& x, const RefSys& rs);
         // construction from components with respect
         // to the reference system rs, where the components
         // are represented by a dimension-independent type

      explicit Spc(const C& a):p(a[1],a[2],0){}

      C toC()const{ return p.toC();}

      Rdim toRdim()const{ return p.toRdim();}

      bool isVal()const{ return p.isVal();}

      static Rdim toRdimStatic(const Spc& x)
      { return x.p.toRdim();}
```

```
static F<Spc,Rdim> FSR;

R& operator[](Z i){return p[i];}
const R& operator[](Z i)const{return p[i];}
   // access to the components with respect to the master system
   // of reference

Z dim()const{ return p.dim();}

void setRefLen(R rl)
{ cpmassert(rl>0,"Spc::setRefLen(R)");refLen=rl;}
R getRefLen(){ return refLen;}

friend Spc operator * (const Spc& x, const Group& g)
   // transforming a point via the fundamental transformation
   // group
{ return Spc(x.p*g.A+g.a);}

Spc& operator *=(const Group& g)
   // action of Group as a mutating operation
{ p=p*g.A+g.a; return *this;}

Spc& operator /=(const Group& g)
   // inverse action of Group as a mutating operation
   // for 'utmost efficiency' not written as *=!g
 { p-=g.a; p*=!g.A; return *this;}

Spc& rot_(Spc const& x0,AxVec const& a)
   //: rotation
   // rotation by a, which leaves the point x0 fixed
{ Vec pr=p-x0.p; p=x0.p+pr*a; return *this;}

Spc& rot(const Spc& x0, const AxVec& a){ return rot_(x0,a);}
   // old style

friend Spc operator + (const Spc& x, const Vec& a)
   // shifting position by adding a vector
 { return Spc(x.p+a);}

friend Spc operator - (const Spc& x, const Vec& a)
   // shifting position by adding the vector -a
{ return Spc(x.p-a);}

Spc& operator +=(const Vec& a)
   // adding a vector as a mutating operation
{ p+=a; return *this;}

Spc& operator -=(const Vec& a)
   // subtracting a vector as a mutating operation
{ p-=a; return *this;}
```

```
friend Vec operator - (const Spc& x, const Spc& y)
   // getting a vector from subtracting two points
{ return Vec(x.p-y.p);}

Vec vecTo(const Spc& goal)const{ return Vec(goal.p-p);}
   // Vector which shifts *this into goal

Spc cen(Spc const& x)const{ return Spc((p+x.p)*0.5);}
   //: center
   // returns the center (midpoint) between *this and x

Spc scl(Spc const& fix, R fac)const{ return fix+(*this-fix)*fac;}
   //: scale
   // Returns the point resulting from *this by
   // multiplying the position vector relative to
   // the point 'fix' by factor fac.

Spc& scl_(Spc const& fix, R fac){ return *this=scl(fix,fac);}

static Spc convex(const V<Spc>& x, V<R>& w, bool careful=true);
   // convex linear combination of the x.
   // Most important application: center-of-mass of a particle
   // system.
   // w is a family of as many positive weights (that add up to 1)
   // as there are members of x.
   // If the boolean parameter careful is set to true no
   // assumptions on w are made and w will be changed if positivity
   // and normalization are not satisfied on input.
   // The mode of enforcement is natural: Changing the sign of
   // negative w[i]'s and finally dividing by the sum of the
   // w[i]'s.
   // If careful is set to false, the w's will not be changed and
   // used as they are. The resulting point then may not belong to
   // the convex hull of the x[i]'s

static X2<Spc,R> envSph(V<Spc> const& vs);
   //: enveloping sphere
   // center and radius of an envelopping sphere of an array
   // of points

void convexAsMember(const V<Spc>& x, V<R>& w, bool careful=true)
{ *this=convex(x,w,careful);}
   // changes *this into the convex linear combination of the x

Spc versus(const Spc& a, R frac)const
   { return Spc(p+(a.p-p)*frac);}
   // returns a point which lies on the line from *this to a.
   // frac==0 gives *this, frac==1 gives a;  for values
   // outside [0,1] the meaning is an obvious generalization of
```

```
   // this

R disVal(const Spc& a)const;
   // p1.disVal(p2)=(2/Pi)arctan((Pi/2)*|p1-p2|/refLen)
   // translation invariant function.

R distance(const Spc& a)const // p1.distance(p2):=|p1-p2|
{  return (p-a.p).absVal();}

R distSquare(const Spc& a)const // p1.distSquare(p2):=|p1-p2|^2
{  return (p-a.p).absSqr();}


static R vol(R r){ return (4./3.)*Pi*r*r*r;}
   // volume of the sphere (radius=r) in the space under
   // consideration.

static R sur(R r){ return 4*Pi*r*r;}
   // surface area of the sphere (radius=r) in the space under
   // consideration.

static R radFrmVol(R vol)
   { return vol<=0 ? 0. : pow((3*vol)/(4*Pi),R(1.)/R(3.));}
   // radius from volume (of a sphere)

static R radFrmSur(R ar)
   { return ar<=0 ? 0. : pow(ar/(4*Pi),R(0.5));}
   // radius from surface (of a sphere)

static Z dimension(void){ return 3;}
   // dimension of the space. Avoids to have to say:
   // Spc x; Z d=x.dim()

static V<Spc> tet();
   //: tetraeder

static X4<Spc,R,R,R> meet(Spc const& p1, Vec const& v1,
   Spc const& p2, Vec const& v2);
   //: meet
   // We consider the lines r1 := { p1 + v1*a | a \in R },
   // r2 := { p2 + v2*a | a \in R }. Let res be the return
   // value of the function. Then for pc=res.c1();
   // rho := res.c2(); the sphere Shere(pc,rho) around
   // pc with radius rho is the smallest sphere which containst the
   // two points of closest approach of r1 and r2.
   // With a1:=res.c3(), a2:=res.c4(), these points are
   // q1=p1+v1*a1, q2=p2+v2*a2; thus pc=q1.cen(q2),
   // rho=|q1-q2|/2.
   // The natural situation to which this function applies is that
   // these two lines intersect and we are interested in the
```

```
         // crossing point. The second component of the result then allows
         // to assess the precision with which the lines hit each other.
         // In such cases it is natural that only solutions with
         // a1>0, a2>0 are intended ones.
   };

   inline Group::Group(const Spc& x0, const AxVec& A1):A(A1)
   { Vec x0p=x0.p; a=x0p; a-=(x0p*=A1);}

   inline Group::Group(const Spc& x0, const Angle& phi):A(AxVec(phi))
   { Vec x0p=x0.p; a=x0p; a-=(x0p*=A);}

///////////////////////// class RefSys ////////////////////////////////////
// orthonormal, positively oriented coordinate system in Euclidean affine
// 3-space

class RefSys{
//orthonormal, positively oriented reference system in space

      friend class AxVec;
      friend class Vec;
      friend class Spc;

// independent data
      Group g0;
         // describes the transformation which generates RefSys *this
         // from the master system of reference
// dependent data for efficiency
      Group g0Inv;
      Vec v1,v2,v3;
         // triplet of axis unit vectors
      Spc x0;
         // actual origin

      void update();

      static Vec u1,u2,u3;
         // axis triplet of Master system
      static Spc orig;
         // origin of master system

   public:
      typedef RefSys Type;
      CPM_IO
      Word nameOf()const{ return Word("CpmDim3::RefSys");}
      RefSys();
         // creates the master system of reference. All other systema can
         // be created via group actions

      explicit RefSys(const Group& g0_):g0(g0_){update();}
```

```
RefSys(const Spc& x0_, const AxVec& av);
   // construct the RefSys which has x0_ as origin and the axes of
   // which result from the master axes by applying the rotation
   // described by the axial vector av

RefSys(const V<Spc>& vs);
   // assumption: vs.dim()>=Rdim
   // only the first Rdim components of vs get used.
   // vs[1] is the origin
   // e1:=(vs[2]-vs[1]), normalized is the 1-axis
   // e2:=(component of vs[3]-vs[1] which is orthogonal to e1,
   // normalized is the 2-axis
   // e3:=e1.cross(e2) is the 3-axis
   // If vs is too special for this to work, cpmassert will signal
   // the situation. In case that cpmassert is set not to stop but
   // only to warn (so that the function does not terminate) the
   // default RefSys gets constructed

RefSys& operator *=(const Group g)
   { g0*=g; update(); return *this;}
   // translation and rotation combined

RefSys& operator +=(const Vec& a)
   { g0*=Group(a); update(); return *this;}

RefSys& operator *=(const AxVec& A)
   { g0*=Group(A); update(); return *this;}

RefSys& rotByEulAng(const Angle& phi, const Angle& theta,
   const Angle& psi);
   // rotate by Euler angles
   // transforms the system by Euler angle rotations referring
   // to system *this, leaving the origin of *this unchanged.

RefSys& rot(const Angle& phi, Z i);
   // rotate
   // transforms the system by phi-rotation arround axis
   // i of *this, leaving the origin of *this unchanged.

RefSys per(const Word& p)const;
   // permutation
   // p may be one of the following words:
   // "123","231","312","1-32","32-1","-213" which correspond
   // directly to the 6 permutations of the set {1,2,3} if the
   // minus signs are ignored.
   // The return value is a reference system rs  which has the same
   // origin and the same set of axes as *this (up to a reversal of
   // at most one axis, this is indicated by the - sign), but
   // differs in the indexing of these axes:
```

```
         // Let any of the allowed p's be written as 'abc'
         // (e.g. a=1, b=-3, c=2 for the fourth entry in the list), then
         // the 1-axis of *this is the a-axis of rs, the 2 axis of *this
         // is the b-axis of rs, and the 3-axis of *this is the c-axis of
         // rs. If abc, as a permutation of {1,2,3}, is a transposition
         // (of negative parity in a more general context) one axis (or 3
         // axes) have to change sign in order for the permutation of
         // axes to be achievable by a rotation in space (notice that we
         // consider only reference systems which originate from the
         // master system by rotations and translations; reflections are
         // excluded).
         // There are much more possible permutations of axes if other
         // positions of the - sign and also three minus signs are taken
         // into account. Here a minimal set is selected that does not
         // need rotations of 180° (which depend on a regularization of a
         // singularity in our parameter space)

// getting the components of objects with respect to the reference system
// *this
      R3 operator()(const Spc&)const;
      R3 operator()(const Vec&)const;
      R3 operator()(const AxVec&)const;
// getting the objects from their components
      Spc operator()(Rdim x)const{ return x0+x[1]*v1+x[2]*v2+x[3]*v3;}
      Vec operator[](Rdim x)const{ return x[1]*v1+x[2]*v2+x[3]*v3;}

      Group frmMasSys()const{ return g0;}  // from master system
         // returns the transformation which creates *this from the
         // master system of reference (which is created by the default
         // constructor) In self-explanatory pseudo-code (pseudo-code
         // elements '...' and '==')
         // RefSys rs=...; Group g=rs.frmMasSys(); RefSys master;
         // master*=g;
         // master==rs;

      AxVec rot()const{ return g0.rot();}
         // evident

      Vec shift()const{ return g0.shift();}
         // evident

      Group toMasSys()const{ return g0Inv;} // to master system
         // returns the transformation which creates the master system of
         // reference from system *this
         // RefSys rs=...; Group g=rs.toMasSys(); RefSys master; rs*=g;
         // master==rs;

      Group operator /(const RefSys&)const;
         // Let r1 and r2 two RefSys's. Then there is a uniquely
         // determined Group instance g such that r2 is obtained from r1
```

```
         // by group action:
         // r2 equals r1*=g. We write r2/r1 for this g.
         // ( plausible since then (r1*g)/r1==g )
         // This establishes between RefSys and Group a relation which is
         // analogous to the relation between Spc and Vec which is
         // characteritic for affine geometry. Group g=r2/r1 corresponds
         // to Vec v=x2-x1 for x2+=v for Spc x1,x2. (Vec is a group,
         // wheras Spc is'nt)

      Group toSys(const RefSys& rs)const{ return rs/(*this);}
         // More suggestive (at least at a first glance) notation
         // for the / operator.
         // RefSys rs1=...,rs2=...;
         // Group g=rs1.toSys(rs2);
         // bool b=(rs2==(rs1*=g)); // b==true always

// getting axes and origin associated with the RefSys
      Vec e1()const{ return v1;}
         // e1 and e2 are orthogonal and lie in the RefSys; they
         // determine the orientation of the RefSys

      Vec e2()const{ return v2;}

      Vec e3()const{ return v3;}

      Vec n()const{ return v3;}
         // it is useful to have a dimension-independent name for the
         // 'last vector'

      Spc o()const{ return x0;}

      V<Vec> getVectors()const
         { return V<Vec>("",v1,v2,v3);}

      F<Spc,Rdim> toFSR()const;
         // creates a CPM-function which transforms from Spc to
         // coordinates with respect to system *this

      F<Rdim,Spc> toFRS()const;
         // creates a CPM-function which transforms from coordinates
         // (with respect to system *this) to Spc
   };

//////////////////////// class Plane ////////////////////////////////////

   class Plane{ // oriented affine plane
      RefSys rs;
         // the reference system in which *this is the 12-plane
   public:
      typedef Plane Type;
```

```
CPM_IO
Word nameOf()const{ return Word("CpmDim3::Plane");}

Plane();
   // creates the 12-plane of the master system of reference

explicit Plane(const RefSys& rs_):rs(rs_){}
   // creates a plane, which is the 1-2-plane of RefSys rs_

Plane(const Vec& n, const Spc& x0);
   // creates a plane containing point x0 and having a normal unit
   // vector parallel to n (irrespective of normalization of n)

R dist(const Spc& x)const{ return rs(x)[3];}
   // returns the signed distance of x from the plane. Positive
   // distance means positive 3-component with respect to a
   // positively oriented 'Dreibein' the 1-direction and
   // 2-direction of which are in the plane and follow after each
   // other in the orientation of the plane.

Spc reflect(const Spc& x)const
   { R3 y=rs(x); y[3]=-y[3]; return Spc(y,rs);}
   // returns the mirror point of x with respect to the plane *this

Vec reflect(const Vec& x)const
   { R3 y=rs(x); y[3]=-y[3]; return Vec(y,rs);}
   // returns the mirror image of vector x with respect to the
   // plane *this

AxVec reflect(const AxVec& x)const{ return x;}
   // returns the mirror image of axial vector x with respect to
   // the plane *this

Spc project(const Spc& x)const
   { R3 y=rs(x); y[3]=0; return Spc(y,rs);}
   // returns the projected image of x on the plane *this

Vec project(const Vec& x)const
   { R3 y=rs(x); y[3]=0; return Vec(y,rs);}
   // returns the projected image of x on the plane *this

Plane& operator *=(const Group g){ rs*=g; return *this;}
   // translation and rotation combined

Plane& operator +=(const Vec& a){ rs*=Group(a); return *this;}

Plane& operator *=(const AxVec& A){ rs*=Group(A); return *this;}

R3 operator()(const Spc& x){ return rs(x);}
```

```
      R3 operator()(const Vec& v){ return rs(v);}

      R3 operator()(const AxVec& av){ return rs(av);}

// getting directions associated with the plane
      Vec e1()const{ return rs.e1();}
         // e1 and e2 are orthogonal and lie in the plane; they determine
         // the orientation of the plane

      Vec e2()const{ return rs.e2();}

      Vec n()const{ return rs.n();}
         // normal direction

      RefSys getRefSys()const{ return rs;}

      Word des()const;
         // description as a Word for diagnosis
   };

   class SubsetOfPlane{
// subset of a Plane defined via an indicator function
         // stream I/O makes no sense for this class due to f being not
         // serializable
      RefSys rs;
         // the plane under consideration is the 12-plane of rs
      F<R2,bool> f;
         // f(x1,x2)==true says that a point x with rs-coordinates
         // (x1,x2,0) belongs to the subset to be defined here.

   public:
      Word nameOf()const{ return Word("CpmDim3::SubsetOfPlane");}
      SubsetOfPlane(const RefSys& rs_,const F<R2,bool>& f_):
         rs(rs_),f(f_){}
         // the constructor for a subset the 12-plane of system rs_ such
         // that the rs-coordinates (x1,x2,x3) of its points satisfy
         // just f(x1,x2)==true

      explicit SubsetOfPlane(const RefSys& rs_):rs(rs_),f(true){}
         // this constructor thus creates the whole 12-plane of the
         // system rs_

      SubsetOfPlane():rs(),f(true){}
         // the default constructor thus creates the whole 12-plane of
         // the master system of reference

      pair<R,bool> ind(const Spc& x)const  // indicator
      { R3 y=rs(x); return pair<R,bool>(y[3],f(R2(y[1],y[2])));}
         // returns the distance of x from the plane and the truth value
         // of the 'belonging to set'-relation for the projection of x
```

```
         // onto the plane

      bool indLean(const Spc& x)const  // lean indicator
      { R3 y=rs(x); return f(R2(y[1],y[2]));}
         // returns the truth value of the 'belonging to set'-relation
         // for the projection of x onto the plane

      SubsetOfPlane& operator *=(const Group g){ rs*=g; return *this;}
         // translation and rotation combined. Notice that f has not to
         // be transformed since it works on the components in the
         // 'body-fixed' system of reference

      SubsetOfPlane& operator +=(const Vec& a)
         { rs*=Group(a); return *this;}

      SubsetOfPlane& operator *=(const AxVec& A)
         { rs*=Group(A); return *this;}
   };

//////////////////// class InTens//////////////////////////////////////////
// inertial tensor, tensor of inertia

   class InTens{ // tensors of inertia
      // Space of symmetric tensors. Data refer to a spectral
      // representation. Here used as Tensors of Inertia.

      RR3 eigVec; // again trying to improve speed by having a
      // pointer-less data type here

      R3 eigVal;  // list of eigenvalues
      // |eigVal[1]|>=|eigVal[2]|>=|eigVal[3]|

   public:
      typedef InTens Type;
      CPM_IO

      InTens():eigVec(){eigVec[1][1]=1; eigVec[2][2]=1; eigVec[3][3]=1;
         eigVal[1]=1; eigVal[2]=1; eigVal[3]=1;}
         // standart orthonormal basis, eigenvalues all 1

      explicit InTens(R3 v):eigVec()
         // standart orthonormal basis, eigenvalues given by v
         {
            eigVec[1][1]=1; eigVec[2][2]=1; eigVec[3][3]=1;
            eigVal[1]=v[1]; eigVal[2]=v[2]; eigVal[3]=v[3];
         }

      InTens(RR3 const& );
         // construction from a matrix representation, the input matrix
         // will not be taken as it is, but a symmetrized version of it
```

```
   // will be created. So the most obvious consistency condition
   // for a inertia matrix will be enforced. Other conditions of
   // elements of a tensor of inertia (e.g. Huette(Springer,1991),
   // E31, between (9) and (10), will not be considered.

AxVec operator * (const AxVec& a )const;
   // inertia tensors transform axial vectors (eg. applied to
   // angular velocity to give angular momentum). Since we know the
   // eigen vectors and -values this can be expressed as a sum of
   // dyades

void balance();
   // changes the inertia tensor *this into one which has equal
   // eigenvalues and the Euclidean length of eigVal remains the
   // same. This is useful for plausibility checks

InTens& operator *=(const AxVec& a );
   // The definition is such that (I*a)*(x*a)=(I*x)*a

InTens adjointAction(AxVec const& a)const
   // non-mutating version of operator *=(const AxVec& a )
{ InTens res=*this; return res*=a;}

InTens& operator *=(const Group& g )
   // g acts only through the rotation component
{ return operator*=(g.rot());}

InTens& mir_(Z i) // mirror (reflection), clear from code.
{
   if (i>=1 && i<=3){
      eigVec[1][i]=-eigVec[1][i];
      eigVec[2][i]=-eigVec[2][i];
      eigVec[3][i]=-eigVec[3][i];
   }
   return *this;
}

void scale(R fac){eigVal*=fac;}
   // multiplying the eigenvalues by a common factor fac. No
   // dimension reflecting power of fac is taken. If, for instance,
   // the material system described by *this is geometrically
   // enlarged by a (linear) factor y and the density remains
   // constant, then fac=pow(y,5)

R3 volume(R m)const;
   // Let res be the return value. Then res[1], res[2], res[3] are
   // the sites of a homogeneous Quader of total mass m
   // which has the inertial tensor *this (res[i] is along axis
   // eigVec[i] i=1,2,3)
```

```
   InTens operator !()const;
      // inverse matrix, here implemented by inverting the eigen
      // values

   Vec e1()const{ return Vec(eigVec.x1);}
   Vec e2()const{ return Vec(eigVec.x2);}
   Vec e3()const{ return Vec(eigVec.x3);}

   R i1()const{ return eigVal.x1;}
   R i2()const{ return eigVal.x2;}
   R i3()const{ return eigVal.x3;}

};

InTens rotate(const InTens&, const AxVec& );
```

49 cpmdim3cpp.h

```
//? cpmdim3cpp.h
//? Status of work 2008-10-25.
//?
using namespace CpmArrays;
using namespace CpmFunctions;
using namespace CpmGeo;
   // using declarations/directives seem to work
   // only reliable if not separated from its scope by include mechanisms
   // so we did this not in cpmdim.cpp

//////////////////////////// class AxVec ////////////////////////////////
// some local tools:

namespace {

   const R hugeVal=1e12;
   // rotation vectors should never be longer than this
   // hugeVal=1e6 says that instead of rotating by 180° (which has a
   // singular representation) we rotate by an angle which deviates by
   // 2/hugeVal rad = 2e-6 rad = 0.41" from 180°. This is OK for
   // scientific applications.
   // The closer the rotation angle to 180°, the larger the roundoff
   // errors in calculating the action of this transformation on space.

   const R hugeValInv=1./hugeVal;
   const R piHalfAppr=atan(hugeVal);
   const R piAppr=2*piHalfAppr;

R tanHalf(R x)
// safe way of forming nonsingular tan(x/2)
{
   if (!(x<=Pi)) cpmerror("tanHalf(R x):x<=Pi required, x="&cpm(x));
   if (!(x>=-Pi)) cpmerror("tanHalf(R x):x>=-Pi required x="&cpm(x));
      // was x>-Pi till 2005-06-28. Running PaLa under Linux once
      // resulted in calling the function for x=-Pi which under Windows
      // never happened
   if (x==0.) return 0.;
   R xh=0.5*x;
   if (xh>piHalfAppr) xh=piHalfAppr;
   if (xh<-piHalfAppr) xh=-piHalfAppr;
   return tan(xh);
}

} // namespace

AxVec::AxVec(const Angle& phi, const R3& n)
{
```

```
   R3 unitAxis=n.polDec().second();
   R arc=phi;
      // now arc in (-pi,pi] , only for this range, forming the
      // half-angle has the desired effect
   R da=tanHalf(arc);
   x1=da*unitAxis.x1;
   x2=da*unitAxis.x2;
   x3=da*unitAxis.x3;
}

AxVec::AxVec(const Angle& phi, Z i)
{
   cpmassert(i>0 && i<4,"AxVec(Angle,Z)");
   R arc=phi;
   R da=tanHalf(arc);
   if (i==1){ x1=da; x2=0; x3=0; return;}
   if (i==2){ x1=0; x2=da; x3=0; return;}
   if (i==3){ x1=0; x2=0; x3=da; return;}
}

AxVec::AxVec(const Angle& phi, const Vec& axis)
// since we describe the object to be constructed by components
// with respect to the master system, it is OK to use the components
// of axis with respect to the master system
{
   R3 unitAxis=axis.polDec().second();
   R arc=phi;
      // now arc in (-pi,pi] , only for this range, forming the
      // half-angle has the desired effect
   R da=tanHalf(arc);
   x1=da*unitAxis.x1;
   x2=da*unitAxis.x2;
   x3=da*unitAxis.x3;
}

AxVec::AxVec(Angle const& phi, AxVec const& axis)
{
   R3 unitAxis=axis.polDec().second();
   R arc=phi;
      // now arc in (-pi,pi] , only for this range, forming the
      // half-angle has the desired effect
   R da=tanHalf(arc);
   x1=da*unitAxis.x1;
   x2=da*unitAxis.x2;
   x3=da*unitAxis.x3;
}

Angle AxVec::phi()const
{
  R av=absVal(); // tan(phi/2)
```

```
  if (av>=hugeVal) av=piAppr; else if (av>0) av=2*atan(av); else ;
  return Angle(av,RAD); // i.e. don't interpret the argument as degrees
   // RAD was missing till 2000-2-28
}

AxVec& AxVec::phi_(Angle const& phi)
{
   nor_();
   R arc=phi;
   R da=tanHalf(arc);
   return operator*=(da);
}

pair<Angle,Vec> AxVec::toAngAxs()const
{
   R3 x(x1,x2,x3);
   R av=x.normalize();
   if (av>=hugeVal) av=piAppr; else if (av>0) av=2*atan(av); else ;
   return pair<Angle,Vec>(Angle(av,RAD),Vec(x));
}

AxVec AxVec::operator *(const AxVec& a)const
{
   R nom=1-x1*a.x1-x2*a.x2-x3*a.x3;
      // Count of operations: 3M+3A
   R nomAbs=(nom>=0 ? nom : -nom);
      // A
   R fac=(nomAbs<=hugeValInv ? hugeVal : 1./nom);
      // D
   AxVec res;
   res.x1=(x1+a.x1-x2*a.x3+x3*a.x2)*fac;
   res.x2=(x2+a.x2-x3*a.x1+x1*a.x3)*fac;
   res.x3=(x3+a.x3-x1*a.x2+x2*a.x1)*fac;
      // 9M+9A
      // Total count: 12M+D+13A
   return res;
}

AxVec& AxVec::operator *=(const Group& g )
{
   return *this*=g.A;
}

Vec AxVec::cross(Vec const& v)const
{ return Vec(crsPrd(v));}

/*
Vec CpmDim3::cross(const AxVec& a, const Vec& v)
{ return Vec(a.crsPrd(v));}
*/
```

```
namespace{ // anon


/*
R3 actionOfRotationVectors(const R3& x, const R3& d)
// The result is just R(d)x in my old terminology. For |d|<<1 this
// implies:
// x*d=x+dx=x+2d.cross(x)=x+dphi*n.cross(x)  for d=tan(dphi/2)*n,  |n|=1.
// Thus if d has only a 3-component and x only a 1-component, dx has only
// the component d[3]*x[1]. This is what a dphi-rotation arround the
// 3-axis in the positive sense (the one given in the (1,2) plane as
// going from the 1-axis to the 2-axis).
// The comments to the following expressions give the 'big-O' behavior
// for large |d|; for small |d|, there is not even a quasi-problem.
// This algorithmic block will be used for AxVec(d) and x and return
// value being either Vec or AxVec. In order not to dublicate code,
// it appears here as a auxiliary function private to this file.
// Without introducing ad hoc subexpressions one has
// x*d := [(1-(d|d))*x +2*(x|d)*d + 2*vectorProduct(d,x)]/[1+(d|d)]
{
   R ha=d|d;         // O(|d|^2) for large |d|
      // multiplication (M), division (D) and addition (A) count:
      // 3M+2A
   if (ha==0.) return x;
   R hb=x|d;         // O(|d|)
      // 3M+2A
   R hc=2./(1.+ha);  // O(|d|^-2)
      // D+A
   R hd=0.5*(1.-ha); // O(d^2)
      // M+A
   R he=hc*hd;       // O(|d|^0)
      // M
   R3 ds=d*hc;    // O(|d|^-1)
      // 3M
   R r1=he*x.x1 + hb*ds.x1 + ds.x2*x.x3 - ds.x3*x.x2;
      // O(|d|^0)+O(|d|^0)+O(|d|^-1)-O(|d|^-1); thus no large terms for
      // large |d|
   R r2=he*x.x2 + hb*ds.x2 + ds.x3*x.x1 - ds.x1*x.x3;
   R r3=he*x.x3 + hb*ds.x3 + ds.x1*x.x2 - ds.x2*x.x1;
   // each: 4M+3A thus whole group: 12M+9A
   // Total count: 23M+1D+15A
   return R3(r1,r2,r3);
}
*/

R3 actionOfRotationVectors(const R3& x, const R3& d)
// This implementation is 6 Multiplications less effective
// as the optimum but saves 16M+1D+13A if the same
// d gets applied to one vector more. If the same d is applied
```

```
// to more vectors in succession with each vector this saving
// occurs again. For instance, rotating a whole basis is much more
// efficient this way.
{
   static bool firstrun=true;
   static R b11,b12,b13,b21,b22,b23,b31,b32,b33;
   static R d1,d2,d3,fac;
   bool setStatics; // will get initialized with certainty
   if (firstrun){
      setStatics=true;
      firstrun=false;
   }
   else{
      setStatics=d1!=d.x1||d2!=d.x2||d3!=d.x3;
   }
   if (setStatics){
      // all this stuff is saved if d is the same as in the last
      // call of this function
      d1=d.x1;
      d2=d.x2;
      d3=d.x3;
      R a11=d1*d1;
      R a12=d1*d2;
      R a13=d1*d3;
      R a22=d2*d2;
      R a23=d2*d3;
      R a33=d3*d3;
   // 6M

      R rr=a11+a22+a33;
      fac=2./(1+rr);
   // D+3A
      R c=(1-rr)*0.5;
   // M+A

      b11=a11+c;
      b12=a12+d3;
      b13=a13-d2;

      b21=a12-d3;
      b22=a22+c;
      b23=a23+d1;

      b31=a13+d2;
      b32=a23-d1;
      b33=a33+c;
      // 9A

      // Total in this block: 7M+1D+13A, only to be done if
      // d has a new value
```

```
   }
   return R3(
      (b11*x.x1+b21*x.x2+b31*x.x3)*fac,
      (b12*x.x1+b22*x.x2+b32*x.x3)*fac,
      (b13*x.x1+b23*x.x2+b33*x.x3)*fac
      );
      // 12M+6A
   // Total 19M+1D+19A (in worst case that d has a new value)
}

} // namespace

AxVec AxVec::adjointAction(const AxVec& d)const
{
   return AxVec(actionOfRotationVectors(*this,d));
}

AxVec AxVec::exp(const R& t, bool accurate)const
{
   if (!accurate){// return (*this)*0.5*t;
      AxVec res=*this;
      return res*=(0.5*t); // t was missing here till 2005-02-14
   }
   else{
      X2<R,R3> pd=polDec();
      R phiDot=pd.first();
      R3 axis=pd.second();
      Angle dphi(phiDot*t,RAD); // not in degrees!
      return AxVec(dphi,axis);
         // notice that n*r |--> n*tan(r*t/2) so that for small r
         // we have n*r |--> (n*r)*(t/2) which is the special
         // version for accurate==false
   }
}

AxVec AxVec::pow(R p)const
{
   X2<R,R3> pd=polDec();
   R3 axis=pd.second();
   Angle phi_=phi();
   phi_*=p;
   return AxVec(phi_,axis);
}

R AxVec::disVal(const AxVec& a)const
// method free of potential singularities
{
   static const Vec e1(1,0,0);
   static const Vec e2(0,1,0);
   static const Vec e3(0,0,1);
```

```
   static const R inv6=1./6.;
   R3 v1=e1*(*this); // R3 could be replaced by Vec
   R3 v2=e2*(*this);
   R3 v3=e3*(*this);
   R3 w1=e1*a;
   R3 w2=e2*a;
   R3 w3=e3*a;
   return
      ((v1-w1).absVal()+(v2-w2).absVal()+(v3-w3).absVal())*inv6;
      // <=1,
      // g1=*this=id and g2=total inversion e1|-->-e1, e2|-->-e2,
      // e3|-->-e3,
      // would give g1.disValue(g2)=1; however the total inversion is not
      // created by a AxVec so the value 1 will presumably not reached
      // within AxVec by considering (*this)*!a we could do with v1,v2,v3
      // instead of v1,v2,v3,w1,w2,w3 at the price of relaying on
      // AxVec*AxVec in addition to AxVec*Vec. Presently AxVec*AxVec has
      // to do a polar decomposition so that the present method should be
      // slightly faster (no big concern anyway!)
}

RR3 toMat(const AxVec& d)
      // rotation matrix belonging to the axial vector
      // looks quite efficient. The result is R(-d) in my old terminology
{

   R d11=d.x1*d.x1;
   R d22=d.x2*d.x2;
   R d33=d.x3*d.x3;
   R d2=d11+d22+d33;
   R ha=0.5*(1-d2);
   R hb=2./(1+d2); // OK without test for 0

   RR3 res;
// diagonal
   res.x1.x1=hb*(ha+d11);
   res.x2.x2=hb*(ha+d22);
   res.x3.x3=hb*(ha+d33);
// off-diagonal
   R dxy=d.x1*d.x2;
   res.x1.x2=hb*(dxy+d.x3);
   res.x2.x1=hb*(dxy-d.x3);
   dxy=d.x1*d.x3;
   res.x1.x3=hb*(dxy-d.x2);
   res.x3.x1=hb*(dxy+d.x2);
   dxy=d.x2*d.x3;
   res.x2.x3=hb*(dxy+d.x1);
   res.x3.x2=hb*(dxy-d.x1);
   return res;
}
```

```
namespace{

AxVec fromMat(const RR3& mat)
// natural return value would be R3
// Theory see: Werner Graeub: Lineare Algebra, Springer
// (Grundlehren...Band XCVII) 1958
// p. 147-148
{
   R asy23=mat[2][3]-mat[3][2];
   R asy13=mat[1][3]-mat[3][1];
   R asy12=mat[1][2]-mat[2][1];
   R3 a(-asy23,asy13,-asy12);
   R trace=mat[1][1]+mat[2][2]+mat[3][3];
   R cs2=trace-1; // 2*cos(phi)
   R si2=a.absVal(); // 2*sin(phi)
   Angle phi(C(cs2,si2));
   return AxVec(phi,a);
      // a needs not to be normalized
}

} // namespace

AxVec::AxVec(const V<Vec>& b1, const V<Vec>& b2, bool careful)
{
   V<Vec> e1=b1;
   V<Vec> e2=b2;
   if (careful){
      e1=e1[1].base(e1[2]);
      e2=e2[1].base(e2[2]);
   }
   RR3 mat;
   for (Z i=1;i<=3;i++){
      for (Z j=1;j<=3;j++){
         mat[i][j]=e1[i]|e2[j];
      }
   }
   AxVec prel=fromMat(mat);
   x1=prel[1]*e1[1][1]+prel[2]*e1[2][1]+prel[3]*e1[3][1];
   x2=prel[1]*e1[1][2]+prel[2]*e1[2][2]+prel[3]*e1[3][2];
   x3=prel[1]*e1[1][3]+prel[2]*e1[2][3]+prel[3]*e1[3][3];
}



AxVec AxVec::ranVal(Z j)const
{
   static unsigned long int j0=137;
   Z i=(j!=0 ? j : j0++);
   R alpha=phi();
```

```
   Iv iv(alpha,-alpha);
   R alphai=iv.ranVal(i);
   Vec ni=Vec::ranUnitVec(i);
   return AxVec(Angle(alphai,RAD),ni);
}

//////////////////// class Vec ////////////////////////////////////////

Vec& Vec::operator *=(const Group& g )
   // The definition is such that (I*a)*(x*a)=(I*x)*a
{
   return *this*=g.A;
}

Vec operator *(const Vec& x, const AxVec& d)
// see actionOfRotationVectors()
{
   return Vec(actionOfRotationVectors(x,d));
      // was AxVec(...) till 2000-2-28
}

X2<Vec,Vec> Vec::tanNormDec(const Vec& a)const
{
   X2<R,R3> pdt=polDec();
   R3 t=pdt.second();
   Vec tanComp=t*(t|a);
   Vec normComp=a-tanComp;
   return X2<Vec,Vec>(tanComp,normComp);
}

V<Vec> Vec::base(const Vec& a)const
   // returns a positively oriented triplet of orthogonal unit vectors
   // e[1],e[2],e[3].
   // If *this is not zero, e[1] is parallel to *this
   // else e[1]=Vec(1,0,0).
   // If e[1] and b determine a plane, then e[1],b and e[1],e[2]
   // determine
   // the same oriented plane.
   // Otherwise, e[2] is the first vector from the list
   // Vec(1,0,0), Vec(0,1,0), Vec(0,0,1) which is not proportional to
   // e[1].

{
   static bool firstrun=true;
   static Vl<Vec> test(4); // valid indexes are 0,1,2,3
   static const R small=1e-9;
      // two vectors are considered to define a plane with sufficient
      // precision if the tangent of the Angle which they form is larger
      // than this
   if (firstrun){
```

```
      test[1]=Vec(1,0,0);
      test[2]=Vec(0,1,0);
      test[3]=Vec(0,0,1);
      firstrun=false;
   }
   V<Vec> e(3);
   X2<R,R3> p=polDec();
   if (p.first()>0) e[1]=p.second(); else e[1]=test[1];
   Z iTest=0;
   test[iTest]=a;
   while (iTest<4){
      X2<Vec,Vec> pa=e[1].tanNormDec(test[iTest]);
      R pafa=pa.first().absVal(); // length of part of test[iTest]
         // which is parallel to e[1]

      X2<R,R3> p2=pa.second().polDec(); // part of test[iTest]
         // which is normal to e[1]; we start with test[iTest]=a
         // and replace a by canonical basis vectors if it does not fit

      if (p2.first()>small*pafa){
         e[2]=p2.second(); // is a unit vector
         e[3]=e[1].crsPrd(e[2]);
         return e;
      }
      iTest++;
   }
// never reach  this point
   cpmerror("unexpected behavior in Vec::tripod(...)");
   return e;
}

AxVec Vec::rotTo(const Vec& v)const
{
   X2<R,R3> p1=polDec();
   if (p1.first()==0.) return AxVec();
   Vec e1=p1.second();
   Vec e3(e1.crsPrd(v));
   X2<R,R3> p3=e3.polDec();
   if (p3.first()==0.) return AxVec();
   e3=p3.second();
   Vec e2(e3.crsPrd(e1));
   C z((v|e1),(v|e2));
   return AxVec(Angle(z),e3);
}

AxVec Vec::rotParOrAntiPar(const Vec& v)
{
  AxVec res1=rotTo(v);
  AxVec res2=rotTo(-v);
  return (res1.absVal()<=res2.absVal() ? res1 : res2);
```

```
}

Vec Vec::ranUnitVec(Z j)
{
   static bool jIni=false;
   static CpmRoot::N j0;
   if (!jIni){
      j0=137;
      jIni=true;
   }
   R r1,r2,r3;
   R3 r;
   if (j==0){
Again:
      r1=CpmRoot::randomR();
      r2=CpmRoot::randomR();
      r3=CpmRoot::randomR();
      r1-=0.5;
      r2-=0.5;
      r3-=0.5;
      r=R3(r1,r2,r3);
      R len2=r.absSqr();
      if (len2>0.25 || len2==0) goto Again; else r.nor_();
   }
   else{
      Z jAct=j0*3+j;
   Again2:
      r1=CpmRoot::randomR(jAct++);
      r2=CpmRoot::randomR(jAct++);
      r3=CpmRoot::randomR(jAct++);
      j0+=3;
      r1-=0.5;
      r2-=0.5;
      r3-=0.5;
      r=R3(r1,r2,r3);
      R len2=r.absSqr();
      if (len2>0.25 || len2==0) goto Again2; else r.nor_();
   }
   return Vec(r);
}

#include <cpmhomunitvecxcpp.h>
   // a piece of implementation which is common to 2D and 3D

//////////////////////////// class Spc ////////////////////////////////////

R Spc::refLen=1e-3;
   // a mm, this suitable for small particle applications

static R f(R x)
```

```
//f: [0,Infinity]-->[0,1], f'(0)=1
{
   static R PiHalf=0.5*CpmRoot::Pi;
   static R PiHalfInv=1/PiHalf;

   return PiHalfInv*atan(PiHalf*x);
}

R Spc::disVal(const Spc& a)const
   // p1.sistValue(p2)=(2/Pi)arctan((Pi/2)*|p1-p2|/refLen)
   // translation invariant function
   // Was a bad construct having a hidden reference to the conventional
   // length 1
   // (corresponding to 1 m in my applications till 2000-01-27)
{
   R x=(p-a.p).absVal()/refLen;
   return f(x);
}

Spc Spc::convex(const V<Spc>& x, V<R>& w, bool careful)
{
   Z n=x.dim(),i;
   cpmassert(n==w.dim(),"CpmDim3::convex");
   if (careful){
      R sum=0,val;
      for (i=1;i<=n;i++){
         val=w[i];
         if (val<0) val=-val;
         sum+=val;
         w[i]=val;
      }
      if (sum>0){
         sum=1./sum;
         for (i=1;i<=n;i++) w[i]*=sum;
      }
      else{
         cpmwarning("CpmDim3::convex: sum=0");
         R wi=1./n;
         for (i=1;i<=n;i++) w[i]=wi;
      }
   }
   Vec pSum;
   for (i=1;i<=n;i++) pSum+=x[i].p*w[i];
   return Spc(pSum);
}

F<Spc,Rdim> Spc::FSR=F<Spc,Rdim>(toRdimStatic);

X2<Spc,R> Spc::envSph(V<Spc> const& vs)
{
```

```
   Z n=vs.dim();
   if (n==0){
      Spc o;
      return X2<Spc,R>(o,0);
   }
   R nInv=1./n;
   Vec pSum;
   Z i;
   for (i=1;i<=n;i++) pSum+=vs.cui(i).p*nInv;
   Spc cen(pSum);
   Vo<R> rs(n);
   for (i=1;i<=n;i++) rs.cui(i)=(vs.cui(i)-cen).absVal();
   R r=rs.sup();
   return X2<Spc,R>(cen,r);
}

V<Spc> Spc::tet()
{
   const R h=::sqrt(3.)*0.5;
   const R hz=::sqrt(2./3);
   const R hs=h/3;
   const R hl=hs*2;
   Rdim p1(hl,0,0);
   Rdim p2(-hs,0.5,0);
   Rdim p3(-hs,-0.5,0);
   Rdim p4(0,0,hz);
   return V<Spc>("",Spc(p1),Spc(p2),Spc(p3),Spc(p4));
}

//////////////////////// class RefSys ////////////////////////////////////

Vec RefSys::u1(1,0,0);
Vec RefSys::u2(0,1,0);
Vec RefSys::u3(0,0,1);
Spc RefSys::orig(0,0,0);

RefSys::RefSys() // g0,g0Inv initialized as unity, x0 as (0,0,0)
{
   v1=u1; // calling update() here would be unnecessarily complicated
   v2=u2;
   v3=u3;
}

RefSys::RefSys(const Spc& x0_, const AxVec& av)
         // construct the RefSys which has x0_ as origin and the axes of
         // which
         // result from the master axes by applying the rotation
         // described
         // by
         // the axial vector av
```

```
{
   Vec shift=x0_-orig;
   g0=Group(av,shift);
   update();
}

void RefSys::update()
{
   v1=u1*g0;
   v2=u2*g0;
   v3=u3*g0;
   x0=orig*g0;
   g0Inv=!g0;
}

R3 RefSys::operator()(const Spc& x)const
{
   Vec v=x-x0;
   return R3(v|v1,v|v2,v|v3);
}

R3 RefSys::operator()(const Vec& v)const
{
   return R3(v|v1,v|v2,v|v3);
}

R3 RefSys::operator()(const AxVec& v)const
{
   return R3(v|v1,v|v2,v|v3);
}

Group RefSys::operator /(const RefSys& rs)const
{
   return rs.g0Inv*g0;
      // notice that g0 = rs.g0 * ( rs.g0Inv*g0) is the relation
      // to be satisfied by the expression in brackets
}

RefSys RefSys::per(const Word& p)const
{
   static const R fac3=1./sqrt(3.);

   if (p=="123") return *this;
   R angleDeg;
   R3 axis;
   if (p=="231"){
      axis=(e1()+e2()+e3())*fac3;
      angleDeg=120.;
   }
   else if (p=="312"){
```

```
      axis=(e1()+e2()+e3())*fac3;
      angleDeg=-120.;
   }
   else if (p=="1-32"){ // transposition 3<-->2, 1 fixed
      axis=e1();
      angleDeg=90.;
   }
   else if (p=="32-1"){ // transposition 3<-->1, 2 fixed
      axis=e2();
      angleDeg=90.;
   }
   else if (p=="-213"){// transposition 1<-->2, 3 fixed
      axis=e3();
      angleDeg=90.;
   }
   else{
      Word mess="RefSys::per(...) argument ";
      cpmerror(mess&p&" is not valid");
   }
   Angle alpha(angleDeg);
   AxVec av(alpha,axis);
   Group grot(x0,av);
   RefSys res=*this;
   return res*=grot;
}

AxVec::AxVec(const R3& a, const RefSys& rs)
{
   x1=a[1]*(rs.v1)[1]+a[2]*(rs.v2)[1]+a[3]*(rs.v3)[1];
   x2=a[1]*(rs.v1)[2]+a[2]*(rs.v2)[2]+a[3]*(rs.v3)[2];
   x3=a[1]*(rs.v1)[3]+a[2]*(rs.v2)[3]+a[3]*(rs.v3)[3];
}

Vec::Vec(const R3& a, const RefSys& rs)
{
   *this=a[1]*rs.v1+a[2]*rs.v2+a[3]*rs.v3;
}

Spc::Spc(const R3& x, const RefSys& rs)
{
    *this=rs.x0+x[1]*rs.v1+x[2]*rs.v2+x[3]*rs.v3;
}

Spc::Spc(const V<R>& x, const RefSys& rs)
{
   R3 a(x);
   *this=rs.x0+a[1]*rs.v1+a[2]*rs.v2+a[3]*rs.v3;
}

RefSys& RefSys::rotByEulAng(const Angle& phi, const Angle& theta,
```

```
         const Angle& psi)
{
   AxVec phiv3(phi,v3);
   operator *=(Group(x0,phiv3));
   AxVec thetav1(theta,v1);
   operator *=(Group(x0,thetav1));
   AxVec psiv3(psi,v3);
   return operator *=(Group(x0,psiv3));
}


RefSys& RefSys::rot(const Angle& phi, Z i)
{
   AxVec phiv(phi,getVectors()[i]);
   return operator *=(Group(x0,phiv));
}


////////////////////////// class Plane ////////////////////////////////////

Plane::Plane(){}

Plane::Plane(const Vec& n, const Spc& x)
{
   Spc x0;
   Vec a=x-x0;
   Vec e3(0,0,1);
   AxVec A=e3.rotTo(n);
   Group g(A,a);
   rs=RefSys(g);
}

Word Plane::des()const
{
   ostringstream ost;
   ost<<endl<<"Description of a instance of Plane:";
   ost<<endl<<"Origin is"<<rs.o();
   ost<<endl<<"e1() is"<<e1();
   ost<<endl<<"n() is"<<n();
   string res=ost.str();
   return Word(res);
}

//////////////////////////// class InTens ////////////////////////////////

InTens& InTens::operator *=(const AxVec& a )
// The definition is such that (I*a)*(x*a)=(I*x)*a
{
   eigVec.x1=actionOfRotationVectors(eigVec.x1,a);
   eigVec.x2=actionOfRotationVectors(eigVec.x2,a);
```

```
   eigVec.x3=actionOfRotationVectors(eigVec.x3,a);
   return *this;
}

AxVec InTens::operator * (const AxVec& a )const
{
   R a1=eigVal.x1*(eigVec.x1|a);
   R a2=eigVal.x2*(eigVec.x2|a);
   R a3=eigVal.x3*(eigVec.x3|a);
   Vec res=eigVec.x1*a1+eigVec.x2*a2+eigVec.x3*a3;
   return AxVec(res);
}

InTens InTens::operator !()const
{
   InTens res=*this;
   res.eigVal.x1=1./eigVal.x1;
   res.eigVal.x2=1./eigVal.x2;
   res.eigVal.x3=1./eigVal.x3;
   return res;
}

InTens rotate(const InTens& I, const AxVec& a )
// far from optimum at least for symmetric I's
{
   InTens res=I;
   return res*=a;
}

InTens::InTens(const RR3& g):eigVec()
// construction from a matrix representation
{
   bool testing=false;
      // no verbose testing
   using CpmArrays::R_Matrix;
   Word loc("InTens(const RR3&)");
   R_Matrix s(3,3);
   R_Matrix eigenVectors;
   Z i,j;
   for (i=1;i<=3;i++) for (j=1;j<=3;j++) s[i][j]=g[i][j];
   s=0.5*(s+s.con());
   R_Vector d=eigen(s, eigenVectors, testing);
   cpmassert(d[3]>0,loc);
   cpmassert(d[1]>=d[2],loc);
   cpmassert(d[2]>=d[3],loc);
   for (i=1;i<=3;i++) eigVal[i]=d[i];
   for (i=1;i<=3;i++) for (j=1;j<=3;j++) eigVec[i][j]=eigenVectors[i][j];
}

R3 InTens::volume(R m)const
```

```
   // Let res be the return value. Then res[1], res[2], res[3] are
   // the sites of a homogeneous Quader of total mass m which has the
   // inertial tensor *this (res[i] is along axis eigVec[i] i=1,2,3)
   // see my pages UM/Ariton 99-6-8/4-4
{
   Word loc("InTens::volume(R)");
   cpmassert(m>0,loc);
   R ev1=eigVal[1];
   R ev2=eigVal[2];
   R ev3=eigVal[3];
   R a=-ev1+ev2+ev3;
   cpmassert(a>0,loc);
   R b= ev1-ev2+ev3;
   cpmassert(b>0,loc);
   R c= ev1+ev2-ev3;
   cpmassert(c>0,loc);
   R fac=6./m; // gave compiler error in speed maximizing optimation
// mode
   a*=fac;
   b*=fac;
   c*=fac;
   a=::sqrt(a);
   b=::sqrt(b);
   c=::sqrt(c);
   return R3(a,b,c);
}

void InTens::balance()
   // changes the inertia tensor *this into one which has equal
   // eigenvalues
   // and the euklidean length of eigVal remains the same.
   // This is useful for plausibility controls
{
   R eps=eigVal.polDec().first();
   eps/=3.;
   eigVal.set(eps);
}

//////////////// I/O for all classes ////////////////////////////////////

bool Spc::prnOn(ostream& out)const
{
   CpmRoot::writeTitle("Spc",out);
   return p.prnOn(out);
}

bool Spc::scanFrom(istream& in)
{
   return p.scanFrom(in);
}
```

```
bool Group::prnOn(ostream& out)const
{
   CpmRoot::writeTitle("Group",out);
   return (A.prnOn(out)&&a.prnOn(out));
}

bool Group::scanFrom(istream& in)
{
   return(A.scanFrom(in)&&a.scanFrom(in));
}

bool RefSys::prnOn(ostream& out)const
{
   CpmRoot::writeTitle("RefSys",out);
   return g0.prnOn(out);
}

bool RefSys::scanFrom(istream& in)
{
   bool res=g0.scanFrom(in);
   update();
   return res;
}

bool Plane::prnOn(ostream& out)const
{
   CpmRoot::writeTitle("Plane",out);
   return rs.prnOn(out);
}

bool Plane::scanFrom(istream& in)
{
   return rs.scanFrom(in);
}

bool InTens::prnOn(ostream& out)const
{
   CpmRoot::writeTitle("InTens",out);
   return (eigVal.prnOn(out) && eigVec.prnOn(out) );
}

bool InTens::scanFrom(istream& in)
{
   return ( eigVal.scanFrom(in) && eigVec.scanFrom(in) );
}
```

50 cpmdimdef.h

```
//? cpmdimdef.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_DIMDEF_H_
#define CPM_DIMDEF_H_
/*********************************************************
*   Author: Ulrich Mutze                                 *
*   Copyright (c) 1999 Heidelberg Digital Finishing GmbH  *
*   All rights reserved                                  *
*                                                        *
*********************************************************
   cpmdimdef.h
   Description: In this file one defines wether the
   model to be built will be 2D or 3D
   Cpm = Classes for Physics and Mathematics || C+- (C plus minus)
         -           -           -                    - -    -
*********************************************************************/

#define CPM_DIMDEF2
#define CPM_DIMDEF3
   // on my new computer (256 MB RAM) there is no problem for having both
   // defined

#endif
```

51 cpmdimx.h

```
//? cpmdimx.h
//? Status of work 2008-10-25.
//?
/*

    Description:
      declares and partly defines those geometricl objects which,
      though depending on dimension (as Spc, Vec ... do) but can
      be coded in a way not depending essentially on dimension

*/
using CpmRoot::Rh;
using CpmFunctions::FncObj;
using CpmFunctions::F1;
using CpmFunctions::F2;

using CpmArrays::X2;
using CpmArrays::X3;
using CpmArrays::VV;
using CpmGraphics::VecGraph;

//doc namespace CpmDimx{

//////////////////////// function grad ///////////////////////////

   Vec grad(const Spc& x, const F<Spc,R>& f, const R& h);
   // gradient of a scalar function as a vector-valued function of
   // position. h is the distance used for finite differencing

/////////////////////////// class BndSph ///////////////////////////

   namespace{ // technicalities
      const Z nFindDef=500;
         // default for the number of points to find in a
         // SubsetOfSpace for basing an estimation of volume or diameter
         // on
      const Z nMaxDef=10000;
         // default for the maximum number of points to try
         // to find points in a SubsetOfSpace
   }

   class BndSph{ // BndSph = Bounding Sphere
   // an abstraction from a specific tool that helped to implement
   // effectively the function hit in class SubsetOfSpace

   Spc x;
      // center of a sphere
```

```
   R r;
      // radius of a sphere

public:

   Spc get_x()const{ return x;}
   R get_r()const{ return r;}

   BndSph(Spc x_,R r_):x(x_),r(r_){}
   BndSph():r(0){}
   BndSph operator|(const BndSph& bs)const;
      // minimum sphere which contains the set union of
      // *this and bs
   BndSph operator&(const BndSph& bs)const;
      // minimum sphere which contains the set intersection
      // *this and bs
   Iv hit(const Spc& x, const Vec& v)const;
      // {x+a*v| a in Iv} is the intersection of *this with the
      // ray { x+a*v| a>=0}
      // v has to be a unit vector

   X2<V<Spc>,Z> ran(Z n, const F<Spc,bool>& cond,
      Z tryMax=nMaxDef)const;

   X2<V<Spc>,Z> ran(Z n, Z tryMax=nMaxDef)const
   {
      F<Spc,bool> cond(true);
     // return ran(n,cond,nMaxDef);
      return ran(n,cond,tryMax);
         // changed 2006-06-22 old form seems to be nonsense
   }
      // returns a random sequence of n uniformely distributed
      // points in the interior of the bounding sphere.
      // The second argument specifies a condition that is guaranteed to
      // evaluate to true for the points of the returned list.
      // Further we return the number of points which were tried.
      // we never try more than tryMax points. If after trying tryMax
      // points we did not find n valid points for the return list, we
      // return the (shorter) list of the valid points found so far.
      // Unfortunately the gnu compiler does not understand the natural
      // version with default arguments
      //  pair<V<Spc>,Z> ran(Z n, const F<Spc,bool>& cond=
      //    F<Spc,bool>(true), Z tryMax=nMaxDef)const;

   X2<Z,Z> ranLean(Z n, const F<Spc,bool>& cond,
      Z tryMax=nMaxDef)const;

   X2<Z,Z> ranLean(Z n, Z tryMax=nMaxDef)const
   {
      F<Spc,bool> cond(true);
```

```
     // return ranLean(n,cond,nMaxDef);
      return ranLean(n,cond,tryMax);
          // changed 2006-06-22 old form seems to be nonsense
   }
      // similar to function ran. Let (v,n) be the return value
      // of ran, then (v.dim(),n) is the return value of the present
      // function.

   R vol()const{ return Spc::vol(r);}
};

//////////////// struct DisPar ////////////////////////////////////////

   struct DisPar{  // discretization parameters
      // the discretization parameters need not be set
      // when Camera::show(const SubsetOfSpace&,...) is
      // used for visualization. Quantities corresponding to
      // ds and dr are set from the pixel size of the camera
      R1 dr;
         // small length compared to the whole object, e.g.
         // used as step length in numerical differentiation
      R1 ds;
         // characteristing length for sampling tasks
      Z1 nFind;
      Z1 nMax;
      void ini(){nFind=nFindDef; nMax=nMaxDef;}
      typedef DisPar Type;
      CPM_IO
      DisPar(R dr_, R ds_):dr(dr_),ds(ds_){ini();}
      DisPar(){ini();}
   };

//////////////////////// class SubsetOfSpace //////////////////////////

   class SubsetOfSpace{ // subsets of space with boolean operations
   // (union, section, ... of sets)
   // Visualisation by Camera::show(const SubsetOfSpace&,...)
   // and via
   // Scalarfield f(aSubsetOfSpace);
   // f.mark(aCamera)
   // (this is much faster and allows to locate even objects which
   // may not be found by Camera::show(const SubsetOfSpace&,...)).
   // See also cpmshapesx.h for derived classes that have to be
   // visalized by just the same means.

   F<Spc,R> f;
      // f(x)<0 says that a point x belongs to the interior of the subset
      // to be defined here. f(x)==0 characterizes the surface
      // and f(x)>0 the exterior. In some algorithms it helps if
      // |f(x)| is close to the distance from the surface
```

```
   // at least for points close to the surface.
   // If this is the case we say that f is 'locally metric'.
   // In the present implementation f is a dependent quantity
   // which is set by function update() from the independent data
   // fr and rs. In a previous version, fr and rs were not present
   // and the following outdated comment gave the motivation for
   // this: It is reproduced here in order to prevent me from falling
   // into this trap again:
   // 'Notice that here it makes little sense to replace this f by a
   // RefSys and a F<R3,R> since the boolean combinations of two
   // SubsetOfSpace's having different RefSys's would be unnatural to
   // define'.
   // Now a natural definition has been found (see implementation
   // of the boolean combinations). Introducing rs on this
   // level has the benefit that operator *=(const Group&) can be
   // implemented in a way that consecutive applications of *= don't
   // grow in execution time.
   // Although it is standard to describe the surface of a smooth
   // point set by a equation f(x)=0, the extension of this
   // description -as given here- seems not to be in use in
   // computational geometry.
   // Among the strong points of this description is the elegant
   // implementation of boolean ('set theoretical') combinations
   // of point sets (see implementation of operators &,|,-, and ^).

F<Rdim,R> fr;
   // f,fr, and rs are interrelated by f(x)=fr(rs(x)) for
   // all x of type Spc. (Notice that rs(x) is of type Rdim for all
   // x of type Spc). Thus fr describes the subset of space as a
   // subset of coordinate tuples referring to the actual
   // system of reference rs

RefSys rs;
   // system of reference to which fr refers. On this attribute an
   // efficient implementation of transformations by Group
   // (see operator *=(const Group&)) is based.

Spc x;
   // x and r belong together: Ideally BndSph(x,r) is a bounding
   // sphere for *this. The value of x and r has an influence
   // only on the result of the following functions:
   //    R improveBoundingSphere(...)
   //    R vol()const
   //    V<Spc> ran()const
   //    pair<Spc,Z> hitFast(...)const
   // In my present applications by far the most important effect
   // is made by the last function on which all my visualization
   // methods for SubseOfSpace's is based. This implies that
   // only the part of the point set which is in BndSph(x,r)
   // is visible for these methods. Since it might be useful
```

```
      // to have only a part of a point set in 'visual scope'
      // an instance of SubsetOfSpace is not considered corrupted
      // if BndSph(x,r) is not a bounding sphere for *this.
      // Therefore we allow a constructor to be public which does not
      // care about the relation between f and BndSph(x,r).
      // Also, we would like to define unbounded sets like half-spaces
      // or infinite plates for which BndSph only makes sense as
      // a visual scope

   R1 r;
     // see x

   DisPar dp;
      // DisPar = discretizing parameters
      // data helping algorithms that need to discretize
      // *this in some way

   void update();
      // adjusts f to rs and fr

public:
   typedef SubsetOfSpace Type;
   virtual SubsetOfSpace* clone()const
      { return new SubsetOfSpace(*this);}

   CPM_O_V
      // no stream input unless we have a constructor from streamable
      // date as in the derived classes defines in cpmshapesx.h

   virtual Word nameOf()const{ return Word("SubsetOfSpace");}

// constructors

   SubsetOfSpace():f(F<Spc,R>(1)),fr(F<Rdim,R>(1)){}
         // constructor of the void subset of Spc

explicit SubsetOfSpace(const F<Rdim,R>& fr_, const RefSys& rs_=RefSys(),
      const Spc& x_=Spc(), R r_=0, const DisPar& dp_=DisPar()):
         fr(fr_),rs(rs_),x(x_),r(r_),dp(dp_)
   { update();}
      // Defining the object by prescribing fr.
      // Does not make sure that x_,r_ constitutes
      // a bounding sphere of the set described by f_
      // see explanation of x for reasons that may legitimate
      // to make such a disorganized function public

explicit SubsetOfSpace(const F<Spc,R>& f_, const RefSys& rs_=RefSys(),
      const Spc& x_=Spc(), R r_=0, const DisPar& dp_=DisPar());
      // Defining the object by prescribing f.
      // The further comment of the previous function applies.
```

```
// access functions

   void setShape(const F<Rdim,R>& fr_, const RefSys& rs_=RefSys())
   // tool for defining f,fr,rs consistently.
   // Notice that x and r may not match the shape generated
   // by these settings. See comment to previous function
   {
      fr=fr_;
      rs=rs_;
      update();
   }

   void set_x(const Spc& x_){ x=x_;}
   void set_x(const Rdim& xr){ x=rs(xr);}
   void set_r(R r_){ r=r_;}
      // see comment to attribute x for reasons that allow this to be
      // public

   void set_dr(R dr){ dp.dr=dr;}
   void set_ds(R ds){ dp.ds=ds;}
   void set_nFind(Z nF){dp.nFind=nF;}
   void set_nMax(Z nM){dp.nMax=nM;}
   void set_dp(const DisPar& dp_){ dp=dp_;}

   Spc get_x()const{ return x;}
   R get_r()const{ return r;}
   RefSys getRefSys()const{ return rs;}

   Group frmMasSys()const{ return rs.frmMasSys();}
   Group toMasSys()const{ return rs.toMasSys();}

   R get_dr()const{ return dp.dr;}
   R get_ds()const{ return dp.ds;}
      // only usage so far in
      // void Camera::mark(const SubsetOfSpace& sos,
      // SitRec k, const Color& c)const
   Z get_nFind()const{ return dp.nFind;}
   Z get_nMax()const{ return dp.nMax;}
   DisPar get_dp()const{ return dp;}

   bool ind(const Spc& x)const{ return f(x)<=0;}  // indicator
   R operator()(const Spc& x)const{ return f(x);}

   Rdim refSys(const Spc& x)const{ return rs(x);}
   Spc refSys(const Rdim& x)const{ return rs(x);}
      // notice that there is no automatic conversion Rdim --> Spc
      // since the corresponding constructor is explicit

   R improveBoundingSphere(Z n=nFindDef, Z tryMax=nMaxDef);
```

```
   // A series of n randomly selected space points of *this will be
   // used to get an improved estimate for the bounding sphere.
   // As an illustration for the need of such an operation consider
   // two parallel long cylinders which are close together but
   // don't intersect. The bounding sphere computed for the
   // intersection will have a diameter similar to the length of the
   // cylinders, wheras the intersection is void, so that the radius
   // of the minimal bounding sphere is zero. The present function
   // will give radius zero.
   // It will, however, give radius zero also in cases where the set
   // fills only a very small fraction of the un-improved bounding
   // sphere.
   // The return value is the fraction of tried points that was found
   // to belong to *this. This is helpful as an estimate for the
   // volume of the set *this.
   // In order to avoid infinit work in case of void sets (which
   // easily result, for instance, from forming intersections of sets)
   // we will try not more than tryMax points

virtual R vol()const;
   // estimates the volume of the set by testing nFind random points
   // in the bounding sphere for the property to belong to the subset
   // or not. If we don't find nFind such points after having tried
   // nMax points, we base the estimate on what we have got so far.

virtual V<Spc> ran()const;
   // random sequence of nFind points in the subset; if
   // we don't find nFind such points after having tried
   // nMax points, we return what we have got so far.

Plane tangentialPlane(const Spc& x)const;
   // returns the tangential plane to the boundary near x

X2<R,Vec> distAndDirection(const Spc& x)const;
   // The first component of the return pair is the distance of x from
   // the surface (positive for points outside the point set described
   // by *this; needs not to be correct for points far away from the
   // set).
   // The second component is the outwards-directed normal unit
   // vector.
   // Thus, denoting the returned pair as (d,n), the surface point
   // nearest to x is approximately x-n*d. This does not assume that
   // f is locally metric. The step size for numerically comutating
   // grad f is get_dr()

void shiftExternalPointToSurface4(Spc& x, R& dOut,
   R& fxBefore, R& fxAfter)const;
   // Remark to a new notational device:
   // if a function name ends in a number n this means that the first
   // n arguments are non-constant references and that the intended
```

```
      // usage is to change the values of those (using them effectively
      // as 'return'-values).
      // If the function is mutating itself, I use a _ directly after the
      // letter part of the name. So 'X::some_3' would be the
      // name of a mutating member function of class with n>=3 arguments
      // the first 3 of which are mutable.
      //
      // Consider f(x). If this is <=0 we have an non-external point
      // so that x will not be changed by function call and
      // it will be set fxBefore=fxAfter=f(x) and dOut=0
      // (dOut=distance by which the point is 'outside').
      // If f(x)>0, after function call x is very close to the surface of
      // *this if it was not too far away before.
      // The second and third argument give f(x) for x before and after
      // function call.
      // dOut is the (xBefore-xAfter).absVal() and thus a good
      // approximation to the distance of the outside point x to the
      // surface of *this.

   BndSph toBndSph()const{ return BndSph(x,r);}
      // returning x and r as a BndSph

// boolean operations

   SubsetOfSpace operator &(const SubsetOfSpace& sos)const;
      // forming the intersection of *this and sos (corresponds to the
      // 'and' operation for indicator functions)
      // Notice (sos1&sos2)(x)=max(sos1(x),sos2(x))

   SubsetOfSpace operator |(const SubsetOfSpace& sos)const;
      // forming the union of *this and s (corresponds to the
      // 'or' operation for indicator functions)
      // Notice (sos1|sos2)(x)=min(sos1(x),sos2(x))

   SubsetOfSpace operator -(const SubsetOfSpace& sos)const;
      // forming *this\sos i.e. the set of all those elements of
      // *this which don't belong to sos (relative complement)

   SubsetOfSpace operator ^(const SubsetOfSpace& sos)const;
      // (*this-sos)|(sos-*this) i.e.
      // the symmetric difference of *this and sos
      // (corresponds to 'exclusive or' for indicator functions)
      // a direct definition as for the previous operations is
      // slightly more effective so that we do not
      // simply say:
      // { return (*this-sos)|(sos-*this);}

   friend  SubsetOfSpace unionOfArray(const V<SubsetOfSpace>& vs);
      // returns the union of all vs[i]
```

```
   friend  SubsetOfSpace sectionOfArray(const V<SubsetOfSpace>& vs);
      // returns the intersection of all vs[i]

    SubsetOfSpace combine(const V<SubsetOfSpace>& vs)const;
      // here we consider *this as a region of interest and return a
      // body which is formed out of all intersections vs[i]&*this
      // Those terms for which this intersection is void, are left out of
      // the result for efficiency.

// transformations. Notice that derived classes will have additional
// members to define the shape by serializable data. These will have to
// become included into the transformation. Since the MS compiler does
// not allow redefinition of virtual functions in a way that the return
// value is a reference to a derived class, we have to deviate from
// established customs to define transformations as returning a reference
// to the transformed object. This is a minor inconvenience if any.

   virtual void operator *=(const Group& g);
      // translation and rotation combined

   void operator *=(const F<Spc,Spc>& f);
      // If y is the generic point belonging to
      // *this before the action, then f^(-1)(y) is the generic point
      // of *this after the function call.
      // No change of rs and of x,r.
      // For x the natural change would be to f^(-1)(x).
      // change of r would be more complicated.
      // Note that x,r define only the visual scope and not
      // the set itself. So this is no problem.
      // So a perfect implementation is only possible if
      // f and f^(-1) are given on input.
      // Notice the strange situation: we need f for the implementation
      // and f^(-1) for a description of the meaning (semantics)
      // Notice also that action of Group hides this problem since
      // for g also g^(-1) is available

   void operator +=(const Vec& a){ *this*=Group(a);}
      // translation

   void rot(const AxVec& A){ *this*=Group(x,A);}
      // not defined as *= since overloading of virtual functions creates
      // unexpected (though correct) name hiding phenomena in derived
      // classes rotation arround the central point x

   void setCenter(const Spc& xNew){ operator+=(xNew-x);}
      // bringing the center into the new position by translation
      // Should set_x become replaced by this function ???
      // no !!!! Making this replacement caused severe malfunctions in
      // the padyan program (detected 2002-2-12, the bad change was made
      // not earlier than 2 weeks ago).
```

```
   void setRefSys(const RefSys& rsNew)
 //     { operator*=(rs.toSys(rsNew));}
   {
      rs=rsNew;
      x=rs.o();
      update();
   }
      // moving *this such that rs equals the input RefSys rsNew.

   Group toSys(const RefSys& rsNew)const{ return rs.toSys(rsNew);}

   F<Spc,R> getF()const{ return f;}

   void rotByEulAng(const Angle& phi,const Angle& theta,const Angle& psi)
   { rs.rotByEulAng(phi,theta,psi); update();}
      // rotation by Euler angles (see Goldstein: Klassische Mechanik,
      // Chapter 4-4, p. 119)

   X2<Spc,bool> hit(const Spc& xi, const Vec& v,
      R sFine, R sCoarse)const;
      // Let the return value be written as (r1,r2). Then:
      // If r2==true (regular case), the point r1 has a distance from
      // the surface which is not larger than sFine.
      // If r2==false, no indication of a body surface was found.
      // No assumption is made on whether xi is outside the body or
      // inside.
      // Improved method, based on a two-level analysis of the function
      // t |--> f(xi+t*v) , t>=0
      // The first step scans this function at step size sCoarse. So
      // detail much finer than this value will not be detected.
      // v needs to be a unit vector !!!
      // Only the part of the body within the assigned bounding
      // sphere is considered for hitting.

// Static generating function (more convenient than a constructor
// notice that cpmshapes.h defines also special instance by
// defining derived classes with constructors. So the names of
// these derived classes provide the intended meaning that here
// the function name indicates. I'm not sure which (if at all) of the
// two mechanisms is superior.

   static SubsetOfSpace band(Z i, const Iv& iv,
      const RefSys& rf=RefSys());
      // returns the 'set of points' x \in Spc such that
      // iv.contains(rf(x)[i])==true
      // Notice that a box can be formed as a cross section of such
      // bands

   static SubsetOfSpace boxoid(const Rdim& x0, const Rdim& s0,
```

```
      const F<Spc,Rdim>& f );
      // Consider X=
      // {x \in Rdim | x0[i]-s0[i]<=x[i]<=x0[i]+s0[i], i=1,...dim}
      // The SubsetOfSpace returned by the present function is
      // f^-1(X). If f is isometric, the radius of this set is simply
      // given by s0.absVal(). The problem is, that f -as used here-
      // does not carry a tag saying 'I am isometric'. We thus
      // have to make tests, and set r accordingly.

   static SubsetOfSpace sphere(Spc const& x0, R r0, R accRel=1e-3);
      // r0*accRel will be the typical discretization length

   static R box(const Rdim& x, const Rdim& x0, const Rdim& s0);
      // auxiliar function quantity
      // indicator function of the the box x0[i]-s0[i]<=x[i]<=x0[i]+s0[i]
      // Thus s0[i] is  h a l f  the interval length in i-direction
};

//////////////////////////// class Texture ////////////////////////////////
// tool for 2,3-dimensional visualization

class Texture: public CpmFields::ScalarField<Spc,R,Group>{// textures
   // method to assign a periodic pattern to space. This is essential for
   // enabling anaglyphic visualization of arbitrary shapes.
   // By deriving from ScalarField we inherit linear arithmetics and
   // operator *=(Group)
public:
   typedef CpmFields::ScalarField<Spc,R,Group> Base;
   typedef Texture Type;
   virtual F<Spc,R>* clone()const{ return new Texture(*this);}

explicit Texture(const V<R>& omega, const V<R>& delta,
   R p=1, Z method=1);
   // The return value of operator()(const Spc& sp) is a function of
   // yy[i]=0.5*(1+cos(delta[i]+sp[i]*omega[i])), i=1,...Spc.dimension().
   // From this list of numbers one forms the final value of
   // operator()(const Spc& sp) by expressions which are controlled
   // by method and p. p matters only for method 2.
   // method=1: 'additive method' (see code in cpmdimxcpp.h)
   // method=2: 'extended additive method'
   // method= not 1 or 2: 'multiplicative method'

explicit Texture(R width=0.001);
   // Special case of previous constructor. All components of delta are
   // 0, and those of omega are 2*pi/width.

Texture(const VV<R>& mat, Iv xI, Iv yI, Z method=1);
   // Defines a texture on the 1-2-plane of RefSys() by repeating
   // the rectangle xI x YI and filling each rectangle with the
   // 'image matrix' mat. The argument method controls the extension
```

```
   // of mat from Z x Z to R x R. method=0 means pixel-wise
   // representation
   // method=1 is a fast simplex interpolation. Other values give error.
   // For this application, the values of mat[i][j] should belong to
   // the interval [0,1]

explicit Texture(const F<Spc,R>& fIn):Base(fIn){}
   // flexible definition from any function of the right type

explicit Texture(const F<C,R>& fIn);
   // Uses the function fIn in order to define the pattern in the
   // 1-2-plane. If Spc::dimension()==3, the pattern extends into
   // the 3-direction in the trivial manner that it does not depend
   // on the 3-coordinate

R eval(const RefSys& rf,const Spc& sp)const
   // Here the value is computed under the assumtion that the
   // pattern is fixed with respect to a system of reference
   // given as the first argument. It would be nice to have the
   // name operator()(arg1,arg2) for this function. Unfortunately the
   // name operator() declared in this class would hide the
   // most important operator() of the base class. This would be
   // more inconvenient than introducing a verbal name here.
{
   Group g0=rf.toMasSys(); Spc x=sp;
   return operator()(x*=g0);
}

};

//////////////////////// class Triangle ////////////////////////////////

class Triangle{ // triangle, few methods
   Spc p1_, p2_, p3_;
   typedef Triangle Type;
public:
   CPM_IO
   Triangle():p1_(),p2_(),p3_(){}
   Triangle(Spc const& p1, Spc const& p2, Spc const& p3):
      p1_(p1),p2_(p2),p3_(p3){}
   Word nameOf()const{ return "Triangle";}
   Spc p1()const{ return p1_;}
   Spc p2()const{ return p2_;}
   Spc p3()const{ return p3_;}
   Triangle& operator *=(Group const& g);
      // translation and rotation combined
   Triangle& operator +=(Vec const& a);
      // translation
   Triangle& operator *=(AxVec const& A)
      { return operator*=(Group(A));}
```

```
      // rotation
   Triangle& scl_(Spc const& fix, R fac)
      //: scale
   {
      p1_.scl_(fix,fac);
      p2_.scl_(fix,fac);
      p3_.scl_(fix,fac);
      return *this;
   }
};

//////////////////////// class Path ////////////////////////////////////////

class Path{ // array of points as a discrete representation of a curve
   // array of points considered as following one after the other
   // just according to their indexing. Lean structure that allows
   // the natural definition of operations that make sense for
   // curves but not for general graphs.
   // Correctly formed path satisfy (x_[i+1]-x_[i]).absVal()>acc_
   // for all i \in {1,...n_-1}.
   // 2003-11-11: adding the dependent quantities y_,e_,l_
   // that allow a very efficient implementation of the relevant
   // algorithms.

   V<Spc> x_;
      // list of points
   Vo<R> y_;
      // y_[i] = arc length (i.e. length along the path *this
      // from x_[1] to y_[i]
   V<Vec> e_;
      // e_[i]= unit vector from x_[i] to x_[i+1]
   Vo<R> l_;
      // l_[i]=length of vector x_[i+1]-x_[i]
   R1 length_;
      // total length
   Z1 n_;
      // number of points
   R1 acc_;
      // accuracy: points that are closer to each other than
      // this are considered not as different and one of them will
      // be purged

   Z adjust(R acc);
      // eliminates repetition points (acc determines what these are
      // precisely) and sets n_ and length_ accordingly.
      // Returns the number of points that had to be eliminated.
      // if the argument is 0, then the real accuracy is set
      // according to a specified fraction of the shortest inter-
      // point distance
```

```
   void autoAcc(){ if (n_>1) acc_=(1e-3)*length_/n_;}
      // sets the accuracy reasonably (not according
      // to all existing points in x_, since x_ may already be spoilt
      // by adding points that differ from intended ones merely by
      // numerical noise

   R update();
      // adjusts the values of n_,length_,y_,l_,e_ according to x_
      // returns the minimum of the l_[i]

   typedef Path Type;

public:
   CPM_IO
   explicit Path():x_(0){}
   Path(V<Spc> const& vsp, R acc=0):x_(vsp)
   {adjust(acc);cpmassert(isSmooth(),"Path(V<Spc>)");}

   Path(Spc p1, Spc p2, R acc=0):x_(V<Spc>("",p1,p2))
   {adjust(acc);}

   Path(Spc p1, Spc p2, Spc p3, R acc=0):x_(V<Spc>("",p1,p2,p3))
   {adjust(acc);cpmassert(isSmooth(),"Path(V<Spc>)");}

   Path(Spc p1, Spc p2, Spc p3, Spc p4, R acc=0):
      x_(V<Spc>("",p1,p2,p3,p4))
      {adjust(acc);cpmassert(isSmooth(),"Path(V<Spc>)");}

   Path(Spc p1, Spc p2, Spc p3, Spc p4, Spc p5, R acc=0):
      x_(V<Spc>("",p1,p2,p3,p4,p5))
      {adjust(acc);cpmassert(isSmooth(),"Path(V<Spc>)");}

   Path(Spc p1, Spc p2, Spc p3, Spc p4, Spc p5, Spc p6, R acc=0):
      x_(V<Spc>("",p1,p2,p3,p4,p5,p6))
      {adjust(acc);cpmassert(isSmooth(),"Path(V<Spc>)");}

   Path(Spc const& p1, Vec const& v1, Spc const& p2,
      Angle const& phi, Z n=10);
      // Constructs a polygon Path of n points, starting at point p1
      // in direction v1 and ends at p2 with direction v1*av
      // where av is the AxVec has a rotation angle and a
      // rotation axis perpendicular to v and p2-p1.
      // Based on a three point Bezier curve.

   Path(Spc const& p1, Vec const& v1, Spc const& p2,
      Vec const& v2, Z n=10);
      // Constructs a polygon Path of n points, starting at point p1
      // in direction v1 and ending at p2 with direction v2.
      // Actually it is the three point Bezier curve defined by
      // p1,pc,p2 where the control point pc is
```

```
      // taken as the point of closest approach of the lines
      // (p1,v1) and (p2,v2). If there is no plane in which
      // p1,p2,v1,v2 lie (and thus also pc) the actual initial and final
      // directions, pc-p1, p2-pc will deviate from the intended
      // directions v1 and v2.

   Path(Spc const& p1, Spc const& p2,
      Vec const& v2, R s=0.25, Z n=10);
      // Constructs a polygon Path of n points , starting at point p1
      // and ending at p2 with direction v2. The parameter s should
      // be selected in [0,1]. For small s, the initial direction
      // points nearly to p2 and the requested final direction will be
      // achieved by a narrow curve. For s close to 1, the initial
      // direction and the final direction are close to perpendicular
      // to each other and the path is as constanly curved as
      // possible.

   Z dim()const{ return n_;}
// non-mutating access to the points
   R length()const{ return length_;}
   // old name
   R len()const{ return length_;}
   //: length
   Spc firstPoint()const{ return x_[1];}
   Spc lastPoint()const{ return x_[n_];}
   Spc operator[](Z i)const{ return x_[i];}
      // returning the points
// non-mutating access to directions
   Vec firstDir()const;
      // first direction,
      // does not return unit vectors
      // returns the zero-vector if no reasonable
      // direction can be given
   Vec lastDir()const;
      // last direction

   Vec dir(Z i)const;
      //: direction
      // direction at point i where this has a meaning and Vec() else

   R acc()const{ return acc_;}
      //: accuracy

   void setAcc(R acc){ acc_=CpmRootX::sup<R>(0.,acc);adjust(acc_);}
      //: set accuracy

   Path& operator *=(Group const& g);
      // translation and rotation combined
   Path& operator +=(Vec const& a);
      // translation
```

```
   Path& operator *=(AxVec const& A);
      // rotation arround the first point
   Path& operator *=(R fac);
      // scaling relative to the first point

   Path& rot_(Spc const& fix, AxVec const& A);

   Path scl(Spc const& fix, R fac)const;
      //: scale
      // Returns a path resulting from *this by
      // multiplying all position vectors relative to
      // the origin 'fix' by factor fac.
      // Notice that having the path's first point as fixed
      // point (as in operator*=(R)) is not always what is needed.
   Path& scl_(Spc const& fix, R fac){ return *this=scl(fix,fac);}

   V<Path> thc(R r, Z n)const;
      //: thick
      // This function returns n shifted
      // copies, where the n shift vectors have length r and
      // result from equispaced points on a circle (not disk!).
      // This is a brut force method to simulate a line thickness
      // of 2*r. Works fine. But the suitable n has to be found by
      // experimentation for given r. If n is too small, there remain
      // voids, if it is too large, the function takes too much time.

   V<Path> brkLine(R d, R frac=0.5)const;
      //: broken line
      // Returned is a list of Paths which result from *this by leaving
      // out a fraction of relative length frac of the periodicity
      // element d. The actual periodicity element may differ a bit
      // from d since a minimum adjustment is done which lets the
      // end points of *this always are the end of a visible line piece.

   Path rev()const{ return Path(x_.rev(),acc_);}

   Path operator()(const F<Spc,Spc>& g)const
   { return Path(x_(g),0);}
      // transforming the components by a function
      // Since the function may span orders of magnitude
      // the accuracy has to be set from x_(g) which is
      // triggered by the 0-argument

// generation functions as static functions (constructors
// can't have a descriptive name, so named functions
// are more convenient to use).

   static Path arc(const Spc& p1, const Spc& center,
      const AxVec& rot, Z n=10);
      // circular arc obtained by applying a rotation rot arround
```

```
   // center in n steps on point p1.

static Path arc2(const Spc& p1, const Spc& aimPoint,
   const AxVec& rot, Z n=10);
   // circular arc obtained by deviating from initial direction
   // p1 --> aimPoint, having arc length |aimPoint-p1| and
   // lies in the plane containing p1 and aimPoint and being
   // perpendicular to rot.

static Path line(const Spc& pi, const Spc& pf, Z n=10);
   // straight line starting at pi and ending at pf
   // represented as a chain of n equidistant connected
   // points. (n-1 edges). cpmassert for n>1.

static Path parabola(const Spc& pi, const Spc& pc, const Spc& pf,
   Z n=10);
   // parabola with initial point pi, final point pf, and
   // Bezier control point pc. The intermediate points
   // correspont to equidistant positions in time, where the time span
   // needed for the whole path is 1. The meaning of the control
   // point is to express the initial velocity as (pc-pi)/(1/2)
   // and the final velocity as (pf-pc)/(1/2).
   // cpmassert for n>2.

static Path ellipse(const Spc& pc, R a1, R a2, Z n=16);
   // generates an ellipse in the 1-2-plane around point pc
   // approximated by a closed flattened n-polygon.
   // We start from a circle
   // of radius a1 and multiply the 2-coordinates by a2/a1
   // (everything relativ to the master system of reference).
   // half-axis a1 in 1-direction
   // half-axis a2 in 2-direction
   // For a1==a2 (circle) more efficient code is generated by
   // avoiding the flattening loop.

static Path quarterEllipse(Spc const& p1, Vec const& v1,
   Spc const& p2, Z n=16);
   // generates an elliptic quadrant line, approximated as a
   // polygon of n+1 points.
   // The eliptic arc starts in p1 at direction v1 and ends in p2 with
   // a direction perpendicular to v1. The direction from the
   // beginning to the end of the arc varies monotoneously from
   // the initial direction di to the final direction
   // di +/- 90 degrees. If p2 does not lie in the half-plane
   // determined by ((p2-p1)|v1)>=0 an assertion (cpmassert) fails,
   // and the consequences of this range from ignoration to program
   // stop depending on the status of macro cpmdebug.

static Path circle(const Spc& pc, R r, Z n=16); // 4 is experiment
   // see function ellipse of which the present function is a special
```

```
   // case

static Path bezier(V<Spc> const& vs, Z n);
   // Bezier curve connecting vs.first() with vs.last() represented as
   // a polygon of n points

static Path locInt(V<Spc> const& vs, Z n);
   //: local interpolation
   // smooth local interpolated curve connecting vs.first() with
   // vs.last() represented as a polygon of n points

static Path cubCur(Spc const& p1, Vec const& v1,
   Spc const& p2, Vec const& v2, Z n=64);
   //: cubic curve
   // Represents the unique cubic curve x: [0,1] --> Spc
   // satisfying x(0)=p1, x(1)=p2, x'(0)=v1, x'(1)=v2 as
   // a polygon of n points (n>1 asserted).
   // See e.g. the nice book G.Aumann, K.Spitzmueller:
   // Computerorientierte Geometrie, BI 1993 p. 330, (11.7).
   // Although this function has the same argument structure
   // as the constructor
   // Path(Spc const& p1, Vec const& v1, Spc const& p2,
   // Vec const& v2, Z n=10);
   // it depends on the directions in a different way: In the
   // present function also the length of v1 and v2 matter (as it
   // obviously does for tangent vectors of a curve). For assessing
   // what length should be used in a particular curve design
   // situation one has to take into account that the parameter
   // interval for the whole curve is [0,1] which may be very
   // different from the geometrical length of the path, so that
   // the parametrization may be far away from the natural one
   // by arc length.

static Path cubCur(Spc const& p1, Vec const& v1, R k1,
   Spc const& p2, Vec const& v2, R k2, Z n=64);
   // This function allows representing exactly the same curves
   // as the previous one. However, the parameters are are
   // better suitable for curve design. Independent of the
   // length of v1 and v2, they enter only by the unit vectors which
   // they determine. The factors k1 and k2 influence the relative
   // influence of the direction (the larger the corresponding factor
   // the longer piece of path that follows the tangent line closely.
   // Changing k1 and k2 by a common factor makes the curve longer
   // until it finally may be forced to develop a crossing point.
   // See p. 335, Fig. 11.3 in the book by Aumann and Spizmueller.
   // For k1=k2=1, one gets a curve which does not deviate much from
   // the connecting line from p1 to p2. For larger values, say 2
   // or 3, the increased arc length of the curve gives freedom to
   // follow the directions given at the ends to a more visible
   // degree and make the change from the initial direction into
```

```
   // the final one more smoothly.
   // With this function it was easy to create a greek lower case theta
   // (the variant which is completely different from the upper case
   // theta) by only three of such curves.

 static Path waveLine(Spc const& p1, Spc const& p2, R ampl,
    R omega, Vec const& tr=Vec(1,0), Z n=100);
   //: wave line
   // A wavy line (as photon lines in Feynman diagrams) from
   // p1 to p2. Actually this is a line in space in the form of a
   // linearly polarized wave. The actual spatial angular frequency
   // differs from the input omega as little as possible and
   // nevertheless lets the distance between p1 and p2 be an integer
   // number of half-waves. Notice lambda=2*pi/omega.
   // The actual transversal amplitude vector
   // is a unit vector which lies in the plane detemined by the
   // vectors p2-p1 and tr and is (of course) perpendicular to
   // the direction p2-p1.
   // The amplitude of the wave is given exactly by the argument ampl
   // whereas tr enters only as direction.

Word nameOf()const{ return "Path";}
   //: name of

VecGraph flatten()const;
   // returns the VecGraph object resulting from projecting
   // *this to the (1,2) plane

const V<Spc>& operator()(void)const{ return x_;}
   // accessing the basic data via () as in
   // Path p=...;
   // V<Spc> vs=p();

X2<Path,Path> divideByArc(R t)const;
   // let res be the return value.Then the intention is
   // *this == res.c1() & res.c2()
   // res.c1().length()==t

X2<Path,Path> divideByPoint(const Spc sep)const
   // direct derivative from the previous function
{
   return divideByArc(project(sep).c3());
}

Path operator &(const Path& p)const;
   // append p at the end of a copy of *this
   // and returns the result. If *this and p
   // fit perfectly together, the connection point
   // does not get doublicated. If they don't fit together
   // they simply get connected by a segment (i.e.
```

```
   // the list of points gets simply concatenated)

Path moveAndMerge(const Path& p)const;
   // moves p such that it can be smoothly (without a kink)
   // appended at the end of a copy of *this. The result of
   // this appending procedure is returned

Path replace(const Spc& s1, const Spc& s2, const Path& p)const;
   // The operation has a clear meaning only if s1 and s2
   // ly on the path or near to it so that their projected
   // points s1_, s2_ are conceptually clear.
   // Denote u1 the first of points {s1_,s2_} that we encounter
   // when going from firstPoint() along the path. Let u2
   // be the other one in {s1_,s2_}.
   // The resut is: (*this) from firstPoint() to u1
   // then p (connected by move and merge)
   // then (*this) from u2 to lastPoint() (connected by move and
   // merge)

X3<Spc,Vec,R> project(const Spc& s)const;
   // Returning geometrical data, which allow to move s onto the
   // path *this in cases in which s is close to this path.
   // Precisely:
   // For n_==0 the function returns X2<Spc,R>(Spc(),0).
   // For n_==1 the function returns X2<Spc,R>(x_[1],0).
   // For n>=2 (the only case of interest) the function returns
   // (s0,e0,l0) \in X3<Spc,Vec,R> with s0,e0, and l0 defined as
   // follows:
   // We consider the n_-1 segments ('Strecken') x_[i]-->x_[i+1]
   // and treat them as point sets. Then s has a well-defined distance
   // to each of the segments and we select the segment for which
   // this distance is minimal. (If there are more with the same
   // distance, which is extremly unprobable, we select the one with
   // the smallest index). s0 is by definition the point on the line
   // which is determined by the selected segment which is closest to
   // s. The unit vector corresponding to the selected segment is e0.
   // Appart from points that are not near to the path or close to its
   // s0 will belong to its line-determining segment and will thus
   // belong to the path (which is made of these segments).
   // l0 is the distance of s0 from the initial point x_[1], measured
   // along the path.

R toArc(const Spc& s)const{ return project(s).c3();}
   //: to arc
   // returns a number that characterizes the position of s along the
   // path. The function does not assume that s belongs to *this, but
   // if it does the result is easier to explain: it is the distance
   // from x_[1] measured along the path. Actually clear from code.

X3<Spc,Vec,Z> fromArc(R t)const;
```

```
      //: from arc
      // Consider:
/*
         Path p=...;
         R t=...;
         Spc s=p.fromArc(t).c1();
         R t0=p.toArc(s);
         bool test=(t==t0);
*/
      // Then we get always the value true for the variable test.
      // Also returns the unit vector of tangent direction
      // as ret.c2() and i0=ret.c3() such that l_[i0]<=t<l_[i0+1]

   R turn()const;
      // returns the total winding angle from beginning to end,
      // e.g 2*pi for a closed circle, or 2*n*pi for a path obtained from
      // walking around a circle n times before meeding the initial
      // point again. Notice that it would waist information
      // if we would give the result as an Angle (which is unable
      // to hold the 'number of full turns'
      // The accurate mode should use the value of
      // e_[1].angTo(e_[n_-1]) for correcting the rounding errors
      // occuring during many steps in possibly many turns of the path.
      // Tests with very large number of points showed that there is
      // hardly a need for such an accuratification

   Group con()const;
      //: connection
      // connection along the whole path. Consider
      // Path p=...;
      // Group g=p.con();
      // Then the following expressions evaluate to true:
      // (p.firstPoint()*=g) == p.lastPoint()
      // (p.firstDir()*=g) == p.lastDir()
      // Very interesting concept. I knew connections only for
      // linear groups; the present form for an affine (inhomgeneous)
      // transformation group is in my head since ,say, two years
      // and was implemented here in order to describe the release of
      // a bended wire, whithout knowing the chain of states through
      // which bending was done. (program CO-Former)

   R turn2(R t1, R t2)const;
      // Going along the parth from fromArc(t1).c1() to fromArc(t2).c1()
      // (this might be a walk opposite to the inherent path direction)
      // we add together the rotations from path element to path element
      // (as numbers not as angles, i.e no mod-2*pi-reduction is
      // involved) and the sum is the result. One has
      //    turn2(t1, t2) == - turn2(t2, t1)
      // and for t1 <= t2 and t2-t1 sufficiently small that no completed
      // turn is made:
```

```
      //  turn2(t1,t2)==
      // (Vec e1=fromArc(t1).c2(), Vec e2=fromArc(t2).c2(), e1.angTo(e2) )

   Group con2(R t1, R t2)const;
      // generalizes con in just the same way in which turn2 generalizes
      // turn

   void static test(Z i);
      // Test tool

   bool isSmooth()const;
      // returns true if there are no reversions in direction
      // from point to point
};

//////////////////////////// class Skeleton ////////////////////////////////

class Skeleton:public DiGraph<Spc>{// directed graph of points
   // Basic tool for creating spatial
   // structures to be visualized with objects of type Camera
   typedef DiGraph<Spc> Base;
   typedef DiGraph<Spc> CloneBase;
   typedef Skeleton Type;
public:
   virtual DiGraph<Spc>* clone()const{ return new Skeleton(*this);}
      // I/O from Base
   explicit Skeleton(Z n=0):Base(n){}
   explicit Skeleton(const V<Spc>& vsp, bool connect=false):
      Base(vsp){ if (connect) connectAsPath();}
      // notice that there are no edges yet, so that the
      // object is invisible when invocing
      // Camera ca=...; Skeleton sk=...; Color col=...;
      // ca.mark(sk,col);

   Skeleton(Spc const& p1, Spc const& p2, Spc const& p3,
      bool connect=false): Base(V<Spc>("",p1,p2,p3))
      { if (connect) connectAsPath();}
      // constructing a Skeleton from 3 points

   explicit Skeleton(const VV<Spc>& vsp, bool connect=false):
      Base(vsp.lin()){ if (connect) connectAsPath();}
      // using a matrix of points as input. If connections are
      // generated these duo not (yet ?) reflect the matrix mesh
      // geometry

   const Spc& lastPoint()const
      { return operator[](dim());}
   const Spc& firstPoint()const
      { return operator[](1);}
   Skeleton& operator *=(Group const& g);
```

```
      // translation and rotation combined
   Skeleton& operator +=(Vec const& a);
      // translation
   Skeleton& operator *=(AxVec const& A);
      // rotation

   Skeleton& scl_(Spc const& fix, R fac);
      //: scale

   Skeleton con(R w=0.5)const;
      // creates the dual graph, where each edge is transformed into the
      // vertex which is the center of the edge in the normal case w=0.5.
      // Since the edges are directed, it makes sense to put the new
      // vertex closer to the old source vertex (w<0.5) or closer to the
      // old destination vertex (w>0.5). The edges of the new graph are
      // lines that join just those vertices which in the old picture
      // where adjacent edges.

   void connectAsPath();
      // adds edges 1->2,2->3,4-->5,... n-1->n
      // and makes the edges vertexes visible

   void clsPath_(){ addEdge(x_.dim(),1);}
      //: close path
      // adds an edge from the last vertex to the first one

// generation functions as static functions (constructors
// can't have a descriptive name, so named functions
// are more convenient to use).

   static Skeleton arc(const Spc& p1, const Spc& center,
      const AxVec& rot, Z n=10);
      // circular arc obtained by applying a rotation rot arround
      // center in n steps on point p1.

   static Skeleton line(const Spc& pi, const Spc& pf, Z n=10);
      // straight line starting at pi and ending at pf
      // represented as a chain of n equidistant connected
      // points. (n-1 edges). cpmassert for n>1.

   static Skeleton parabola(const Spc& pi, const Spc& pc, const Spc& pf,
      Z n=10);
      // parabola with initial point pi, final point pf, and
      // Bezier control point pc. The intermediate points
      // correspont to equidistant positions in time, where the time span
      // needed for the whole path is 1. The meaning of the control
      // point is to express the initial velocity as (pc-pi)/(1/2)
      // and the final velocity as (pf-pc)/(1/2).
      // cpmassert for n>2.
```

```
   static Skeleton ellipse(const Spc& pc, R a1, R a2, Z n=16,
      bool connect=true);
      // generates an ellipse in the 1-2-plane arround
      // the origin approximated by a flattened
      // n-polygon. We start from a circle of radius a1
      // and multiply the 2-coordinates by a2/a1
      // (everything relativ to the master system of reference).
      // half-axis a1 in 1-direction
      // half-axis a2 in 2-direction
      // if connect==true the points are connected as a polygon,
      // otherwise the graph has no edges.
      // For a1==a2 (circle) more efficient code is generated by
      // avoiding the flattening loop.

   static Skeleton circle(const Spc& pc, R r, Z n=16, bool connect=true);
      // generates a circle in the 1-2-plane arround point pc
      // the origin approximated by a  n-polygon.
      // if connect==true the points are connected as a polygon,
      // otherwise the graph has no edges.

   static Skeleton bezier(V<Spc> const& vs, Z n);
      // Bezier curve connecting vs.first() with vs.last() represented as
      // a polygon of n points

   static Skeleton cone(R r, R h, Z n=16);
      // Pyramide over a regular n-polygon of higth h

   static Skeleton cylinder(R r, R h, Z n=16);
      // Cylinder over a regular n-polygon of higth h

   static Skeleton dblPyr(R length, R d, Z n=8);
      // method for generating a double pyramide ....

   static Skeleton rec(R a, R b);
      //: rectangle
      // Returns a rectangle with corners at (0,0),(a,0),(a,b),(0,b)

   Word nameOf()const{ return "Skeleton";}

   VecGraph flatten(Z connect)const;
      // returns the VecGraph object resulting from projecting
      // *this to the (1,2) plane
      // Argument connect will be given to the VecGraph constructor
};

//////////////////////////// class WireFrame ////////////////////////////////

class WireFrame{ // array of Skeleton objects
   // An array of skeletons is considered a new
   // entity; notice that it would be probably quite inefficient to
```

```
   // convert such an array into a single Skeleton by forming the obvious
   // union of sets.

   V<Skeleton> rep;

   typedef WireFrame Type;
public:
   CPM_IO
   virtual WireFrame* clone()const{ return new WireFrame(*this);}

explicit WireFrame(const V<Skeleton>& vsk):rep(vsk){}
explicit WireFrame(const Skeleton& sk):rep(V<Skeleton>(1,sk)){}
   WireFrame():rep(){}

   WireFrame con(R w=0.5)const;
      // creates the dual graph. See Skeleton::con(R)

   WireFrame& operator *=(Group const& g);
   WireFrame& operator +=(Vec const& a);
   WireFrame& operator *=(AxVec const& A);

// transforming the components of the WireFrame separately
// no restriction on dimensions since only those indexes are
// used for which both arrays have components
   WireFrame& operator *=(V<Group> const& g);
   WireFrame& operator +=(V<Vec> const& a);
   WireFrame& operator *=(V<AxVec> const& A);

// combining WireFrames
   WireFrame& operator &=(WireFrame const& wf)
      { rep&=wf.rep; return *this;}
   WireFrame& operator &=(Skeleton const& sk)
      { rep&=sk; return *this;}

// access
   const Skeleton& operator[](Z i)const{ return rep[i];}
   Z dim()const{ return rep.dim();}
   Word nameOf()const{ return "WireFrame";}

   static WireFrame tripod(R length, R relWidth=0.025, R relGap=0.05,
      Z n=3);
      // generates a system of orthogonal axes such that
      // axis n of the master system is made out of n pieces.

   static WireFrame globe(R r, R f=0, Z nLong=8, Z nLat=4, Z nCirc=16,
      bool axes=false);
      // method for generating a rotation ellipsoid like
      // the earth body.
      // r: equatorial radius
      // f: flattening factor: polar radius=r*(1-f)
```

```
      // nLong: we show longitude circles of angular distance
      //    180 degrees/nLong
      // nLat: we show latitude circles of angular distance
      //    90 degrees/nLat
      // nCirc: each 'circle' actually is a regular nCirc-polygon.
      // For f!=0 the latitude circles neither refer to topocentric
      // latitude nor to bodycentric latitude. Rather, they simply
      // result from flattening of a spherical globe and its
      // uniquely defined latitude circles. Flattening occurs in the
      // 2-direction (?? or 3 ?)

   static WireFrame spline(const V<Spc>& vs, const Spc& pc1, Z n=10);
      // 'smooth' merge of parabolas where the first one is
      // Skeleton::parabola(vs[1],pc1,vs[2]),
      // the second one:
      // Skeleton::parabola(vs[2],pc2,vs[3]),
      // where pc2 is given by pc2-vs[2]=vs[2]-pc1
      // the third one:
      // Skeleton::parabola(vs[3],pc3,vs[4]),
      // where pc3 is given by pc3-vs[3]=vs[3]-pc2
      // ....
      // So, when considered a path in space (parametrized by time)
      // the position and velocity are continuous functions,
      // only the acceleration changes discontinuously from one
      // parabola to the next
};
```

52 cpmdimxcpp.h

```
//? cpmdimxcpp.h
//? Status of work 2008-10-25.
//?
/*
   Description: dimension-independent part of implementation
   of geometrical objects
*/
using namespace CpmFunctions;
using CpmGeo::Iv;

// symmetric version
// second order, seems to be as accurate as fourth order in my
// applications

Vec grad(Spc const& x, const F<Spc,R>& f, const R& h0)
{
   const R hTiny=1e-12;
   R h=(h0==0 ? hTiny : h0);
   Z n=x.dim();
   Vec res;
   R hInv=0.5/h;
   for (Z i=1;i<=n;i++){
      Spc a=x;
      Spc b=x;
      a[i]+=h; // shifted on i-axis
      b[i]-=h;
      res[i]=(f(a)-f(b))*hInv;
   }
   return res;
}
// symmetric version 4th order
// elegant employment of function type, presently not used

/*
namespace{
   R gradFunc(R const& x, Z const& i, Spc const& p, F<Spc,R> const& f)
   {
      Vec h;
      h[i]=x;
      return f(p+h);
   }
}

Vec grad(Spc const& p, F<Spc,R> const& f, R const& h0)
{
   Vec res;
   Z n=res.dim();
```

```
   for (Z i=1;i<=n;++i){
      R_Func g=F3<R, Z,Spc,F<Spc,R>, R>(i,p,f)(gradFunc);
      res[i]=g.der(1,h0)(0.);
   }
   return res;
}
*/

//////////////////////// class Vec ////////////////////////////////////////

Vec Vec::randomize(R dv, Angle const& a, Z j)const
{
   Vec res=*this;
   if (dv!=0.){ // then the result should have a length
      // in the interval [max(0,|v|-|dv|),|v|+|dv|]
      R avL=0,avU=0;
      R adv=cpmabs(dv); // |dv|
      R av=res.normalize();
      if (av>0.){
         avU=av+adv;
         avL=av-adv;
         if (avL<0) avL=0.;
      }
      else{
         res=Vec::ranUnitVec(j++);
         avU=adv; // notice avL=0
      }
      // now res is a unit-vector
      Iv iv(avL,avU);
      av=iv.ran(j++);
      res*=av;
   }
   AxVec ax(a);
   ax=ax.ranVal(j);
   res*=Group(ax);
   return res;
}

//////////////////////// class RefSys ////////////////////////////////////

RefSys::RefSys(const V<Spc>& vs)
{
   Z mL=3;
   Word loc("RefSys::RefSys(const V<Spc>& vs)");
   CPM_MA
   Z d=vs.dim();
   Z ns=orig.dim();
   R a1,a2,a3;
   bool b1,b2,b3,b4;
```

```
   b1=(d>=ns);
   cpmassert(d>=ns,loc);
   AxVec phi;
   if (!b1) goto trivial;
   x0=vs[1];
   v1=vs[2]-x0;
   a1=v1.normalize();
   b2=(a1>0);
   cpmassert(a1>0,loc);
   if (!b2) goto trivial;

   if (ns==2){
      v2=v1*AxVec(Angle(90.,DEG));
      phi=u1.rotTo(v1);
   }
   else{
      v2=vs[3]-x0;
      a2=v2.normalize();
      b3=(a2>0);
      cpmassert(a2>0,loc);
      if (!b3) goto trivial;
      Vec v2Ortho=v1.tanNormDec(v2).second();
      a3=v2Ortho.normalize();
      b4=(a3>0);
      cpmassert(a3>0,loc);
      if (!b4) goto trivial;
      Vec v2=v2Ortho;
      AxVec v3_=v1.cross(v2);
      AxVec u3_=u1.cross(u2);
      Vec v3;
      Vec w3; // in order not to re-declare u3 which is
         // already defined for ns==3; however, the compiler does
         // not know that we come here only in this case
      for (Z i=1;i<=ns;++i){
         v3[i]=v3_[i];
         w3[i]=u3_[i];
      }
      V<Vec> tripodMaster("",u1,u2,w3);
      V<Vec> tripodThis("",v1,v2,v3);
      bool careful=false;
      phi=AxVec(tripodMaster,tripodThis,careful);
   }
   g0=Group(phi,x0-orig);
   update();
   CPM_MZ
   return;
trivial:
   cpmwarning("Trivial return in "&loc);
   *this=RefSys();
}
```

```
////////////////////// class Group ////////////////////////////////////

Z Group::com(const Group& s)const
{
   if (A<s.A) return 1;
   if (A>s.A) return -1;
   if (a<s.a) return 1;
   if (a>s.a) return -1;
   return 0;
}

////////////////////// class Spc ////////////////////////////////////

Z Spc::com(const Spc& s)const
{
   return p.com(s.p);
}

X4<Spc,R,R,R> Spc::meet(Spc const& p1, Vec const& v1,
         Spc const& p2, Vec const& v2)
{
   R v11=(v1|v1);
   R v12=(v1|v2); // v21=v12, not needed
   R v22=(v2|v2);
   Vec diff=p1-p2;
   R a1=(diff|v1);
   R a2=(diff|v2);
   R_Matrix m(2,2);
   m[1][1]=-v11;
   m[1][2]=v12;
   m[2][1]=-v12;
   m[2][2]=v22;
   R_Vector a(2);
   a[1]=a1;
   a[2]=a2;
   R_Vector sol=(!m)*a;
   R s1=sol[1], s2=sol[2];
   Spc q1=p1+v1*s1;
   Spc q2=p2+v2*s2;
   R dev=(q1-q2).absVal()*0.5;
 //  cpmdebug(dev);
   return X4<Spc,R,R,R>(q1.cen(q2),dev,s1,s2);
}

////////////////////// class RefSys ////////////////////////////////////

namespace{

   Rdim srFunc(const Spc& x, const RefSys& rs){ return rs(x);}
```

```
   Spc rsFunc(const Rdim& x, const RefSys& rs){ return rs(x);}

}

F<Spc,Rdim> RefSys::toFSR()const
{
   return F1<Spc,RefSys,Rdim>(*this)(srFunc);
}


F<Rdim,Spc> RefSys::toFRS()const
{
   return F1<Rdim,RefSys,Spc>(*this)(rsFunc);
}

////////////////////// BndSph ////////////////////////////////

BndSph BndSph::operator|(const BndSph& bs)const
{
   Vec n=bs.x-x;
   R d=n.normalize();
   R rMax=CpmRootX::sup<R>(r,bs.r);
   if (d==0) return BndSph(x,rMax);
   Iv iv1(-r,r,true);
   Iv iv2(d-bs.r,d+bs.r,true);
   Iv iv=iv1|iv2;
   R rRes=iv.radius();
   Spc xRes=x+n*iv.mean();
   return BndSph(xRes,rRes);
}

BndSph BndSph::operator&(const BndSph& bs)const
{
   Vec n=bs.x-x;
   R d=n.normalize();
   R r1=r;
   R r2=bs.r;
   if (d>=r1+r2){ // void result
      return BndSph(); // 'exit 1'
   }
   else if (d>=r1 && d>=r2){ // section is a biconvex lens
      R rRes=sqrt((-d+r1+r2)*(d-r1+r2)*(d+r1-r2)*(d+r1+r2))/(2*d);
         // notice: d>0 since d==0 here, implies r1==r2==0.
         // In this case the first exit has been taken.
         // All factors under the root are >=0:
         // the first since exit 1 would have been taken else
         // the second since d>=r1
         // the third since d>=r2 and the fourth obviously.
         // rRes is found from the obvious relation
         // d = sqrt(r1^2-rRes^2)+sqrt(r2^2-rRes^2)
```

```
         // See my Mathematica file boundingshere3.nb.
      R aRes=sqrt(r1*r1-rRes*rRes);
      Spc xRes=x+n*aRes;
      return BndSph(xRes,rRes);
   }
   else{  // result contains a hemi-sphere
      if (r1<r2) return *this;
      else return bs;
   }
}

namespace{

   R sqrtS(R x) // safe form of function sqrt
      // the following algorithm failed frequently when working on
      // data read from file (limited accuracy !) by feeding
      // slightly negative arguments ( ~ -4e-16) into sqrt.
      // Here we treat such values as 0

   {
      const Z mL=3;
      const R limit=-1e-12;
      if (x>=0) return sqrt(x);
      else if (x>=limit){
         if (cpmverbose>=mL){
            Word mes="sqrtS(R): 0>x>=limit, x="&cpmwrite(x);
            cpmmessage(mes);
         }
         return 0;
      }
      else{
         Word mes="sqrtS(R): limit>x, x="&cpmwrite(x);
         cpmerror(mes);
         return 0;
      }
   }
}

Iv BndSph::hit(const Spc& x0, const Vec& n)const
// n unit vector
{
   using CpmAlgorithms::ssqe;
      // positive solutions of quadratic equations
   if (r==0){
      cpmwarning("Iv BndSph::hit(...): unexpected r==0");
      return Iv();
   }
   Vec v=x-x0;
   R d=v.absVal();
   if (d==0) return Iv(0,r);
```

```
// now d>0
   R sp=(n|v);
   R d1=d-r;
   if (d1>0){ // x0 outside *this
         // here analytic geometry works simpler than trigonometry
      if (sp<=0) return Iv(); // no chance for contact
         // dInv not needed in this case
      R dInv=1./d;
      R cs=sp*dInv; // 0<cs<=1
      R si=sqrtS(1-cs*cs);
      if (si>=r*dInv) return Iv();
         // now cs farer away from 0
      R csInv=1./cs;
      R tg=csInv*sqrtS(1-cs*cs); // sin/cos
// Now we solve y=tg*x and y^2+(x-d)^2=r^2 for x
      R a=1+tg*tg;
      R b=-2*d;
      R d2=d+r;
      R c=d1*d2;
      R x1=ssqe(a,b,c,d1,d2);
      R x2=c/(a*x1); // Vieta
      return Iv(x1*csInv,x2*csInv,true);
   }
   else{ // x0 inside, using sine theorem and cosine theorem of
         // trigonometry implies 3 transcendental function calls
      R cs=sp/d;
      R siAlpha=sqrtS(1-cs*cs);
      R siGamma=d*siAlpha/r; // sine theorem; notice d<r
      R beta=Pi-asin(siAlpha)-asin(siGamma);
      R t=sqrtS(d*d+r*r-2*d*r*cos(beta));
      return Iv(0,t,true);
   }
}

X2<V<Spc>,Z> BndSph::ran(Z n, const F<Spc,bool>& cond, Z tryMax)const
{
   V<Spc> res(n);
   Z ir=137;
   using CpmRoot::ranVal;
   Rdim a;
   Spc xTest;
   Z d=Spc::dimension();
   Z iTested=0;
   Z iFound=0;
   R a0=1;
   while(iFound<n && iTested<tryMax){
      bool aValid=false;
      while (!aValid){
         for (Z j=1;j<=d;j++){
            a[j]=ranVal(a0,ir++);
```

```
                //
                // between -1 and 1
            }
            aValid=(a.absVal()<1);
         }
         xTest=x+r*Vec(a);

         if (cond(xTest)){
            iFound++;
            res[iFound]=xTest;
         }
         iTested++;
      }
      if (iFound<n) res=res.resize(iFound);
      return X2<V<Spc>,Z>(res,iTested);
}

X2<Z,Z> BndSph::ranLean(Z n, const F<Spc,bool>& cond, Z tryMax)const
// doublicated code from previous function, all statements refering to
// array res are eliminated, return structure modified.
// Keep in touch with ran, if this should be modified!!!
{
   Z ir=137;
   using CpmRoot::ranVal;
   Rdim a;
   Spc xTest;
   Z d=Spc::dimension();
   Z iTested=0;
   Z iFound=0;
   R a0=1;
   while(iFound<n && iTested<tryMax){
      bool aValid=false;
      while (!aValid){
         for (Z j=1;j<=d;j++){
            a[j]=ranVal(a0,ir++);
               // between -1 and 1
         }
         aValid=(a.absVal()<1);
      }
      xTest=x+r*Vec(a);
      if (cond(xTest)==true){
         iFound++;
      }
      iTested++;
   }
   // at the end either iTested==tryMax or iFound==n
   return X2<Z,Z>(iFound,iTested);
}

///////////////// struct DisPar ////////////////////////////////
```

```
bool DisPar::prnOn(ostream& out)const
{
   if (!CpmRoot::writeTitle("DisPar",out)) return false;
   if (!dr.prnOn(out)) return false;
   if (!ds.prnOn(out)) return false;
   if (!nFind.prnOn(out)) return false;
   if (!nMax.prnOn(out)) return false;
   return true;
}

bool DisPar::scanFrom(istream& in)
{
   if (!dr.scanFrom(in)) return false;
   if (!ds.scanFrom(in)) return false;
   if (!nFind.scanFrom(in)) return false;
   if (!nMax.scanFrom(in)) return false;
   return true;
}

//////////////////// SubsetOfSpace ////////////////////////////////

namespace{ // some tools

 Rdim fss(const Rdim& x, const F<Spc,Spc>& f, const RefSys& rs)
   {
      return rs(f(rs(x)));
   }

 R fsr(const Rdim& x, const F<Spc,R>& f, const RefSys& rs)
   { return f(rs(x));}


 R tf(const Spc& x, const F<Rdim,R>& fr, const RefSys& rs)
   {
      return fr(rs(x));
   }

 R supFunc(const Rdim& x, const F<Spc,R>& f1, const F<Spc,R>& f2)
   {
      Spc sp(x);
      return CpmRootX::sup(f1(sp),f2(sp));
   }

 R infFunc(const Rdim& x, const F<Spc,R>& f1, const F<Spc,R>& f2)
   {
      Spc sp(x);
      return CpmRootX::inf(f1(sp),f2(sp));
   }
```

```
 R comFunc(const Rdim& x, const F<Spc,R>& f1, const F<Spc,R>& f2)
   {
      Spc sp(x);
      return CpmRootX::sup(f1(sp),-f2(sp));
   }

 R symDiffFunc(const Rdim& x, const F<Spc,R>& f1, const F<Spc,R>& f2)
   {
      Spc sp(x);
      R f1s=f1(sp);
      R f2s=f2(sp);
      R val1=CpmRootX::sup(f1s,-f2s);
      R val2=CpmRootX::sup(-f1s,f2s);
      return CpmRootX::inf(val1,val2);
   }

} // namespace

SubsetOfSpace::SubsetOfSpace(const F<Spc,R>& f_, const RefSys& rs_,
   const Spc& x_, R r_, const DisPar& dp_):
   f(f_),rs(rs_),x(x_),r(r_),dp(dp_)
{
   fr=F2<Rdim,F<Spc,R>,RefSys,R>(f,rs)(fsr);
}


void SubsetOfSpace::update()
{
   f=F2<Spc,F<Rdim,R>,RefSys,R>(fr,rs)(tf);
}

SubsetOfSpace SubsetOfSpace::operator &(const SubsetOfSpace& sos)const
{
   BndSph bs(x,r);
   BndSph bssos(sos.x,sos.r);
   BndSph res=bs&bssos;
   F<Rdim,R> f1r=F2<Rdim,F<Spc,R>,F<Spc,R>,R>(f,sos.f)(supFunc);
   return SubsetOfSpace(f1r,RefSys(),res.get_x(),res.get_r());
}

SubsetOfSpace SubsetOfSpace::operator |(const SubsetOfSpace& sos)const
{
   BndSph bs(x,r);
   BndSph bssos(sos.x,sos.r);
   BndSph res=bs|bssos;
   F<Rdim,R> f1r=F2<Rdim,F<Spc,R>,F<Spc,R>,R>(f,sos.f)(infFunc);
   return SubsetOfSpace(f1r,RefSys(),res.get_x(),res.get_r());
}

SubsetOfSpace SubsetOfSpace::operator ^(const SubsetOfSpace& sos)const
```

```
{
   BndSph bs(x,r);
   BndSph bssos(sos.x,sos.r);
   BndSph res=bs|bssos;
   F<Rdim,R> f1r=F2<Rdim,F<Spc,R>,F<Spc,R>,R>(f,sos.f)(symDiffFunc);
   return SubsetOfSpace(f1r,RefSys(),res.get_x(),res.get_r());
}

SubsetOfSpace SubsetOfSpace::operator -(const SubsetOfSpace& sos)const
      // forming *this\sos i.e. the set of all those elements of
      // *this which don't belong to sos (relative complement)
{
   F<Rdim,R> f1r=F2<Rdim,F<Spc,R>,F<Spc,R>,R>(f,sos.f)(comFunc);
   return SubsetOfSpace(f1r,RefSys(),x,r);
}

SubsetOfSpace SubsetOfSpace::combine(const V<SubsetOfSpace>& vs)const
         // here we consider *this as a region of interest and return a
         // body which is formed out of all intersections vs[i]&*this
         // Those terms for which this intersection is void, are left
         // out of the result for efficiency.
{
   Z d=vs.dim();
   V<bool> vb(d);
   V<SubsetOfSpace> vsi(d);
   for (Z i=1;i<=d;i++){
      SubsetOfSpace soi=vs[i]&(*this);
      soi.improveBoundingSphere();
         // to see wether it is void, this can't be seen from the
         // defining function directly
      vsi[i]=soi;
      vb[i]=(soi.r>0);
   }
   V<SubsetOfSpace> res=vsi.select(vb);
   return unionOfArray(res);
}

namespace{
   R unionFunc(const Rdim& xr, const V<SubsetOfSpace>& vf)
   {
      Spc x(xr);
      Z d=vf.dim();
      CpmArrays::R_Vector rv(d);
      for (Z i=1;i<=d;i++) rv[i]=(vf[i])(x);
      return rv.sup();
   }

   R sectionFunc(const Rdim& xr, const V<SubsetOfSpace>& vf)
   {
      Z d=vf.dim();
```

```
      Spc x(xr);
      CpmArrays::R_Vector rv(d);
      for (Z i=1;i<=d;i++) rv[i]=(vf[i])(x);
      return rv.inf();
   }
}

SubsetOfSpace unionOfArray(const V<SubsetOfSpace>& vs)
{
   Z i,d=vs.dim();
   BndSph bs(vs[1].x,vs[1].r);
   for (i=2;i<=d;i++) bs=bs|BndSph(vs[i].x,vs[i].r);
      // we don't use this logic for defining the indicator function of
      // the union although it is not certain that it would not be
      // efficient
   F<Rdim,R> ff=F1<Rdim, V<SubsetOfSpace>, R>(vs)(unionFunc);
      // heavy burden for the F1-template; forms a function out of a
      // sequence of functions;
   return SubsetOfSpace(ff,RefSys(),bs.get_x(),bs.get_r());
}

SubsetOfSpace sectionOfArray(const V<SubsetOfSpace>& vs)
{
   Z i,d=vs.dim();
   BndSph bs(vs[1].x,vs[1].r);
   for (i=2;i<=d;i++) bs=bs&BndSph(vs[i].x,vs[i].r);
      // we don't use this logic for defining the indicator function of
      // the union although it is not certain that it would not be
      // efficient
   F<Rdim,R> ff=F1<Rdim, V<SubsetOfSpace>, R>(vs)(sectionFunc);
      // heavy burden for the F1-template; forms a function out of a
      // sequence of functions
   return SubsetOfSpace(ff,RefSys(),bs.get_x(),bs.get_r());
}

void SubsetOfSpace::operator *=(const Group& g)
{
   rs*=g;
   x*=g;
   update();
}

void SubsetOfSpace::operator *=(const F<Spc,Spc>& f_)
{
   f.leftCombine(f_);
// fr_: action of f_ on coordinates with respect to rs
   F<Rdim,Rdim> fr_=F2<Rdim,F<Spc,Spc>,RefSys,Rdim>(f_,rs)(fss);
   fr.leftCombine(fr_);
      // function update would only set f according to fr
      // in our case setting directly is much cheaper
```

```
}

namespace{

   bool fComb(const R& t, const Spc& x0, const Vec& v,
      const F<Spc,R>& f, const Z& k)
      // the most natural implementation turns out to be the fastest!
      // So, no reason to avoid the semantic complexity to have a
      // F<>-object as an argument of a function to be used for creating
      // another F<>-object.
   {
      return k*f(x0+t*v)>0;
   }

    R fComb2(const R& t, const Spc& x0, const Vec& v,
      const F<Spc,R>& f)
      // the most natural implementation turns out to be the fastest!
      // So, no reason to avoid the semantic complexity to have a
      // F<>-object as an argument of a function to be used for creating
      // another F<>-object.
   {
      return f(x0+t*v);
   }

} // namespace

namespace{
   Spc ray(const R& r, const Spc& x0, const Vec& v)
   { return x0+r*v;}

   bool fInterior(R x){ return x<0;}
}

X2<Spc,bool> SubsetOfSpace::hit(const Spc& xi, const Vec& v,
                                   R sFine, R sCoarse)const
{
   Z mL=3;
   Word loc("SubsetOfSpace::hit(Spc,Vec,R,R)");
   CPM_MA
   cpmassert(sFine>0,loc);
   cpmassert(sCoarse>=sFine,loc);
   BndSph bs(x,r);
   Iv iv=bs.hit(xi,v);
   if (iv.isVoid()){
      cpmdata<<endl<<"iv.isVoid(), trivial result returned";
      return X2<Spc,bool>(Spc(),false);
   }
   R xMin=iv.inf()-sFine;
   R xMax=iv.sup()+sCoarse+sFine;
      // not understood why '+sCoarse' is not always sufficient
```

```
      // numerics???
   F<R,Spc> fs=F2<R,Spc,Vec,Spc>(xi,v)(ray);
   bool silent=true;
   X2<R,bool> prr=CpmAlgorithms::root(fs&f,xMin,xMax,sFine,sCoarse,
      silent);
      // for silent==false one would get always warnings if *this
      // would not be in the field of view of the object providing
      // xi and v
   Spc xLast=ray(prr.c1(),xi,v);
   CPM_MZ
   return X2<Spc,bool>(xLast,prr.c2());
}

R SubsetOfSpace::vol()const
{
   Z nMax=dp.nMax;
   Z nFind=dp.nFind;
   BndSph bs(x,r);
   X2<Z,Z> pzz=bs.ranLean(nFind,f&F<R,bool>(fInterior),nMax);
   return (bs.vol()*pzz.first())/pzz.second();
}

V<Spc> SubsetOfSpace::ran()const
{
   Z nMax=dp.nMax;
   Z nFind=dp.nFind;
   BndSph bs(x,r);
   X2<V<Spc>,Z> pvz=bs.ran(nFind,f&F<R,bool>(fInterior),nMax);
      // here we use limitation by nLarge
   return pvz.c1();
}

R SubsetOfSpace::improveBoundingSphere(Z n,Z tryMax)
{
   const R saftyFactor=1.1;
      // muliplies the radius that we found. Takes into account that
      // by definition, the true radius is >= the one we can find by the
      // following method
   using CpmArrays::Vo;
   BndSph bs(x,r);
   X2<V<Spc>,Z> pvz=bs.ran(n,f&F<R,bool>(fInterior),tryMax);
   V<Spc> xa=pvz.c1(); // x-array
   Z m=xa.dim();
   if (m==0){
      r=0; return 0;
         // no point found which belongs to the set *this.
   }
   R ret=((R)(m))/pvz.c2();
   Z mp=m*(m-1);
   mp/=2;
```

```
   Vo<R> dis(mp);
   Z jp=1,i,j;
   for (i=1;i<=m;i++){
      for (j=i+1;j<=m;j++){
         dis[jp++]=(xa[i]-xa[j]).absVal();
      }
   }
   Z jpMax=dis.indInfSup().second;
   r=0.5*dis[jpMax]*saftyFactor;
// retrieving i and j in a simple and safe manner. The obvious lack
// of utmost efficiency dosn't matter compared to the workload
// of the other operations
   jp=1;
   for (i=1;i<=m;i++){
      for (j=i+1;j<=m;j++){
         if (jp==jpMax) goto done;
         jp++;
      }
   }
done:  // now we know i and j belonging to jp
   V<Spc> xp("",xa[i],xa[j]);
   V<R> wp("",0.5,0.5);
   x=Spc::convex(xp,wp,false);
   return ret;
}

Plane SubsetOfSpace::tangentialPlane(const Spc& x)const
{
   R d=f(x);
   Vec n=grad(x,f,dp.dr);
   Spc xBase=x-n*d;
   return Plane(n,xBase);
}

X2<R,Vec> SubsetOfSpace::distAndDirection(const Spc& x)const
// Re-implementation 2006-02-15 in order to avoid that the axis of
// a Drum is seen by particles as an elastic obstacle
{
   Word loc("SubsetOfSpace::distAndDirection(Spc)");
   const R hDef=1e-6; // should never be used, using it
   const R gradLengthLimit=3.;
      // creates a warning and a failing assert
   R h=dp.dr;
   if (h==0) h=dp.ds;
   cpmassert(h>0,loc);
   if (h==0){
      h=hDef;
      cpmwarning(loc&": dp.dr==0, dp.ds==0, thus h set to "&cpm(h));
   }
   Vec n=grad(x,f,h);
```

```
   R gradLength=n.normalize();
   Spc xTest=x;
   Z wc=0;
   Z iTest=1;
   Z d; V<Vec> b;
   while (gradLength==0.){ // searching a point nearby
      // for which the grad is not 0
      wc++;
      cpmwarning(loc&": gradLength==0, whileCount="&cpm(wc));
      if (wc==1){
         Vec v;
         v[1]=1;
         b=v.base();
         d=b.dim();
      }
      Spc p=xTest+b[iTest]*h;
      if (iTest==d){
         iTest=1;
         xTest=p;
      }
      else{
         iTest++;
      }
      n=grad(p,f,h);
      gradLength=n.normalize();
   }
   if (gradLength>gradLengthLimit) gradLength=gradLengthLimit;
      // if gradLength is very large, the value of
      // f(x)*cpminv(gradLength) is very small and thus simulates
      // a point near the surface, which might not be correct
   return X2<R,Vec>(f(x)*cpminv(gradLength),n);
}

void SubsetOfSpace::shiftExternalPointToSurface4(Spc& x, R& dOut,
   R& fxBefore, R& fxAfter)const
{
   fxBefore=f(x);
   if (fxBefore<=0){ // then the point is internal
      dOut=0;
      fxAfter=fxBefore;
   }
   else{
      X2<R,Vec> prv=distAndDirection(x);
      dOut=prv.c1();
      Vec shift=prv.c2()*(-dOut);
      x+=shift;
      fxAfter=f(x);
   }
}
```

```
bool SubsetOfSpace::prnOn(ostream& str)const
{
   if (!CpmRoot::writeTitle("SubsetOfSpace",str)) return false;
   if (!rs.prnOn(str)) return false;
   if (!x.prnOn(str)) return false;
   if (!r.prnOn(str)) return false;
   if (!dp.prnOn(str)) return false;
   return true;
}

namespace{

   R bandFunc(const Rdim& x, const Z& i, const Iv& iv){ return iv(x[i]);}

   R boxFunc(const Spc& x, const Rdim x0, const Rdim& s0,
      const F<Spc,Rdim>& f )
   {
      Rdim y=f(x);
      return SubsetOfSpace::box(y,x0,s0);
   }

   R sphereFunc(Spc const& p, Spc const& p0, R const& r)
   { return (p-p0).absVal()-r;}
}

R SubsetOfSpace::box(const Rdim& x, const Rdim& x0, const Rdim& s0)
{
   Z d=Spc::dimension();
   Rdim val;
   for (Z i=1;i<=d;i++){
      R hw=cpmabs(s0[i]);
      R c=x0[i];
      Iv iv(c-hw,c+hw);
      val[i]=iv(x[i]);
   }
   return val.sup();
}

// static generating function
SubsetOfSpace SubsetOfSpace::band(Z i, const Iv& iv, const RefSys& rf)
{
   F<Rdim,R> f=F2<Rdim,Z,Iv,R>(i,iv)(bandFunc);
   return SubsetOfSpace(f,rf);
}

SubsetOfSpace SubsetOfSpace::boxoid(const Rdim& x0, const Rdim& s0,
   const F<Spc,Rdim>& f )
{
   const R tol=1e-5;
   const R sec=4; // security
```

```
   Z mL=3;
   Word loc("SubsetOfSpace::boxoid(...)");
   CPM_MA
   F<Rdim,R> f1=F2<Rdim,Rdim,Rdim,R>(x0,s0)(box);
   F<Spc,R> f2=f&f1;
   SubsetOfSpace res(f2);
   Z i;
   for (i=1;i<=dimVec;++i){
      if (s0[i]==0.){
         CPM_MZ
         return res;
            // res has radius 0 by definition
      }
   }
// In order to set r properly, we test whether f determines
// fixed magnification factors along the master axes
/******************************/
   Rdim trueAxes;
   V<Rdim> vv=Rdim::canBas();
   RefSys rs=res.getRefSys();
   bool distortion=false;
   for (i=1;i<=dimVec;++i){
      R si=cpmabs(s0[i]);
      cpmassert(si>0,loc);
      R siInv=1./si;
      Rdim v1=vv[i]*si;
      Rdim v2=v1*2.;
      Spc p1=rs(v1);
      Spc p2=rs(v2);
      Spc m1=rs(-v1);
      Spc m2=rs(-v2);
      R2 mag;
      mag[1]=0.5*f(p1).dist(f(m1))*siInv;
      mag[2]=0.25*f(p2).dist(f(m2))*siInv;
      R fi=mag.meanValue();
      cpmassert(fi>0,loc);
      trueAxes[i]=si/fi;
      if (mag.mod()>tol) distortion=true;
   }
   R radius=trueAxes.absVal();

   if (!distortion){
      res.set_r(radius);
   }
   else{
      res.set_r(sec*radius);
      res.improveBoundingSphere();
   }
/****************************************/
   CPM_MZ
```

```
   return res;
}

SubsetOfSpace SubsetOfSpace::sphere(Spc const& x0, R r0, R accRel)
{
   F<Spc,R> f=F2<Spc,Spc,R,R>(x0,r0)(sphereFunc);
   RefSys rs;
   R dis=r0*accRel;
   DisPar dp(dis,dis);
   return SubsetOfSpace(f,rs,x0,r0,dp);
}

///////////////// class Texture ////////////////////////////////////////

namespace {

R fAux1(const Spc& sp, const V<R>& omega, const V<R>& delta,
   const R& p, const Z& method)
{
   Z i,d=Spc::dimension();
   R dInv=1./d;
   R y=0;
   Rdim yy;
   for (i=1;i<=d;i++){
      yy[i]=0.5*(1+cos(delta[i]+sp[i]*omega[i]));
   }
   if (method==1){ // additive method
      for (i=1;i<=d;i++) y+=yy[i];
      y*=dInv;
   }
   else if (method==2){ // extended additive method
      for (i=1;i<=d;i++) y+=pow(yy[i],p);
      y*=dInv;
      y=pow(y,R(1.)/p);
   }
   else { // multiplicative method
      y=1;
      for (i=1;i<=d;i++) y*=yy[i];
   }

   return y-floor(y);
}

R fAux2(const Spc& sp, const CpmFunctions::R2_Func& r2f, const Iv& iv1,
        const Iv& iv2)
{
   C z(sp[1],sp[2]);
   R a1=iv1.periodic(sp[1]);
   R a2=iv2.periodic(sp[2]);
   return r2f(C(a1,a2));
```

```
}

R fAux3(const Spc& sp, const F<C,R>& fCR)
{
   return fCR(C(sp[1],sp[2]));
}

} // anonymous namespace

Texture::Texture(const V<R>& omega, const V<R>& delta, R p, Z method):
Base(F4<Spc,V<R>,V<R>,R,Z,R>(omega,delta,p,method)(fAux1))
{}

Texture::Texture(R width)
{
   R p=1;
   Z method=1;
   R omegaR=2*CpmGeo::Pi*cpminv(width);
   R deltaR=0;
   Z d=Spc::dimension();
   V<R> omega(d,omegaR);
   V<R> delta(d,deltaR);
   *this=
      Type(Base(F4<Spc,V<R>,V<R>,R,Z,R>(omega,delta,p,method)(fAux1)));
}

Texture::Texture(const VV<R>& mat, Iv xI, Iv yI, Z method):
Base(F3<Spc,R2_Func,Iv,Iv,R>(R2_Func(mat,method),xI,yI)(fAux2))
{}

Texture::Texture(const F<C,R>& fIn):
Base(F1<Spc,F<C,R>,R>(fIn)(fAux3))
{}

//////////////////////////////// class Triangle ///////////////////////////

Triangle& Triangle::operator *=(Group const& g)
{
   p1_*=g;
   p2_*=g;
   p3_*=g;
   return *this;
}

Triangle& Triangle::operator +=(Vec const& a)
{
   p1_+=a;
   p2_+=a;
   p3_+=a;
   return *this;
```

```
}

bool Triangle::prnOn(ostream& str)const
{
   cpmwat;
   cpmp(p1_);
   cpmp(p2_);
   cpmp(p3_);
   return true;
}

bool Triangle::scanFrom(istream& str)
{
   cpms(p1_);
   cpms(p2_);
   cpms(p3_);
   return true;
}


namespace{

V<C> circlePoints(R r, Z n)
// generating a regular n-gon ( a polygon, poly=n)
// in the complex plane. Let res be the result, then
// res[1] and res[n] are different; res[1] follows
// res[n] just as res[2] follows res[1]. The case n=2
// is useful for clarification.
{
   R phi=2*Pi/n; // phi=Pi for n==2
   C z1(r,0.);
   C fac(1.,phi,"polar"); //e^(i*phi)
   V<C> p(n);
   p[1]=z1;
   Z i;
   for (i=1;i<n;i++){
      p[i+1]=p[i]*fac;
   }
   return p;
}

} //anon

//////////////////////// Path ////////////////////////////////////////

R Path::update()
{
   n_=x_.dim();
   y_=Vo<R>(n_,0.);
   if (n_<=1){
```

```
      length_=0;
      l_=V<R>(0);
      e_=V<Vec>(0);
      return 0;
   }
   else { // here n_-1 >=1
      R gap=CpmRootX::hugeNumber;
      length_=0;
      Z n=n_-1;
      l_=V<R>(n);
      e_=V<Vec>(n);
      for (Z i1=1,i2=2;i2<=n_;++i1,++i2){
         Vec ei=x_[i2]-x_[i1];
         R li=ei.normalize();
         if (li<gap) gap=li;
         e_[i1]=ei;
         l_[i1]=li;
         length_+=li;
         y_[i2]=length_;
      }
      return gap;
   }
}

bool Path::isSmooth()const
{
   if (n_<=2) return true;
// now n_>=3
   for (Z i=1;i<=(n_-2);++i){
      R scp=(e_[i]|e_[i+1]);
      if (scp<=0){
         cpmdebug(x_[i]);
         cpmdebug(x_[i+1]);
         cpmdebug(x_[i+2]);
            // to help diagnosis
         return false;
      }
   }
   return true;
}

Z Path::adjust(R acc)
{
   const R reducFac=0.25;
      // should be <0.5. By this factor the smallest inter-point distance
      // is multiplied for obtaining acc
   Z np=x_.dim();
   if (np==0){
      update();
      return 0;
```

```
   }
      // then length and n are initialized to 0 anyway
   if (acc>0){
      acc_=acc;
      V<bool> sel(np,false);
      sel[1]=true;
      length_=0;
      for (Z i=2;i<=np;++i){
         R dis=(x_[i]-x_[i-1]).absVal();
         if (dis>=acc_){
            sel[i]=true;
            length_+=dis;
         }
      }
      x_=x_.select(sel);
      R gap=update();
      Z leftOut=np-n_;
      return leftOut;
   }
   else{
      acc_=update();
      acc_*=reducFac;
      return 0; // no point left out
   }
}

namespace{
   Word mesDir(
   "result vector probably too short for defining a useful direction");
}

Vec Path::firstDir()const
{
   Word loc("Path::firstDir()");
   if (n_<2) return Vec();
   else return e_[1];
}

Vec Path::lastDir()const
{
   Word loc("Path::lastDir()");
   if (n_<2) return Vec();
   else return e_[n_-1];
}

Vec Path::dir(Z i)const
{
   Word loc("Path::dir()");
   Vec res;
   if (n_<2 || i<1 || i>n_) return res;
```

```
   else if (i==1) return firstDir();
   else if (i==n_) return lastDir();
   else{
      Z ic1=i-1;
      Z ic2=i+1;
      R dis;
again:
      res=x_[ic2]-x_[ic1];
      dis=res.absVal();
      if (dis<acc_){
         bool suc=false;
         if (ic1>1){ ic1--; suc=true;}
         if (ic2<n_){ ic2++; suc=true;}
         if (suc==false){ // there are no components available
            cpmwarning(loc,mesDir);
            return res;
         }
         goto again;
      }
      return res;
   }
   return res; // to avoid warning
}

Path& Path::operator *=(Group const& g)
{
   for (Z i=1;i<=n_;i++) x_[i]*=g;
   update(); // may change directions
   return *this;
}

Path& Path::operator +=(Vec const& a)
{
   for (Z i=1;i<=n_;i++) x_[i]+=a;
      // does not change directions
   return *this;
}

Path& Path::operator *=(AxVec const& A)
{
   Spc fix=x_[1];
   for (Z i=2;i<=n_;i++) x_[i].rot_(fix,A);
   update();
   return *this;
}

Path& Path::operator *=(R fac)
{
   Spc fix=x_[1];
   for (Z i=2;i<=n_;i++) x_[i]=x_[i].scl(fix,fac);
```

```
   update();
   return *this;
}

Path& Path::rot_(Spc const& fix, AxVec const& A)
{
   for (Z i=1;i<=n_;i++) x_[i].rot_(fix,A);
   update();
   return *this;
}

Path Path::scl(Spc const& fix, R fac)const
{
   V<Spc> x(n_);
   for (Z i=1;i<=n_;i++) x[i]=x_[i].scl(fix,fac);
   return Path(x);
}

V<Path> Path::thc(R r, Z n)const
{
   if (r<=0 || n<=1) return V<Path>("",*this);
   Spc o;
   Path disk=Path::circle(o,r,n);
   V<Spc> cir=disk();
   Z nc=cir.dim(); // paranoia, should be n
   V<Vec> vcir(nc);
   Z i;
   for (i=1;i<=nc;++i) vcir[i]=cir[i]-o;
   V<Path> res(nc,*this);
   for (i=1;i<=nc;++i){
      res[i]+=vcir[i];
   }
   return res;
}

V<Path> Path::brkLine(R d, R frac)const
{
   Z mL=3;
   Word loc("Path::brkLine(R,R)");
   CPM_MA
   if (d<=0 || frac<=0 || frac >=1){
      CPM_MZ
      return V<Path>("",*this);
   }
   R l=len();
   R dAdj=d;
   Z n=CpmAlgorithms::getNerFrc_1(dAdj,l);
   // now the total length consists of n pieces, each of length dAdj.
   // We get two space points for each of these n pieces which we store
   // for later use.
```

```
   R s1=dAdj*(1-frac)*0.5;
   R s2=dAdj-s1;
   // designed such that s2-s1=dAdj*frac. The interval
   // from s1 to s2 is the part of a segment of length dAdj
   // where the line is invisible (i.e. not drawn)
   cpmassert(s1>0,loc);
   cpmassert(s2>0,loc);
   Z n2=n*2;
   V<Spc> vs(n2);
   R t=0; // arc length parameter for defining proper division points
   Z i,j=1;
   typedef X3<Spc,Vec,Z> x3; // return type of function fromArc
   for (i=1;i<=n;++i){
      R t1=t+s1;
      R t2=t+s2;
      x3 a1=fromArc(t1);
      vs[j++]=a1.c1();
      x3 a2=fromArc(t2);
      vs[j++]=a2.c1();
      t+=dAdj;
   }
   V<Path> res(n+1);
   j=1;
   res[1]=Path(firstPoint(),vs[j]);
   for (i=2;i<=n;++i){
      Spc p1=vs[++j];
      Spc p2=vs[++j];
      res[i]=Path(p1,p2);
   }
   res[n+1]=Path(vs[n2],lastPoint());
   CPM_MZ
   return res;
}

VecGraph Path::flatten()const
{
   V<C> vr2(n_);
   for (Z i=1;i<=n_;i++) vr2[i]=C(x_[i][1],x_[i][2]);
   Z connect=2;
   return VecGraph(vr2,connect);
}

Path Path::arc(const Spc& p1, const Spc& center,
   const AxVec& rot, Z n)
{
   pair<Angle,Vec> pav=rot.toAngAxs();
   Angle phi=pav.first;
   Vec axis=pav.second;
   V<Spc> ps(n);
   ps[1]=p1;
```

```
   cpmassert(n>1,"Path::arc");
   R nInv=1./(n-1);
   Angle dPhi=phi*nInv;
   Vec v=p1-center;
   Z i;
   for (i=2;i<=n;i++){
      v*=AxVec(dPhi,axis);
      ps[i]=center+v;
   }
   return Path(ps);
}

Path Path::arc2(const Spc& p1, const Spc& aimPoint,
      const AxVec& rot, Z n)
{
   Z mL=3;
   Word loc("Path::arc2(...)");
   CPM_MA
   Path res;
   Angle phi=rot.toAngAxs().first;
   if (phi==0.){
      res=line(p1,aimPoint,n);
   }
   else{
      Vec dir=aimPoint-p1;
      R length=dir.normalize();
      Vec dir2=dir;
      dir2*=rot;
      X2<Vec,Vec> pvv=dir.tanNormDec(dir2);
      Vec orth=pvv.c2();
      orth.normalize();
      R r=cpmabs(length/phi); // abs important for negative angles
      Spc center=p1+orth*r;
      res=arc(p1,center,rot,n);
   }
   CPM_MZ
   return res;
}

Path::Path(Spc const& p1, Vec const& v1, Spc const& p2,
      Vec const& v2, Z n)
{
   Z mL=3;
   Word loc("Path(Spc,Vec,Spc,Vec,Z)");
   CPM_MA
   X4<Spc,R,R,R> meet=Spc::meet(p1,v1,p2,v2);
   R a1=meet.c3();
   R a2=meet.c4();
   cpmassert(a1>0,loc);
   cpmassert(a2<0,loc);
```

```
   Spc cp=meet.c1(); // control point
   *this=bezier(V<Spc>("",p1,cp,p2),n);
   CPM_MZ
}

Path::Path(Spc const& p1, Vec const& v1, Spc const& p2,
      Angle const& phi, Z n)
{
   Z mL=3;
   Word loc("Path(Spc,Vec,Spc,AxVec,Z)");
   CPM_MA
   AxVec av=v1.cross(p2-p1);
   av.phi_(phi);
   Vec v2=v1*av;
   *this=Path(p1,v1,p2,v2,n);
   CPM_MZ
}

Path::Path(Spc const& p1, Spc const& p2, Vec const& v2, R s, Z n)
{
   Z mL=3;
   Word loc("Path(Spc,Spc,Vec,R,Z)");
   CPM_MA
   R vv=(v2|v2);
   cpmassert(vv>0,loc);
   Vec dp=p2-p1;
   R pv=(dp|v2);
   R x=pv/vv;
   Spc q=p2-v2*x;
   Spc pc=p2.versus(q,s);
   *this=bezier(V<Spc>("",p1,pc,p2),n);
   CPM_MZ
}

Path Path::bezier(V<Spc> const& vs, Z np)
// essentially taken from function Skeleton::bezier
{
   Z mL=3;
   Word loc("Path::bezier(V<Spc>,Z)");
   CPM_MA
   Z i,j,vsd=vs.dim();
   V<R_Vector> vvr(dimVec, R_Vector(vsd));
   for (i=1;i<=dimVec;++i) for (j=1;j<=vsd;++j) vvr[i][j]=vs[j][i];
   V<R_Func> vrf(dimVec);
   R2 iv(0,1);
   for (i=1;i<=dimVec;++i) vrf[i]=R_Func::Bernstein(vvr[i],iv);
   V< Spc > ps(np);
   R du=1./(np-1);
   R u=0;
   for (i=1;i<=np;++i){
```

```
      for (j=1;j<=dimVec;++j){
         ps[i][j]=(vrf[j])(u);
      }
      u+=du;
   }
   CPM_MZ
   return Path(ps);
}

Path Path::locInt(V<Spc> const& vs, Z n)
{
   Z mL=3;
   Word loc("Path::locInt(V<Spc>,Z)");
   CPM_MA
   bool d3=(Spc::dimension()==3);
   Z i,vd=vs.dim();
   V<R> x(vd);
   V<R> y(vd);
   V<R> z= d3 ? V<R>(vd) : V<R>(0);
   V<R> r(vd);
   for (i=1;i<=vd;++i){
      Spc pi=vs[i];
      x[i]=pi[1];
      y[i]=pi[2];
      if (d3) z[i]=pi[3];
      r[i]=i;
   }
   V<Spc> res(n);
   cpmassert(n>1,loc);
   R fac=(vd-1.)/(n-1.);
   IpolControl c("second_order");
   for (i=1;i<=n;++i){
      R ic=1+(i-1)*fac;
      R xi,yi,zi;
      xi=CpmArrays::locIpol(ic,r,x,c);
      yi=CpmArrays::locIpol(ic,r,y,c);
      if (d3) zi=CpmArrays::locIpol(ic,r,z,c);
      Spc pi;
      pi[1]=xi;
      pi[2]=yi;
      if (d3) pi[3]=zi;
      res[i]=pi;
   }
   CPM_MZ
   return Path(res);
}

namespace{
//See Aumann,Spitzmueller:Computerorientierte Geometrie, BI 1993
// p. 330, (11.7) for this formula:
```

```
   Spc cub(R const& x, Spc const& p1, Vec const& v1,
      Spc const& p2, Vec const& v2) // 0 <= x <= 1 assumed
   {
      Vec v0=p2-p1;
      R x2=x*x;
      R xm=x-1;
      R f0=x2*(3.-x*2.);
      R f1=x*xm*xm;
      R f2=x2*xm;
      // f0=F^2(u), f1=F^3(u), f2=F^4(u) in the notation of the
      // book mentioned above
      return p1+v0*f0+v1*f1+v2*f2;
   }
}

Path Path::cubCur(Spc const& p1, Vec const& v1,
      Spc const& p2, Vec const& v2, Z n)
{
   Z mL=3;
   Word loc("Path::cubCur(Spc,Vec,Spc,Vec,Z)");
   CPM_MA
   cpmassert(n>1,loc);
   V<Spc> ps(n);
   ps[1]=p1;
   ps[n]=p2;
   if (n>2){ // this is the perfectly normal case
      R dx=1./n;
      R xi=dx;
      for (Z i=2;i<n;++i){
         ps[i]=cub(xi,p1,v1,p2,v2);
         xi+=dx;
      }
   }
   CPM_MZ
   return Path(ps);
}

Path Path::cubCur(Spc const& p1, Vec const& v1, R k1,
      Spc const& p2, Vec const& v2, R k2, Z n)
{
   Vec e1=v1, e2=v2;
   e1.nor_();
   e2.nor_();
   R s=(p2-p1).absVal();
   Vec w1=e1*(s*k1);
   Vec w2=e2*(s*k2);
   return cubCur(p1,w1,p2,w2,n);
}

Path Path::waveLine(Spc const& p1, Spc const& p2, R ampl,
```

```
   R omega, Vec const& tr, Z n)
{
   if (n<=2) return Path(p1,p2);
   Word loc("Path::waveLine(Spc,Spc,R,R,Vec,Z)");
   Vec e=p2-p1;
   R l=e.nor_();
   X2<Vec,Vec> xvv=e.tanNormDec(tr);
   Vec tr1=xvv.c2();
   tr1.nor_();
   V<Spc> ps(n);
   R dt=l/(n-1);
   R t=0;
   cpmassert(omega!=0,loc);
   R halfWave=Pi/omega;
   Z nW=CpmAlgorithms::getNerFrc_1(halfWave,l);
   cpmassert(nW>0,loc);
   R omegaAct=Pi/halfWave;
   for (Z i=1;i<=n;++i){
      Spc pi=p1+e*t;
      R phase=omegaAct*t;
      R ai=ampl*sin(phase);
      Vec tri=tr1*ai;
      ps[i]=pi+tri;
      t+=dt;
   }
   return Path(ps);
}

Path Path::line(const Spc& pi, const Spc& pf, Z n)
{
   static Word loc("Path::line(Spc,Spc,Z)");
   cpmassert(n>1,loc);
   V<Spc> ps(n);
   ps[1]=pi;
   ps[n]=pf;
   if (n>2){
      R frac=1./(n-1);
      Vec sh=(pf-pi)*frac; // translation vector from point to point
      for (Z i=2;i<n;i++) ps[i]=ps[i-1]+sh;
   }
   return Path(ps);
}

Path Path::parabola(const Spc& pi, const Spc& pc, const Spc& pf,
   Z n)
{
   static Word loc("Path::parabola(Spc,Spc,Spc,Z)");
   cpmassert(n>2,loc);
   V<Spc> ps(n);
   ps[1]=pi;
```

```
   ps[n]=pf;
   Vec v=(pc-pi)*2;
   Vec aHalf=(pf-pi)-v;
   R tStep=1./(n-1);
   R t=0;
   for (Z i=2;i<n;i++){
      t+=tStep;
      ps[i]=pi+(v+aHalf*t)*t;
   }
   return Path(ps);
}

Path Path::circle(const Spc& pc, R r, Z n)
{
   if (r==0.){
      cpmerror("Path::circle(...) vanishing radius");
   }
   V<C> p=circlePoints(r,n);
   Vec shift=pc-Spc();
   Z i;
   V<Spc> ps(n+1);
   for (i=1;i<=n;i++){
      ps[i]=Spc(p[i])+shift;
   }
   ps[n+1]=ps[1];
   return Path(ps);
}

Path Path::ellipse(const Spc& pc, R a1, R a2, Z n)
{
   if (a1==0.|| a2==0.){
      cpmerror("Path::ellipse(...) vanishing half-axis");
   }
   V<C> p=circlePoints(a1,n);
   Vec shift=pc-Spc();
   Z i;
   if (a2!=a1){
      R f=a2/a1;
      for (i=1;i<=n;i++) p[i][2]*=f;
   }
   V<Spc> ps(n+1);
   for (i=1;i<=n;i++){
      ps[i]=Spc(p[i])+shift;
   }
   ps[n+1]=ps[1];
   return Path(ps);
}

Path Path::quarterEllipse(Spc const& p1, Vec const& v1,
   Spc const& p2, Z n)
```

```
{
   Z mL=3;
   Word loc("Path::quarterEllipse(Spc,Vec,Spc,Z)");
   CPM_MA
   Vec dp=p2-p1;
   R sp=(dp|v1);
   cpmassert(sp!=0,loc);
   Vec v=v1;
   if (sp<0) v=-v;
   Vec vl=v;
   vl.nor_(); // longitudinal unit vector
   Vec dpl=vl*(dp|vl); // longitudinal component
   Vec dpt=dp-dpl; // transversal component
      // now dp=dpl+dpt
   R r=dpt.absVal();
   cpmassert(r>0,loc);
   Spc center=p1+dpt;
   R phi=Pi/(2*n);
   AxVec av;
   av=v.cross(dpt);
   av.phi_(Angle(phi,RAD)); // works also in 2D !
   V<Vec> vv(n);
   vv[1]=p1-center;
   Z i;
   for (i=2;i<=n;++i){
      vv[i]=vv[i-1]*av;
   }
   R fac=(p2-center).absVal()/r;
   Z np=n+1;
   V<Spc> ps(np);
   ps[1]=p1;
   for (i=2;i<=n;++i){
      Vec vi=vv[i];
      Vec vil=vl*(vl|vi);
      Vec vit=vi-vil;
      vil*=fac;
      ps[i]=center+vil+vit;
   }
   ps[np]=p2;
   Path res(ps);
   CPM_MZ
   return res;
}

Path Path::operator &(const Path& p)const
{
   R acc=acc_.inf(p.acc_);
   return Path(x_&p.x_,acc);
}
```

```
Path Path::moveAndMerge(const Path& p)const
{
   Z mL=3;
   Word loc("Path::moveAndMerge(Path)");
   CPM_MA
   Z d=dim();
   Z dp=p.dim();
   Path res;
   if (d==0) res=p;
   else if (dp==0){
       res=*this;
   }
   else if (d==1){
      Vec shift=lastPoint()-p.firstPoint();
      Path p1=p;
      p1*=Group(shift);
      res=p1;
   }
   else if (dp==1){
      res=*this;
   }
   else{
      Vec e=lastDir();
      Vec ep=p.firstDir();
      AxVec rot=ep.rotTo(e);
      Spc fP=p.firstPoint();
      Path p1=p;
      p1*=Group(fP,rot);
      Vec shift=lastPoint()-fP;
      p1+=shift;
      res=(*this)&p1;
   }
   CPM_MZ
   return res;
}

X2<Path,Path> Path::divideByArc(R t)const
{
   Path res1,res2; // void paths
   if (t<=0) res2=*this;
   else if (t>=length_) res1=*this;
   else{
      X3<Spc,Vec,Z> xsvz=fromArc(t);
      Z i1=xsvz.c3();
      Spc sp1=xsvz.c1();
      V<Spc> x1=x_.resize(i1)&sp1;
      res1=Path(x1,acc_);
      V<Spc> x2=x_.eli(1,i1).ins(1,sp1);
      res2=Path(x2,acc_);
   }
```

```
   return X2<Path,Path>(res1,res2);
}

Path Path::replace(const Spc& s1, const Spc& s2, const Path& p)const
{
   Z mL=3;
   Word loc("Path::replace(Spc,Spc,Path)");
   CPM_MA
   cpmassert(isSmooth(),loc);

   X3<Spc,Vec,R> eval1=project(s1);
   X3<Spc,Vec,R> eval2=project(s2);
   R t1=eval1.c3();
   R t2=eval2.c3();
   CpmRootX::order<R>(t1,t2);

   X2<Path,Path> xpp=divideByArc(t2);

   Path p1=xpp.c1(); // process further
   Path p2=xpp.c2(); // last part

   X2<Path,Path> xpp2=p1.divideByArc(t1);
   Path p0=xpp2.c1();  // first part

   Path resPrel=p0.moveAndMerge(p);
   Path res=resPrel.moveAndMerge(p2);

   CPM_MZ
   return res;
}

bool Path::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("Path",str);
   cpmp(x_);
   cpmp(acc_);
   return true;
}

bool Path::scanFrom(istream& str)
{
   cpms(x_);
   cpms(acc_);
   update();
   return true;
}

namespace{
   R dist2FromSegment1(Spc& s, const Spc& x1, const Vec& e, R d)
```

```
      // name end ...1 means that there are 1 mutable argument(s)
   {
      Vec vs=s-x1;
      R scp=(vs|e);
      Spc sProx;
      if (scp<=0){
         sProx=x1;
      }
      else if (scp>=d){
         sProx=x1+e*d;
      }
      else{
         sProx=x1+e*scp;
      }
      R res=(s-sProx).absSqr();
      s=sProx;
      return res;
   }
}

X3<Spc,Vec,R> Path::project(const Spc& s)const
{
   Z mL=3;
   Word loc("Path::project(Spc)");
   CPM_MA
   Spc sRes=s;
   Vec eRes;
   R lRes=0;
   if (n_==0){
      ;
      // then the path gives no information
      // that allows to create a non-trivial projected image from s
   }
   else if (n_==1){
      sRes=x_[1];
         // since the path has only one point, the projected image of
         // s has to be this point.
   }
   else{
      Z m=n_-1;
      V<Spc> vsProx(m);
      Vo<R> dis2(m); // list of squares of distances of s from
         // seqment x_[i],x_[i+1]
      for (Z i=1;i<=m;++i){
         Spc pi=s;
         dis2[i]=dist2FromSegment1(pi,x_[i],e_[i],l_[i]);
         vsProx[i]=pi;
      }
      Z im=dis2.indInf();
      sRes=vsProx[im];
```

```
      eRes=e_[im];
      lRes=y_[im]+(sRes-x_[im]).absVal();
   }
   CPM_MZ
   return X3<Spc,Vec,R>(sRes,eRes,lRes);
}

X3<Spc,Vec,Z> Path::fromArc(R p)const
{
   Z mL=3;
   Word loc("Path::fromArc(R)");
   CPM_MA
   cpmassert(n_>=2,loc);

   Spc sp;
   Vec e;
   Z i0;

   if (p>=length_){
      i0=n_;
      e=lastDir();
      e.normalize();
      sp=x_[n_]+e*(p-length_);
   }
   else if (p<=0){
      i0=0;
      e=firstDir();
      e.normalize();
      sp=x_[1]+e*p; // notice p<0
   }
   else{
      i0=y_.locate(p);
      R pExc=p-y_[i0];
      sp=x_[i0]+e_[i0]*pExc;
      e=e_[i0];
   }
   CPM_MZ
   return X3<Spc,Vec,Z>(sp,e,i0);
}

R Path::turn()const
{
   Z mL=3;
   Word loc("Path::turn()");
   CPM_MA
   R sum=0;
   Z i1,i2;
   for (i1=1,i2=2; i2<n_; ++i1,++i2){
      sum+=e_[i1].angTo(e_[i2]);
   }
```

```
   CPM_MZ
   return sum;
}

Group Path::con()const
{
   Z mL=3;
   Word loc("Path::con()");
   CPM_MA
   Group g;
   Z i1,i2;
   for (i1=1,i2=2; i2<n_; ++i1,++i2){
      Group g1=Group(x_[i2]-x_[i1]);
      Group g2=Group(x_[i2], e_[i1].rotTo(e_[i2]));
      g*=g1;
      g*=g2;
   }
   g*=Group(x_[n_]-x_[n_-1]);
   CPM_MZ
   return g;
}

R Path::turn2(R t1, R t2)const
{
   Z mL=3;
   Word loc("Path::turn2(R,R)");
   CPM_MA
   R u1,u2,sigma;
   if (t1<=t2){
      u1=t1;
      u2=t2;
      sigma=1;
   }
   else{
      u1=t2;
      u2=t1;
      sigma=-1;
   }
   X2<Path,Path>  pp=divideByArc(u1);
   R l1=pp.c1().length();
   Path p1=pp.c2();
   Path p2=p1.divideByArc(u2-l1).c1();
   R res=sigma*p2.turn();
   CPM_MZ
   return res;
}

Group Path::con2(R t1, R t2)const
{
   Z mL=3;
```

```
   Word loc("Path::con2(R,R)");
   CPM_MA
   R u1,u2;
   Z sigma;
   if (t1<=t2){
      u1=t1;
      u2=t2;
      sigma=1;
   }
   else{
      u1=t2;
      u2=t1;
      sigma=-1;
   }
   X2<Path,Path>  pp=divideByArc(u1);
   R l1=pp.c1().length();
   Path p1=pp.c2();
   Path p2=p1.divideByArc(u2-l1).c1();
   Group res=p2.con();
   if (sigma==-1) res=!res;
   CPM_MZ
   return res;
}

void Path::test(Z k)
{
   Z mL=1;
   Word loc("Path::test(Z)");
   CPM_MA
   k; // not used
   Z nd=2001;
   Spc s(0.789,-1000.3);
   R ang=45.4*CpmGeo::degree;
      // only positive
   R r=1.78;
   R dis=-1.679999;
   R rd=r+dis;
   Vec shift(rd*cos(ang),rd*sin(ang));
   Spc st=s+shift;

   Path circ=Path::circle(s,r,nd);
   X3<Spc,Vec,R> xsr=circ.project(st);
   Spc xRes=xsr.c1();
   R lenRes=xsr.c3();
   R rRes=(xRes-s).absVal();
   R rError=r-rRes;
   R lengthError=r*ang-lenRes;
   cpmcerr<<endl<<"rError="<<rError;
   cpmcerr<<endl<<"lengthError="<<lengthError;
```

```
   X3<Spc,Vec,Z> xsv=circ.fromArc(lenRes);
   Spc sp1=xsv.c1();
   Spc sp2=xRes;
   R locError=(sp1-sp2).absVal();
   cpmcerr<<endl<<"locError="<<locError;
   Z i,iS=10370;
   R n=2.387654;
   R phiTot=2*cpmpi*n;
   R phiStep=phiTot/(iS-1);
   R rhoStep=1./(iS-1);
   V<Spc> pq(iS);
   R phi=0;
   R rho=1;
   for (i=1;i<=iS;++i){

      Spc sp(rho*cos(phi),rho*sin(phi));
      phi+=phiStep;
      rho+=rhoStep;
      pq[i]=sp;

   }

   Path pRound=circ;
   R turns=pRound.turn();
   cpmdebug(turns);
   cpmdebug(phiTot);

   Group g=pRound.con();
   Vec vFirst=pRound.firstDir();
   Vec vLast=pRound.lastDir();
   Vec vTrans=vFirst;
   vTrans*=g;
   cpmdebug(vFirst);
   cpmdebug(vLast);
   cpmdebug(vTrans);
   cpmdebug(g);

   CPM_MZ
}

//////////////////// Skeleton ////////////////////////////////////////

void Skeleton::connectAsPath()
{
   Z mL=3;
   Word loc("Skeleton::connectAsPath()");
   CPM_MA
   for (Z i=1;i<x_.dim();i++) addEdge(i,i+1);
   CPM_MZ
}
```

```
Skeleton& Skeleton::operator *=(const Group& g)
{
   for (Z i=1;i<=x_.dim();i++) x_[i]*=g;
   return *this;
}

Skeleton& Skeleton::operator +=(const Vec& a)
{
   for (Z i=1;i<=x_.dim();i++) x_[i]+=a;
   return *this;
}

Skeleton& Skeleton::operator *=(const AxVec& A)
{
   for (Z i=1;i<=x_.dim();i++) x_[i]*=Group(A);
   return *this;
}

Skeleton& Skeleton::scl_(Spc const& fix, R fac)
{
   for (Z i=1;i<=x_.dim();i++) x_[i].scl_(fix,fac);
   return *this;
}

VecGraph Skeleton::flatten(Z connect)const
{
   Z nx=x_.dim();
   V<C> vr2(nx);
   for (Z i=1;i<=nx;i++) vr2[i]=C(x_[i][1],x_[i][2]);
   return VecGraph(vr2,connect);
}

Skeleton Skeleton::arc(const Spc& p1, const Spc& center,
   const AxVec& rot, Z n)
{
   pair<Angle,Vec> pav=rot.toAngAxs();
   Angle phi=pav.first;
   Vec axis=pav.second;
   V<Spc> ps=Path::arc(p1,center,rot,n)();
   Skeleton res(ps);
   for (Z i=1;i<n;i++){
      res.addEdge(i,i+1);
   }
   return res;
}

Skeleton Skeleton::line(const Spc& pi, const Spc& pf, Z n)
{
   static Word loc("Skeleton::line(Spc,Spc,Z)");
```

```
   cpmassert(n>1,loc);
   V<Spc> ps=Path::line(pi,pf,n)();
   Skeleton res(ps);
   res.connectAsPath();
   return res;
}

Skeleton Skeleton::parabola(const Spc& pi, const Spc& pc, const Spc& pf,
   Z n)
{
   static Word loc("Skeleton::parabola(Spc,Spc,Spc,Z)");
   cpmassert(n>2,loc);
   V<Spc> ps=Path::parabola(pi,pc,pf,n)();
   Skeleton res(ps);
   res.connectAsPath();
   return res;
}

Skeleton Skeleton::circle(const Spc& pc, R r, Z n, bool connect)
{
   if (r==0.){
      cpmerror("Skeleton::circle(...) vanishing radius");
   }
   Z i;
   V<Spc> ps=Path::circle(pc,r,n)();
   Skeleton res(ps);
   if (connect){
      for (i=1;i<n;i++){
         res.addEdge(i,i+1);
      }
      res.addEdge(n,1); // close the polygon
   }
   return res;
}

Skeleton Skeleton::ellipse(const Spc& pc, R a1, R a2, Z n, bool connect)
{
   if (a1==0.|| a2==0.){
      cpmerror("Skeleton::ellipse(...) vanishing half-axis");
   }
   Z i;
   V<Spc> ps=Path::ellipse(pc,a1,a2,n)();
   Skeleton res(ps);
   if (connect){
      for (i=1;i<n;i++){
         res.addEdge(i,i+1);
      }
      res.addEdge(n,1); // close the polygon
   }
   return res;
```

```
}

Skeleton Skeleton::bezier(V<Spc> const& vs, Z np)
// see R_Func::Bezier for the pattern followed here
// Notice the seamless cooperation of V, R_Vector, and R_Func
// which allows dimVec treated like a variable.
{
   Z mL=3;
   Word loc("Skeleton::bezier(V<Spc>,Z)");
   CPM_MA
   Z i,j,vsd=vs.dim();
   V<R_Vector> vvr(dimVec, R_Vector(vsd));
   for (i=1;i<=dimVec;++i) for (j=1;j<=vsd;++j) vvr[i][j]=vs[j][i];
   V<R_Func> vrf(dimVec);
   R2 iv(0,1);
   for (i=1;i<=dimVec;++i) vrf[i]=R_Func::Bernstein(vvr[i],iv);
   V< Spc > ps(np);
   R du=1./(np-1);
   R u=0;
   for (i=1;i<=np;++i){
      for (j=1;j<=dimVec;++j){
         ps[i][j]=(vrf[j])(u);
      }
      u+=du;
   }
   Skeleton res(ps);
   for (i=1;i<np;i++) res.addEdge(i,i+1);
   CPM_MZ
   return res;
}

Skeleton Skeleton::cone(R r, R h, Z n)
{
   Spc ori;
   Skeleton res=ellipse(ori,r,r,n,true);
   if (dimVec==2) return res;
   Spc sp(0,h,0);
   res.addVertex(sp);
   Z m=res.dim();
   for (Z i=1;i<=n;i++) res.addEdge(i,m);
   return res;
}

Skeleton Skeleton::cylinder(R r, R h, Z n)
{
   Spc ori;
   Skeleton res=circle(ori,r,n,true);
      // circle in the 1-2 plane
   if (dimVec==2) return res;
   Vec shift;
```

```
   shift[3]=h;
   Skeleton res2=res;
   res2+=shift;
   res&=(res2);
   for (Z i=1;i<=n;i++) res.addEdge(i,i+n);
   return res;
}

Skeleton Skeleton::dblPyr(R length, R d, Z n)
{
   R r=0.5*d;
// p is the polygon in the complex plane
// bringing the polygon to the 2,3 plane for 3D and the 2-axis
// for 2D
   V<C> p=circlePoints(r,n);
   V<Spc> sp(n);
   Z i;
   for (i=1;i<=n;i++){
      R py=p[i][2];
      R pz=p[i][1];
      sp[i]=Spc(0.,py,pz); // works also for 2D
   }
// adding the double pyramide end points on the 1-axis
   R h=length*0.5;
   Spc spL(-h,0,0);
   Spc spU(h,0,0);
   V<Spc> spExt=sp&spL&spU;
// making the Skeleton
   Skeleton res(spExt);
   for (i=1;i<n;i++)  res.addEdge(i,i+1);
   res.addEdge(n,1);
      // the previous two lines are not essential
   for (i=1;i<=n;i++){
      res.addEdge(n+1,i);
      res.addEdge(i,n+2);
   }
   return res;
}

Skeleton Skeleton::con(R w)const
{
   using CpmArrays::S;
   S<Z2> edg=edges();
   Z i,j,n=edg.dim();
   if (n==0) return Skeleton();
   V<Spc> xRes(n);
   for (i=1;i<=n;i++){
      Z2 zi=edg[i];
      Spc p1=x_[zi[1]];
      Spc p2=x_[zi[2]];
```

```
      xRes[i]=p1+(p2-p1)*w;
   }
   Skeleton res(xRes);
   for (i=1;i<=n;i++){
      Z2 zi=edg[i];
      for (j=1;j<=n;j++){
         Z2 zj=edg[j];
         if (zi[2]==zj[1]) res.neg_[i].add(j);
      }
   }
   return res;
}

Skeleton Skeleton::rec(R a, R b)
{
   Spc t1(0,0);
   Spc t2(a,0);
   Spc t3(a,b);
   Spc t4(0,b);
   V<Spc> vtb("",t1,t2,t3,t4);
   Skeleton res(vtb,true);
   res.clsPath_();
   return res;
}

//////////////////// class WireFrame ////////////////////////////////

WireFrame WireFrame::con(R w)const
{
   Z n=rep.dim();
   V<Skeleton> vc(n);
   for (Z i=1;i<=n;i++) vc[i]=rep[i].con(w);
   return WireFrame(vc);
}

WireFrame& WireFrame::operator *=(Group const& g)
{
   Z n=rep.dim();
   for (Z i=1;i<=n;i++) rep[i]*=g;
   return *this;
}

WireFrame& WireFrame::operator +=(Vec const& a)
{
   Z n=rep.dim();
   for (Z i=1;i<=n;i++) rep[i]+=a;
   return *this;
}

WireFrame& WireFrame::operator *=(AxVec const& A)
```

```
{
   Z n=rep.dim();
   for (Z i=1;i<=n;i++) rep[i]*=A;
   return *this;
}

WireFrame& WireFrame::operator *=(const V<Group>& g)
{
   Z n=cpminf<Z>(rep.dim(),g.dim());
   for (Z i=1;i<=n;i++) rep[i]*=g[i];
   return *this;
}

WireFrame& WireFrame::operator +=(const V<Vec>& a)
{
   Z n=cpminf<Z>(rep.dim(),a.dim());
   for (Z i=1;i<=n;i++) rep[i]+=a[i];
   return *this;
}

WireFrame& WireFrame::operator *=(const V<AxVec>& A)
{
   Z n=cpminf<Z>(rep.dim(),A.dim());
   for (Z i=1;i<=n;i++) rep[i]*=A[i];
   return *this;
}

WireFrame WireFrame::tripod(R length, R relWidth, R relGap, Z n)
{
   R fill=(1.-relGap);
   R L1=length, L2=L1*0.5;
   R l1=L1, l2=L2*fill;

   R d=l1*relWidth;
   Z ns=(dimVec==3 ? 6:3);
   V<Skeleton> res(ns);

// 1-axis made out of one Skeleton
   res[1]=Skeleton::dblPyr(l1,d,n);
   res[1]+=Vec(L2,0,0);

// 2-axis made of two Skeletons
   res[2]=Skeleton::dblPyr(l2,d,n);
   res[2]*=AxVec(Angle(90.));
   res[2]+=Vec(0,l2*0.5,0);
   res[3]=res[2];
   res[3]+=Vec(0,L2,0);
      // works for 2D and 3D

   if (dimVec==3){
```

```
// 3-axis made of three Skeletons
      R L3=L1*0.333333333;
      R l3=L3*fill;
      res[4]=Skeleton::dblPyr(l3,d,n);
      AxVec rot2(Angle(-90),2);
      res[4]*=rot2;
      Vec sh1(0,0,l3*0.5);
      res[4]+=sh1;
      res[5]=res[4];
      Vec sh2(0,0,L3);
      res[5]+=sh2;
      res[6]=res[5];
      res[6]+=sh2;
   }
   return WireFrame(res);
}

WireFrame WireFrame::globe(R r, R f, Z nLong, Z nLat, Z nCirc, bool axes)
{
   const R refSysEnlarge=1.2;
   R f_=1.-f;
   Spc ori;
   Skeleton sk1=Skeleton::ellipse(ori,r,r*f_,nCirc);
   if (dimVec==2) return WireFrame(V<Skeleton>(1,sk1));
// now we are in 3D, where we are interested in exclusively
   sk1*=AxVec(Angle(90),1);// now in 1-3-plane
   Angle dLong(180./nLong);
   V<Skeleton> res(nLong);
   res[1]=sk1;
   Z i;
   for (i=2;i<=nLong;i++){
      res[i]=res[i-1];
      res[i]*=AxVec(dLong,3);
   }
   Skeleton eq=Skeleton::ellipse(ori,r,r,nCirc);
   res&=eq;
   Angle dLat(90./nLat);
      // nLat is the number of latitude circles
      // equator included but pole not counting as a
      // latitude circle.
   Angle beta=dLat;
   for (i=1; i<nLat; i++){
      R cb=cos(beta);
      R rLat=r*cb;
      R hLat=r*sin(beta)*f_;
      Skeleton circLat=Skeleton::ellipse(ori,rLat,rLat,nCirc);
      Skeleton skNorth=circLat;
      skNorth+=Vec(0,0,hLat);
      res&=skNorth;
      Skeleton skSouth=circLat;
```

```
      skSouth+=Vec(0,0,-hLat);
      res&=skSouth;
      beta+=dLat;
   }
   if (axes)
      return WireFrame(res)&=tripod(r*refSysEnlarge);
   else
      return WireFrame(res);
}

WireFrame WireFrame::spline(const V<Spc>& vs, const Spc& pc1, Z n)
{
   static Word loc("WireFrame::spline(V<Spc>,Spc,Z)");
   Z i,nr=vs.dim()-1;
   cpmassert(nr>0,loc);
   V<Skeleton> res(nr); // p-1 connecting parabolic arcs between
      // p points
   res[1]=Skeleton::parabola(vs[1],pc1,vs[2],n);
   if (nr>1){
      V<Spc> pc(nr); // list of all control points
         // logic is easier for doing them in one pass and
         // storing them
      pc[1]=pc1;
      for (i=2;i<=nr;i++) pc[i]=vs[i]+(vs[i]-pc[i-1]);
         // making a list of control points
      for (i=2;i<=nr;i++)
         res[i]=Skeleton::parabola(vs[i],pc[i],vs[i+1],n);
            // last i+1 is vs.dim() as it should be
   }
   return WireFrame(res);
}
```

53 cpmdiscretespace.h

```
//? cpmdiscretespace.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_DISSPC_H_
#define CPM_DISSPC_H_
/*
    Description: discrete space for defining fields
      that can be summed together in arbitrary number.
      Note that adding my standard fields accumulates data
      that may create stack overflow for too many terms
      (~3000 in one case)
*/
#include <cpmdigraph.h>
#include <cpmf.h>

namespace CpmGeo{

   using CpmRootX::Z1;
   using CpmArrays::V;
   using CpmArrays::Vr;
   using CpmFunctions::F;
   using CpmFunctions::F1;

   template<class X>
class DisSpc: public DiGraph<X>{
      // directed graph with position descriptors attached to vertices
      // Has nothing to do with CpmGraphics::Graph

      typedef DisSpc Type;
      typedef DiGraph<X> Base;
   public:
      virtual DiGraph<X>* clone()const{ return new DisSpc(*this);}
//      CPM_IO_V
         // IO should be virtual in order to allow
         // additional attributes in derived classes
// all functions test the validity of their arguments and
// stop via cpmerror for unvalid ones.

explicit DisSpc(const V<X>& vert=V<X>()):
      DiGraph<X>(vert){}
         // a graph with n vertices and no edges
   };

// discrete fields
// arithmetic is defined on a common grid.
// What is the right place for the grid?
// Possibilities:
```

```
// 1. template argument of Type DisSpc
// 2. derived from DisSpc
// 3. static member of type DisSpc ; this is chosen here!

template <class X, class Y>
class DisField{ // discrete field
   typedef DisField Type;
   typedef Y ScalarType;
   static DisSpc<X> grid;
   Vr<Y> y;

public:
   CPM_IO_V
   static void setGrid(const V<X> vx){ grid=DisSpc<X>(vx);}
   static Z card(){ return grid.card();}
   void operator+=(const DisField& f){ y+=f.y;}
   DisField(){y=V<Y>(grid.card(),Y());}
   explicit DisField(const F<X,Y>& f);
   F<X,Y> contField()const;
   void setY(const V<Y>& yIn){ y=yIn;}
};

template <class X, class Y>
DisSpc<X> DisField<X,Y>::grid;


template<class X,class Y>
DisField<X,Y>::DisField(const F<X,Y>& f)
{
   y=CpmFunctions::apply<X,Y>(f,grid.get_x());
}

template<class X,class Y>
F<X,Y> DisField<X,Y>::contField()const
{
   Z n=grid.card();
   return F<X,Y>(new CpmAlgorithms::
      AveragingInterpolator<X,Y>(grid.get_x(),y));
}

template<class X,class Y>
bool DisField<X,Y>::prnOn(ostream& str)const
{
   cpmp(y);
   return true;
}

template<class X,class Y>
bool DisField<X,Y>::scanFrom(istream& str)
{
```

```
   cpms(y);
   return true;
}

} // namespace

#endif
```

54 cpmdistribution.h

```
//? cpmdistribution.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_DISTRIBUTION_H_
#define CPM_DISTRIBUTION_H_

/*
   Description: This class Distribution describes data that
      can be binned together and form a histogram. These are the
      discrete version of probability measures
*/

#include <cpminterfaces.h>
#include <cpmtypes.h>
#include <cpminterval.h>
#include <cpmv.h>
#include <cpmf.h>
#include <cpmlinalg.h>
#include <cpmgraph.h>
#include <vector>

namespace CpmArrays{

   using CpmRoot::R;

template <class X>
class Distribution{
   // general framework for distributions ( = histograms)
   // My present workhorse is DisR which is derived from Distribution<R>

protected: // efficient formation of derived classes is important here

   CpmRootX::Z1 n;
      // number of bins

   V<Z> rep;
      // rep[i] is the number of X-instances which belong to the bin
      // indexed i (1<=i<=n)

   CpmFunctions::F<X,Z> f;
      // classifying function such that f(x) is the index of the bin
      // to which x belongs. Values 0 and n+1 indicate 'out of range'.
      // the second argument of function put decides whether
      // out-of-range x's influence rep or not.

public:
   Distribution(){}
```

```
      // notice that n, rep, and f get initialized anyway,
      // since there is a default constructor for their types
   Distribution(Z n_):
      n(n_),rep(n,0){}

   Distribution(Z n_, const CpmFunctions::F<X,Z>& f_):
      n(n_),rep(n,0),f(f_){}

   void put(const X& x, bool all=true);
      // putting x to the distribution. If all==true all x 'get
      // registered' even those out of range. If all==false
      // only those x for which f(x)>=1 && f(x)<=n count.

   void put(const std::vector<X>& v, bool all=true)
      { for (Z i=0;i<(Z)v.size();i++) put(v[i],all);}
      // adding the components of v to the distribution
      // we use std::vector here since many sources of vectors
      // are such that the length is not known in advance. Here
      // std::vector is efficient because of it's efficient
      // push_back function

   void put(const V<X>& v, bool all=true)
      { for (Z i=1;i<=v.dim();i++) put(v[i],all);}
      // adding the components of v to the distribution

   virtual void show(CpmGraphics::Graph& g, bool autoX=false)const;
      // graphical representation of i |--> rep[i].card().
      // g: Screen area in whitch the histogram is shown
      // argument autoX has no effect here (will become
      // meaningful in DisR)

   Z sum()const{ Z s=0; for(Z i=1;i<=n;i++) s+=rep[i]; return s;}
      // sum of the bins

   Z singular()const;
      // Let k := card{ i | i \in {1,...n} and rep[i]>0}
      // We return k if k<2 and -1 else
      // Thus the return value 1 says that the distribution is 'singular'
      // in the sense of being concentrated to a single point.
      // This is an exceptional situation in cases where one wants to
      // extract a parameter from fitting the distribution to a
      // certain family of curves (see function DisR::tBoltzman).
      // The computation is very fast for normal distributions
      // which have not nearly all bins zero.
};

template <class X>
void Distribution<X>::show(CpmGraphics::Graph& g, bool autoX)const
{
   autoX; // not used
```

```
   R_Vector x(n);
   for (Z i=1;i<=n;i++) x[i]=rep[i];
   g.show(x);
}

template <class X>
void Distribution<X>::put(const X& x, bool all)
{
   Z i=f(x);
   if (all){
      if (i<=1) rep[1]++;
      else if (i>=n) rep[n]++;
      else rep[i]++;
   }
   else{
      if (i>=1 && i<=n) rep[i]++;
   }
}

template <class X>
Z Distribution<X>::singular()const
{
   Z count=0;
   for (Z i=1;i<=n;i++){
      Z val=rep[i];
      if (val>0) count++; // val>=0 anyway
      if (count>1) return -1; // then there are two points
   }
   // if we come here there, count==0 or count==1
   return count;
}

//////////////////////// class DisR ////////////////////////////////////

class DisR: public Distribution<R>{ // distribution of reals
   CpmGeo::Iv range;
      // The interval, the subdivision of which defines
      // binning of R values
   void adjust();
      // sets the dependent data according to
      // the independent ones. In our case range and n are
      // independent and f is dependent
public:
   typedef DisR Type;
   typedef Distribution<R> Base;

   static Z verbose;
      // enables visualization of the success of
      // tBoltzmann (presently for verbose>0)
```

```
   DisR():Base(){}

   DisR(CpmGeo::Iv iv, Z n_):Base(n_),range(iv){adjust();}

   DisR(R rMin, R rMax, Z n_):Base(n_),range(rMin,rMax){adjust();}

explicit DisR(const std::vector<R>& v, Z n=16, R fac1=1, R fac2=1,
      bool all=true);
      // here range is determined by the smallest and the largest
      // component of v, vMin, vMax as the interval
      // [fac1*vMin, fac2*vMax].

explicit DisR(const std::vector<Z>& v, Z n=16, R fac1=1, R fac2=1,
      bool all=true);
      // here range is determined by the smallest and the largest
      // component of v, vMin, vMax as the interval
      // [fac1*vMin, fac2*vMax].

explicit DisR(const V<R>& v, Z n=16, R fac1=1, R fac2=1,
      bool all=true);
      // here range is determined by the smallest and the largest
      // component of v, vMin, vMax as the interval
      // [fac1*vMin, fac2*vMax].

   CPM_IO
   virtual DisR* clone()const{ return new DisR(*this);}

   void show(CpmGraphics::Graph& g, bool autoX=false)const;
      // representation in a bar diagram, where the range of the axes is
      // given by g.getIvX() and g.getIvY().

   CpmLinAlg::R2 tBoltzmann(R tShow=0, Word title="")const;
      // Extracting a 'temperature' out of the distribution.
      // Let res be the return value. Then res.x1 is this
      // 'temperature' and res.x2 is a misfit indicator.
      // It is >=0 where 0 indicates perfect fit. Values
      // arround 1 say essentially that the fit is
      // of no value, values >1 indicate that the distribution
      // has no similarity with one that would arrise from
      // a Boltzmann distribution belonging to some (even fuzzy)
      // value of a temperature.
      // More precisely: We try to find c and xm such that
      // sum over i of (res[i]-c*exp(-x[i]/xm))^2, x:=range.centers(n)
      // is as small as possible.
      // For each xm, we determine the Gauss least square solution
      // c(xm) for c, and then we minimize
      // error(xm):=sum over i of (res[i]-c(xm)*exp(-x[i]/xm))^2
      // to get the value res.x1=xm.
      // res.x2 is sqrt(error(xm)/(sum over i of (res[i])^2))
      // which is a dimension-less misfit measure.
```

```
};

// Since the introduction of namespaces in C++ it is much more
// adequate than before to define functions which do not belong to
// a class. In the following function, the concept of a distribution
// is only an auxiliar one and so it looks not natural to make it
// a member function (static or not) of DisR.

CpmLinAlg::R2 disBoltzmann(std::vector<R>& v, R frac=0.9, Z nBins=32);
   // similar to DisR::tBoltzmann() but the distribution is formed out
   // of v such that a fraction (1-frac) of the largest v-values is
   // omitted (to achieve this v gets ordered internally with
   // using std::sort, so v gets changed by call, so it is not of
   // type const &) in forming the distribution which finally gets
   // analyzed.

} // namespace

#endif
```

55 cpmdistribution.cpp

```
//? cpmdistribution.cpp
//? Status of work 2008-10-25.
//?
#include <cpmdistribution.h>
#include <cpmgraph.h>
#include <cpmalgorithms.h>
#include <cpmmini.h>

#include <algorithm>
#include <functional>

using CpmArrays::DisR;
using CpmArrays::R_Vector;
using CpmArrays::R_Matrix;
using CpmArrays::V;
using CpmRoot::R;
using CpmRoot::Z;
using CpmRoot::Word;
using CpmLinAlg::R2;
using CpmGeo::Iv;
using CpmFunctions::F;
using CpmFunctions::F2;
using CpmGraphics::Color;
using CpmGraphics::RED;
using CpmGraphics::Graph;
using CpmGraphics::rainBow;

///////////////////////// class DisR ////////////////////////////////////

Z DisR::verbose=0;

void DisR::adjust()
{
   f=CpmAlgorithms::Quantizer(range,n);
}

DisR::DisR(const std::vector<R>& v, Z n, R fac1, R fac2, bool all):
   Base(n)
{
   Z i,vs=(Z)v.size();
   if (vs<1)
      cpmerror("DisR::DisR(const std::vector<R>& v, Z n_): v.size()==0");
   R vL=v[0];
   R vU=vL;
   for (i=0;i<vs;i++){
      R val=v[i];
      if (val>vU) vU=val;
```

```
      if (val<vL) vL=val;
   }
   range=Iv(vL*fac1,vU*fac2);
   adjust();
   put(v,all);
}

DisR::DisR(const std::vector<Z>& v, Z n, R fac1, R fac2, bool all):
   Base(n)
{
   Z i,vs=(Z)v.size();
   if (vs<1)
      cpmerror("DisR::DisR(const std::vector<Z>& v, Z n_): v.size()==0");
   vector<R> vr(vs);
   for (i=0;i<vs;i++) vr[i]=v[i];
   R vL=vr[0];
   R vU=vL;
   for (i=0;i<vs;i++){
      R val=vr[i];
      if (val>vU) vU=val;
      if (val<vL) vL=val;
   }
   range=Iv(vL*fac1,vU*fac2);
   adjust();
   put(vr,all);
}

DisR::DisR(const V<R>& v, Z n, R fac1, R fac2, bool all):Base(n)
{
   Z i,vs=v.dim();
   if (vs<1)
      cpmerror("DisR::DisR(const V<R>& v, Z n_): v.dim()==0");
   vector<R> vr(vs);
   for (i=0;i<vs;i++) vr[i]=v[i+1];
   R vL=vr[0];
   R vU=vL;
   for (i=0;i<vs;i++){
      R val=vr[i];
      if (val>vU) vU=val;
      if (val<vL) vL=val;
   }
   range=Iv(vL*fac1,vU*fac2);
   adjust();
   put(vr,all);
}

bool DisR::prnOn(std::ostream& str)const
{
   cpmp(n);
   cpmp(rep);
```

```
   cpmp(range);
   return true;
}

bool DisR::scanFrom(std::istream& str)
{
   cpms(n);
   cpms(rep);
   cpms(range);
   adjust();
   return true;
}

void DisR::show(CpmGraphics::Graph& g, bool autoX)const
{
   Z mL=2;
   Word loc("DisR::show(CpmGraphics::Graph& g, bool autoX)");
   CPM_MA
   if (autoX){
      g.setX(range);
      cpmcerr<<endl<<"range="<<range<<endl;
   }
   R_Vector x(g.getIvX().centers(n));
   R_Vector y(n);
   for (Z i=1;i<=n;i++) y[i]=rep[i];
   g.show(x,y,Color(RED),1);
   CPM_MZ
}

namespace{
   R func(const R_Vector& a, const V<R>& x, const V<Z>& y)
      // misfit indicator with suitably normalized return value
   {
      R alpha=a[1];
      R s1=0;
      R s2=0;
      Z n=x.dim(),i;
      for (i=1;i<=n;i++){
         R val=-alpha*x[i];
         s1+=exp(2*val);
         s2+=exp(val)*y[i];
      }
      R c=s2/s1;
      R res1=0;
      R res2=0;
      for (i=1;i<=n;i++){
         R ri1=y[i];;
         R ri2=ri1-c*exp(-alpha*x[i]);
         res1+=ri1*ri1;
         res2+=ri2*ri2;
```

```
      }
      return sqrt(res2/res1);
   }
}

R2 DisR::tBoltzmann(R tShow, Word title)const
{
   const R tol=1e-3; // was 1e-3
      // relative tolerance for localizing a minimum
   const R biasWeight=3; // was 3
      // the larger this value, the closer to 0 one starts with
      // looking for the proper 'thermal energy'

   if (singular()>=0){  // then the
      // distribution is either void or concentrated on a
      // single point. The fit is trivial but perfect.
      // Normally no noticible computational burden.
      return R2(0,0);
   }

   R_Vector x(range.centers(n));
   F<R_Vector,R> f=F2<R_Vector,V<R>,V<Z>,R>(x,rep)(func);
   CpmAlgorithms::Mini mini(f,tol);
 //  mini.verbose=2;
   R_Vector start(1);
   start[1]=biasWeight/range.mean();
   R_Vector res=mini.getPoint(start);
   R tInv=res[1];
   R fit=mini.getValue();
   bool succ=mini.getSuc();
   if (!succ) fit=1;
   if (tShow>0){
      R_Matrix view(2);
      view[1]=R_Vector(n);
      view[2]=R_Vector(n);
      Z i;
      for (i=1;i<=n;i++)view[1][i]=rep[i];
      R s1=0;
      R s2=0;
      for (i=1;i<=n;i++){
         R val=-tInv*x[i];
         s1+=exp(2*val);
         s2+=exp(val)*rep[i];
      }
      R c=s2/s1;
      for (i=1;i<=n;i++){
         view[2][i]=c*exp(-tInv*x[i]);
      }
      Graph g;
      g.show(x,view,rainBow,title,0.2);
```

```
      cpmwait(tShow);
   }
   return R2(1./tInv,fit);
}

R2 CpmArrays::disBoltzmann(std::vector<R>& v, R frac, Z nBins)
{
   const R largeMisFit=2;
   const R uselessValue=0;
   using namespace std;
   Z n=Z(frac*v.size());
   if (n<nBins) return R2(uselessValue,largeMisFit);
      // no useful result possible
   sort(v.begin(),v.end(),less<R>());
   vector<R> vCut(n);
   copy(v.begin(),v.begin()+n,vCut.begin());
   DisR dist(vCut,nBins);
   return dist.tBoltzmann();
}
```

56 cpmdynaux.h

```
//? cpmdynaux.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_DYNAUX_H_
#define CPM_DYNAUX_H_
/*
   Description:auxiliar classes for dynamical systems
*/
#include <cpmtypes.h>
#include <cpmrecord.h>
#include <cpminterval.h>
#include <cpmdimdef.h>

namespace CpmDynSys{

   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmRoot::Word;
   using CpmRootX::R1;
   using CpmRootX::Z1;
   using CpmRootX::B;
   using CpmRootX::hugeNumber;
   using CpmRootX::Record;

   using CpmGeo::Iv;


struct StepControl{ // a data collection providing I/O and initialization
    // and an important method done()

   R1 t;
      // time to which the present state *this refers. I hesitated for
      // some time to hide this important physical quantity among the
      // more technical auxiliaries that follow. However the information
      // flow strongly suggests this placement of t

   R1 tLast;
      // time to which the present state *this refered at the end of
      // the last step.

   R1 dtLast;
      // t-tLast

   R1 tLimit;
      // in function stepAsy(R&) (which implements asynchronous time step
      // adjustment) no final time larger than this value will be
      // proposed or reached by any intermediate time step.
```

```
   B relaxFlag;
      // if relaxFlag is true, function step() implements a 'motion
      // neglecting inertia' by setting all particle velocities to zero
      // at the end of step(). Setting this parameter to 'true' also sets
      // all particle velocities to zero. Thus an equilibrium will
      // effectively approached without 'overshooting'. If, for instance,
      // two particles overlap due to either a violent collision or a
      // careless setting of initial positions, on can achieve a
      // non-overlapping state by doing a few steps for relaxFlag==true.
      // Other than with normal dynamics, one will not get an 'exploding
      // state' but one with all particles at rest. State changes for
      // relaxFlag==true are considered now (since 2000-2-3) as processes
      // in time, i.e. the function done() does the updating of t,
      // dtMean, dtLast just as if relaxFlag were false. Also this is not
      // always the most natural choice from physics it is very economic
      // since so the function run can used to organize and visualize also
      // state evolution under 'relaxing dynamics'.

   R1 dtMean;
      // a floating mean value of the dt-values used over the near past.
      // The precise meaning results from the code in function done().

   R1 disLast;
      // The quantity which measures the difference between a state
      // obtained by a single step and the succession of two half steps
      // that occurred in obtaining the present step is an important
      // quantity in analysing the performance of the timestep adjustment
      // algorithm.

   R1 disMean;
      //  a floating mean value of the dis-values used over the near past.

   void update();

   typedef StepControl Type;
   CPM_IO
   StepControl():tLimit(hugeNumber){}
      // meaningful initialization helps to keep the constructors of
      // ParDynSys<> simple
   void done(R dt_, R dis_);
      // increasing the time by dt_ achieved
   void doneTill(R t_, R dis_);
      // increasing the time to value t_ achieved
};

void adjust_dt(R&,Z);

} // namespace
```

```
#endif
```

57 cpmdynaux.cpp

```
//? cpmdynaux.cpp
//? Status of work 2008-10-25.
//?
#include <cpmdynaux.h>

using namespace CpmRoot;

using CpmDynSys::StepControl;
using CpmRootX::Record;

void CpmDynSys::adjust_dt(R& dt, Z counter)
{
   const R incr1=2;
   const R incr2=sqrt(incr1);
   const R incr1Inv=1./incr1;
   const R incr2Inv=1./incr2;

   if (counter<=1) dt*=incr1;
   else if (counter==2) dt*=incr2;
   else if (counter<=4) ;
   else if (counter<=8) dt*=incr2Inv;
   else  dt*=incr1Inv;
}

//////////////////// struct StepControl ////////////////////

bool StepControl::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("StepControl",str);
   cpmp(t);
   cpmp(tLast);
   cpmp(dtLast);
   cpmp(tLimit);
   cpmp(relaxFlag);
   cpmp(dtMean);
   cpmp(disLast);
   cpmp(disMean);
   return true;
}

bool StepControl::scanFrom(istream& str)
{
   cpms(t);
   cpms(tLast);
   cpms(dtLast);
   cpms(tLimit);
   cpms(relaxFlag);
```

```
   cpms(dtMean);
   cpms(disLast);
   cpms(disMean);
   return true;
}

void StepControl::update()
{
   const R w=0.9; // the nearer to 1 this quantity is, the longer into
   // the past the dt-memory reaches
   dtMean=w*dtMean+(1-w)*dtLast; // floating mean value
   disMean=w*disMean+(1-w)*disLast;
}

void StepControl::done(R dt_, R dis_)
{
   tLast=t;
   t+=dt_;
   dtLast=dt_;
   disLast=dis_;
   update();
}

void StepControl::doneTill(R t_, R dis_)
{
   tLast=t;
   t=t_;
   dtLast=t-tLast;
   disLast=dis_;
   update();
}
```

58 cpmextractor.h

```
//? cpmextractor.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_EXTRACTOR_H_
#define CPM_EXTRACTOR_H_
/*

   Description: declares class Extractor
   for reading in a uniform manner states of
   particle arrays and external systems from
   PaLa 'movie'-files (PaDyAn 'movie'-files before )

*/
#include <cpmbody.h>
#include <cpmsys2dat.h>
#include <cpmpardynsys.h>
#include <cpmfile.h>

#if defined(CPM_DIMDEF2)

#define CPM_DIM CpmDim2
#define CPM_CAMERA CpmCamera2
#define CPM_RIGID_BODY CpmRigidBody2
#define CPM_PHYSICS CpmPhysics2
#define CPM_TOP CpmPala2_2

namespace CPM_TOP{
   #include <cpmextractorx.h>
}

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS
#undef CPM_RIGID_BODY
#undef CPM_TOP

#endif

//////////////////////////////////////////////////////////////////

#if defined(CPM_DIMDEF3)

#define CPM_DIM CpmDim3
#define CPM_CAMERA CpmCamera3
#define CPM_RIGID_BODY CpmRigidBody3
#define CPM_PHYSICS CpmPhysics3
#define CPM_TOP CpmPala2_3
```

```
namespace CPM_TOP{
   #include <cpmextractorx.h>
}

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS
#undef CPM_RIGID_BODY
#undef CPM_TOP

#endif

#endif // guard
```

59 cpmextractorx.h

```
//? cpmextractorx.h
//? Status of work 2008-10-25.
//?
/*
   see cpmextractor.h
*/
   using namespace CpmStd;

   using CpmRoot::Word;
   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmRoot::L;
   using CpmRootX::B;
   using CpmRootX::Record;

   using CpmArrays::VV;
   using CpmArrays::Vp;
   using CpmArrays::R_Vector;

   using CpmSystem::IFile;
   using CpmSystem::OFile;

   using CpmGraphics::ColRef;
   using CpmImaging::ShowControl;
   using CpmImaging::Color;
   using CpmImaging::WHITE;
   using CpmImaging::YELLOW;

   using CpmAlgorithms::CyclicAlarm;
   using CpmAlgorithms::NonCyclicAlarm;

   using CpmProperties::SysControl;

   using CPM_DIM::Spc;

   using CPM_CAMERA::Camera;

// needs to agree with code defining Sys2(RecordHandler)
   using CPM_PHYSICS::Lid;
   using CPM_PHYSICS::Pot;
   using CPM_PHYSICS::Fen;
   using CPM_PHYSICS::SphTank;
   using CPM_PHYSICS::HomFil;
   using CPM_PHYSICS::Drum;
  // using CPM_PHYSICS::Mill;
      // Exact types to which the external systems of Sys2 may belong.
      // These are class names as tokens; these tokens can't be
```

```
      // recovered from data of type Word (which can easily communicated
      // to the present class via reading from file). So there is a lack
      // (so far ?) in supporting polymorphism of classes that get
      // transpoted via files. This is only a question of elegance,
      // though.
//doc namespace CpmPala2_x{
//////////////////////////////// class ParExt ///////////////////////
template <class CC,class CV,class PT,class EX,class Sys>
class ParExt{ // particle extractor
   // for reading in a uniform manner states of
   // particle arrays and external systems from
   // PaLa 'movie'-files (PaDyAn 'movie'-files before )

   V<Word> exTypes;
      // list of types of external sytems
   Vp<EX> ex;
      // polymorphic array of external systems
   Partition parti;
   B partiPresent;
   V<CC> vc;
   V<CV> vv;
   V<PT> vp;
   Z1 np,ne,count;
   B valid;
   IFile file;
      // here it is not sufficient to hold the file name.
      // During the existence of a ParExt instance
      // a stream object has to be active for reading
      // from file.
   OFile outFile;
      // text file for non-graphical output
   void initialize(const Word& fileName,  const Word& outFileName);

public:
   CPM_NAM(ParExt<...>)
   ParExt(){}
      // trivial, all data initialized by default, especially valid=false
      // No file gets opened

   ParExt(Word const& fileName, Word const& outFileName=Word())
      { initialize(fileName,outFileName);}
      // creates such a state that after a call of next()
      // the first data-frame of file is loaded into *this
      // and can be accessed through the access functions to
      // be defined soon. If outFileName!=Word() an output
      // file named outFileName gets created so that it is open
      // for writing on it.

   bool next(bool tolerant=false);
      // loading the data with the values from the next frame
```

```
      // and returning whether this was done successfully
      // tolerant==true is advisable in applications where
      // reading partially completed files necessarily occurs and
      // is handled properly. The the warnings which would otherwise
      // occur in reading a incomplete last image  will be suppressed
      // if tolerant==true

// accessing the frame data read from file
   V<PT>& get_vp(){ return vp;}
   V<PT> const& get_vp()const{ return vp;}

   V<CC>& get_vc(){ return vc;}
   V<CC> const& get_vc()const{ return vc;}

   Vp<EX>& get_ex(){ return ex;}
   Vp<EX> const& get_ex()const{ return ex;}

   Partition& get_parti(){ return parti;}
   const Partition& get_parti()const{ return parti;}

   bool isVal()const{ return valid;}
   Z get_np()const{ return np;}
   Z get_ne()const{ return ne;}
   Z getCount()const{ return count;}
      // returns the number of the frame, the data of which
      // are presently accessible
   void show(Camera& ca, ShowControl const& sc);
      // representing the configuration as a screen image under
      // a variety of controls, which are bundled in struct
      // ShowControl.
};

template <class CC,class CV,class PT,class EX, class Sys>
void ParExt<CC,CV,PT,EX,Sys>::initialize
   (Word const& fileName, Word const& outFileName)
{
   const Z mL=1;
   Word loc( "ParExt<>:initialize(" );
   loc&=fileName;
   loc&=", ";
   loc&=outFileName;
   loc&=")";
   CPM_MA
   vc=V<CC>(0); // the default constructor creates
      // arrays of dimension 1 instead of the more economic
      // ones of dimension 0
   vv=V<CV>(0);
   //Word fn=Message::ext(fileName);
   file=IFile(fileName);
   //file=IFile(fn);
```

```
   if (outFileName.dim()>0) outFile=OFile(outFileName);
   bool succ;
// partition
   succ=parti.scanFrom(file());
      // We write the partition only once at the beginning and thus
      // abandon the elaborate mechanism which allowed to continue
      // long moviefiles without the need to read through them
      // for getting the partition from the beginning
   if (!succ){
      Word mes=loc&" not able to read a partition";
      cpmerror(mes);
   }
   else{
      Word mes=loc&" was able to read partition";
      cpmmessage(mes);
      cpmcerr<<"parti="<<endl<<parti<<endl;
   }
// reading exTypes
   succ=exTypes.scanFrom(file());
   if (!succ){
      Word mes=loc&" not able to read exTypes";
      cpmerror(mes);
   }
   else{
      Word mes=loc&" was able to read exTypes";
      cpmmessage(mes);
      cpmcerr<<"exTypes="<<endl<<exTypes<<endl;
   }
// reading vc
   succ=vc.scanFrom(file());
   if (!succ){
      Word mes=loc&" not able to read vc";
      cpmerror(mes);
   }
   else{
      Word mes=loc&" was able to read vc";
      cpmmessage(mes);
   }
   np=vc.dim();
   ostringstream ost;
   ost<<endl<<"vc read for vc.dim()=np="<<np<<endl;
   cpmmessage(ost);
   if (np==0) cpmwarning("!!! no particles read from file !!!");
// initializing vp
   vp=V<PT>(np);
// initializing ex with proper types of derived classes
// needs to agree with code defining Sys2(RecordHandler)
   ne=exTypes.dim();
   ex=Vp<EX>(ne);
   for (Z i=1;i<=ne;++i){
```

```
      Word wi=exTypes[i];
      if (wi=="Lid") ex[i]=Lid();
      else if (wi=="Pot") ex[i]=Pot();
      else if (wi=="Fen") ex[i]=Fen();
      else if (wi=="SphTank") ex[i]=SphTank();
      else if (wi=="HomFil") ex[i]=HomFil();
      else if (wi=="Drum") ex[i]=Drum();
      else cpmerror("unknown type of external system found");
   }
   valid=true;
   CPM_MZ
}

template <class CC,class CV,class PT,class EX, class Sys>
bool ParExt<CC,CV,PT,EX,Sys>::next(bool tolerant)
{
   const Z mL=2;
   Word loc(nameOf()&"::next()");
   CPM_MA
   cpmassert(valid==true,loc);
   if (!valid){
      ostringstream ost;
      ost<<loc<<" can’t do a step for unvalid state";
      cpmwarning(ost);
      cpmmessage(mL,loc&" done , !valid");
      return valid;
   }
   bool succ;
// reading vv
   succ=vv.scanFrom(file());
   if (!succ || vv.dim()!=np){
      valid=false;
      if (!tolerant){
         ostringstream ost;
         ost<<endl<<loc<<
         " reading vv failed, vv.dim()="<<vv.dim()<<" np="<<np;
         cpmwarning(ost);
      }
      cpmmessage(mL,loc&" done, !succ");
      return valid;
   }
// setting the particles defined by vc and vv
   cpmmessage(mL,loc&" number of particles is "&cpm(np));
   R npInv=(np==0 ? 0 : 1./np);
   Z countProblem=0;
   for (Z j=1;j<=np;j++){
      if (cpmverbose>mL) cpmprogress("read particles",(j-1)*npInv);
      vp[j]=PT(vc[j],vv[j]);
      bool val=vp[j].isVal();
      if (!val){
```

```
         Z id=vp[j].get_id();
         ostringstream ost;
         ost<<loc<<" particle with id="<<id<<" is not valid";
         cpmwarning(ost);
         countProblem++;
      }
      Spc origin;
      val=(vp[j].get_x()==origin);
      if (val){
         Z id=vp[j].get_id();
         ostringstream ost;
         ost<<loc<<" particle with id="<<id<<" is placed at origin";
         cpmwarning(ost);
         countProblem++;
      }
   }
   cpmmessage(mL,"np-loop done");
   if (countProblem>0){
      ostringstream ost;
      ost<<"There were "<<countProblem<<" dubious particles"  ;
      cpmwarning(ost);
   }
// reading ex
   Z i;
   cpmmessage(mL,"need to read "&cpm(ne)&" external systems from file");
   for (i=1;i<=ne;++i){
      cpmmessage(mL,"ex("&cpm(i)&") loop entry started");
      Word namei=ex(i).nameOf();
      cpmmessage(mL,"class type is "&namei);
      succ=ex(i).scanFrom(file());
      if (!succ){
         ostringstream ost;
         ost<<endl<<loc<<" reading ex["<<i<<"] failed";
         cpmwarning(ost);
         valid=false;
         cpmwarning(loc&" done, !succ");
         return valid;
      }
      cpmmessage(mL,"ex("&cpm(i)&") loop entry done");
   }
   count+=1;
   if (cpmverbose>=mL && valid){
      ostringstream ost;
      ost<<"Frame "<<count<<" read";
      cpmmessage(ost);
   }
   if (!valid && !tolerant){
      ostringstream ost;
      ost<<loc<<" unable to read frame "<<count;
      cpmwarning(ost);
```

```
   }
   cpmmessage(mL,loc&" done OK");
   return valid;
}

template <class CC,class CV,class PT,class EX, class Sys>
void ParExt<CC,CV,PT,EX,Sys>::show(Camera& ca, ShowControl const& sc)
// needs class Sys only if sc.modality=="normalView"
// The other modalities need ComPar::showProperty(...)
// (=PT::showProperty(...)).
{
   const Z fieldLength=4;
      // formatting in writing the current image number
   Z mL=3;
   static Word loc("ParExt<>::show(Camera,ShowControl)");
   CPM_MA
   static Color colTon(YELLOW);
   static Color colColorize(WHITE);
   ostringstream ost;
   ost<<endl<<"np="<<np<<" count="<<count<<endl;
   cpmmessage(mL,Word(ost));
   Z offSet=sc.offSet;
  // rc.get("runcontrol","offSet",offSet);
   if (count<offSet) return;

   Z i,j,npLoc=np;
      // don't change np since this spoils the execution of next()
      // the next time. Changing vp does not hurt, since next() builds
      // it from scratch.
   bool waitDone=false, succ=false, cclear=false;
   Word text="frame "&cpmwrite(count);
   CpmArrays::VV<ColRef> imgDat;
// an image should be formed if viewing period or imaging period is met
// and
// the number of offset images is passed or if the one wanted image is
// reached
   Z lastFrameNumber=sc.nocyalImg.last();
   bool scb1=sc.cyalImg();
   bool scb2=sc.nocyalImg();
   bool cimg=scb1||scb2;
   bool cview=sc.cyalView();
   if (cview) cclear=sc.cyalClear();
   bool final=(lastFrameNumber==count);

   parti.autoFrame1(ca,sc.autoFrameFactor);

   if ( cimg || cview || final ){ // only than the movie frame
      // needs to become processed
      bool caclear=cclear || cimg ;
      if (final) caclear=false;
```

```
// restricting the set of particles according to position
      V<PT> vpLoc=vp;
      if (sc.boxSel){
         V<bool> sel(np);
         for (i=1;i<=np;i++){
            Spc xi=vp[i].get_x();
            bool sx=sc.ivx.contains(xi[1]);
            bool sy=sc.ivy.contains(xi[2]);
            bool sz=Spc::dimension()==2 ? true : sc.ivz.contains(xi[3]);
            sel[i]=sx&&sy&&sz;
         }
         vpLoc=vp.select(sel);
         npLoc=vp.dim();
         cpmmessage(mL,"# of particles after boxSel="&cpm(npLoc));
      }
// main program control as a if control structure
      if (sc.modality=="HistogramTonerVelocity"){
         cpmmessage(mL,loc&": HistogramTonerVelocity selected");
         R_Vector parameters(1);
         parameters[1]=SysControl::vCrit;
         PT::showProperty(sc.modality,vpLoc,ca,parameters);
      }
      else if (sc.modality=="HistogramCarrierVelocity"){
         R_Vector parameters(1);
         parameters[1]=SysControl::vCrit;
         cpmmessage(mL,loc&": HistogramCarrierVelocity selected");
         PT::showProperty(sc.modality,vpLoc,ca,parameters);
      }
      else if (sc.modality=="Histogram_xPosition"){
         cpmmessage(mL,loc&": Histogram_xPosition selected");
         R_Vector parameters(1);
         R xMax=0.5*parti.get_xLength();
         parameters[1]=xMax;
         PT::showProperty(sc.modality,vpLoc,ca,parameters);
      }
      else if (sc.modality=="Histogram_yPosition"){
         cpmmessage(mL,loc&": Histogram_yPosition selected");
         R_Vector parameters(1);
         R yMax=0.5*parti.get_yLength();
         parameters[1]=yMax;
         PT::showProperty(sc.modality,vpLoc,ca,parameters);
      }
      else if (sc.modality=="Histogram_zPosition"){
         cpmmessage(mL,loc&": Histogram_zPosition selected");
         R_Vector parameters(1);
         R zMax=0.5*parti.get_zLength();
         parameters[1]=zMax;
         PT::showProperty(sc.modality,vpLoc,ca,parameters);
      }
```

```
else if (sc.modality=="normalView"){
   cpmmessage(mL,loc&": normalView selected");
   Sys sys(vpLoc,ex,parti);
   Z nLoc=vpLoc.dim();
   for (i=1;i<=nLoc;++i){
      PT pi=sys.getPar(i);
   }
   if (SysControl::showMeanFil) sys.prepareStep();
   Z lcd=sc.listColorize.dim();
   if (lcd>0){
      cpmmessage(mL,loc&": "&cpm(lcd)&" particles colorized");
      for (j=1;j<=lcd;j++){
         Z sclc=sc.listColorize[j];
         sys.setShowColor(sclc,colColorize);
      }
   }
   Z ipd=sc.invisibleParticles.dim();
   if (ipd>0){
      cpmmessage(mL,loc&": "&cpm(ipd)&" particles made invisible");
      Color civ(CpmImaging::TRANSPARENT_);
      for (j=1;j<=ipd;++j){
         Z scip=sc.invisibleParticles[j];
         sys.setShowColor(scip,civ);
      }
   }
   if (caclear) ca.clear();
   Word textShow=(cpmverbose<1 ? Word() : text);
   sys.mark(ca,cview,textShow); //This does not handle the
  // text insert cleanly if sys.mark calls ScalarField::mark
  // sys.mark(ca,cview);
   cpmmessage(text,2);
}
if (final){
   ca.write();
   cpmmessage("point final reached");
}
if (cimg){
   sc.countImg+=1;
   Word imf=sc.imgFile;
   imf=imf.cut(4); // cutting ".txt" at the end
   imf=imf.tail(6);
   imf="i_"&imf&"_";
   if (sc.modality=="normalView"&&SysControl::bitmapToShow==3){
      // here an image gets created that has no direct
      // relationship to ca
      CpmGraphics::Img24 img(imgDat);
      img.write(imf&cpm(count,fieldLength));
   }
   else{
      ca.write(imf&cpm(count,fieldLength));
```

```
            // this makes no screen action since it calls finally
            // Img24::write
         }
      }
      if (!waitDone) cpmwait(sc.tWait);
   }
   if (sc.maxNumberImages<=sc.countImg || final) valid=false;
   CPM_MZ
}
```

60 cpmf.h

```
//? cpmf.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_F_H_
#define CPM_F_H_
/*
   Purpose: see cpmfl.h

*/
#include <cpmv.h>
   // includes <cpmfl.h>, where the essential template F<X,Y>
   // is defined
#include <cpmx.h>
   // for cartesian products of classes,

namespace CpmFunctions{

   using CpmArrays::V;

   template <class X, class Y>
   V<Y> apply(F<X,Y>  const& f, V<X>  const& xs)
      // action on sequences: applying f to xs gives the return value of
      // the present function
   {
      Z n=xs.dim();
      V<Y> res(n);
      for (Z i=1;i<=n;i++) res[i]=f(xs[i]);
      return res;
   }

   namespace aux{
template
   <class X, class P1, class P2, class P3, class P4, class P5, class Y>
class Par5FncObj : public FncObj<X,Y>{

   const P1 p1;
   const P2 p2;
   const P3 p3;
   const P4 p4;
   const P5 p5;
   Y (* const fp)
      (X const&, P1 const&, P2 const&, P3 const&, P4 const&, P5 const&);
public:
   Y operator()(X const& x)const
      { return (*fp)(x,p1,p2,p3,p4,p5);}
   Par5FncObj(
      Y (*g)(X const&,P1 const&,P2 const&,P3 const&,P4 const&,P5 const&),
```

```
      P1 const& q1,P2 const& q2,P3 const& q3,P4 const& q4, P5 const& q5):
      fp(g),p1(q1),p2(q2),p3(q3),p4(q4),p5(q5){}
};

template
   <class X, class P1, class P2, class P3, class P4,
   class P5, class P6, class Y>
class Par6FncObj : public FncObj<X,Y>{

   const P1 p1;
   const P2 p2;
   const P3 p3;
   const P4 p4;
   const P5 p5;
   const P6 p6;
   Y (* const fp)(X const&, P1 const&, P2 const&, P3 const&,
      P4 const&, P5 const&, P6 const&);
public:
   Y operator()(X const& x)const
      { return (*fp)(x,p1,p2,p3,p4,p5,p6);}
   Par6FncObj(
      Y (*g)(X const&,P1 const&,P2 const&,P3 const&,P4 const&,
         P5 const&, P6 const&),
         P1 const& q1,P2 const& q2,P3 const& q3,
         P4 const& q4, P5 const& q5, P6 const& q6):
         fp(g),p1(q1),p2(q2),p3(q3),p4(q4),p5(q5),p6(q6){}
};

} // aux

//////////////////////// class F5<> ////////////////////////////////////////

template <class X, class P1, class P2, class P3, class P4,
   class P5, class Y>
class F5{ // functions with five parameters
    const P1 p1;
    const P2 p2;
    const P3 p3;
    const P4 p4;
    const P5 p5;
    typedef F5<X,P1,P2,P3,P4,P5,Y> Type;
    CPM_INVAR(F5)
public:
    F5( P1 const& p1_,P2 const& p2_,P3 const& p3_,P4 const& p4_,
       P5 const& p5_):
    p1(p1_),p2(p2_),p3(p3_),p4(p4_),p5(p5_){}
    F<X,Y> operator()(Y (*f)(X const&,P1 const&,P2 const&,
       P3 const&,P4 const&,P5 const&))const;
};
```

```
template <class X, class P1, class P2, class P3, class P4,
   class P5,  class Y>

F<X,Y> F5<X,P1,P2,P3,P4,P5,Y>::operator()(Y (*f)(X const&,P1 const&,
   P2 const&, P3 const&,P4 const&,P5 const&))const
{
   using namespace aux;
   return
   F<X,Y>(new Par5FncObj<X,P1,P2,P3,P4,P5,Y>(f,p1,p2,p3,p4,p5));
}

//////////////////////// class F6<> ////////////////////////////////////////

template <class X, class P1, class P2, class P3, class P4,
   class P5, class P6, class Y>
class F6{ // functions with six parameters
    const P1 p1;
    const P2 p2;
    const P3 p3;
    const P4 p4;
    const P5 p5;
    const P6 p6;
    typedef F6<X,P1,P2,P3,P4,P5,P6,Y> Type;
    CPM_INVAR(F6)
public:
    F6( P1 const& p1_,P2 const& p2_,P3 const& p3_,P4 const& p4_,
       P5 const& p5_, P6 const& p6_):
    p1(p1_),p2(p2_),p3(p3_),p4(p4_),p5(p5_),p6(p6_){}
    F<X,Y> operator()(Y (*f)(X const&,P1 const&,P2 const&,
       P3 const&,P4 const&,P5 const&, P6 const&))const;
};


template <class X, class P1, class P2, class P3, class P4,
   class P5, class P6, class Y>

F<X,Y> F6<X,P1,P2,P3,P4,P5,P6,Y>::operator()(Y (*f)(X const&,P1 const&,
   P2 const&, P3 const&,P4 const&,P5 const&, P6 const&))const
{
   using namespace aux;
   return
   F<X,Y>(new Par6FncObj<X,P1,P2,P3,P4,P5,P6,Y>
      (f,p1,p2,p3,p4,p5,p6));
}

namespace aux{

template <class X1, class X2, class Y1, class Y2>
class cart: public FncObj<cpmX2<X1,X2>,cpmX2<Y1,Y2> >{ // CART stands for
    // Cartesian
```

```
   const F<X1,Y1> f1;
   const F<X2,Y2> f2;

public:
   cpmX2<Y1,Y2> operator()(const cpmX2<X1,X2>& x)const
   { return cpmX2<Y1,Y2>(f1(x.first),f2(x.second));}

   cart(const F<X1,Y1>& f1_,const F<X2,Y2>& f2_):f1(f1_),f2(f2_){}
};

// pairing of functions which are defined on the same class

//////////////////////// class cart1<> ////////////////////////////////

template <class X, class Y1, class Y2>
class cart1: public FncObj<X,cpmX2<Y1,Y2> >{

   const F<X,Y1> f1;
   const F<X,Y2> f2;
public:
   cpmX2<Y1,Y2> operator()(X const& x)const
   { return cpmX2<Y1,Y2>(f1(x),f2(x));}

   cart1(const F<X,Y1>& f1_,const F<X,Y2>& f2_):f1(f1_),f2(f2_){}
};

} // aux

//////////////////// function pairOf<> ////////////////////////////////

template <class X1, class X2, class Y1, class Y2>
F<cpmX2<X1,X2>,cpmX2<Y1,Y2> > pairOf(const F<X1,Y1>& f1,
   const F<X2,Y2>& f2)
   // Cartesian product of functions:
   // cpmX2(f1,f2)(x1,x2)=(f1(x1),f2(x2)).
   // name changed from pair to pairOf to avoid clash with std name
   // (99-5-31)
{
   using namespace aux;
   return F<cpmX2<X1,X2>,cpmX2<Y1,Y2> >(new cart<X1,X2,Y1,Y2>(f1,f2));
}

//////////////////////// operator&&<> ////////////////////////////////

template <class X, class Y1, class Y2>
F<X,cpmX2<Y1,Y2> > operator &&(const F<X,Y1>& f1,const F<X,Y2>& f2)
   // Cartesian product of functions which are defined on the same class.
    // (f1&&f2)(x)=(f1(x),f2(x)).
    // Elegant way for flexible definition of binary operators by
```

```
    // concatenation
    // with a mapping cpmX2<Y1,Y2> ---> Y3
{
   using namespace aux;
   return F<X,cpmX2<Y1,Y2> >(new cart1<X,Y1,Y2>(f1,f2));
}

// 'changing the number of variables from 2 to 1'

// Bind1 and Bind2 as classes of the same design as FuncPar.
// Assume:
// class X1;
// class X2;
// class Y;
// F<cpmX2<X1,X2>,Y> f(...);
// X1 x1=...;
// F<X2,Y> f1=Bind1<X1,X2,Y>(x1)(f);
// X2 x2=...;
// Y ya=f(cpmX2<X1,X2>(x1,x2));
// Y yb=f1(x2);
// then ya==yb

namespace aux{

template <class X1, class X2, class Y>
class bind1: public FncObj<X2,Y>{
   // bind1 stands for bind first variable
   const X1 x1;
   const F<cpmX2<X1,X2>,Y> f;
public:
   Y operator()(const X2& x2)const{ return f( cpmX2<X1,X2>(x1,x2));}
   bind1( const F<cpmX2<X1,X2>,Y> & f_, const X1& x1_):f(f_),x1(x1_){}
};

template <class X1, class X2, class Y>
class bind2: public FncObj<X1,Y>{
   // bind2 stands for bind second variable
   const X2 x2;
   const F< cpmX2<X1,X2>,Y>  f;
public:
   Y operator()(const X1& x1)const{ return f( cpmX2<X1,X2>(x1,x2));}
   bind2( const F< cpmX2<X1,X2>,Y>& f_, const X2& x2_):f(f_),x2(x2_){}
};

} // aux

//////////////////////// class Bind1<> ////////////////////////////////

template <class X1, class X2, class Y>
class Bind1 { // binding the first parameter
```

```
    const X1 x1;
public:
    Bind1(const X1& x1_):x1(x1_){}
    F<X2,Y> operator()(const F<cpmX2<X1,X2>,Y>& f)const
    { return F<X2,Y>(new aux::bind1<X1,X2,Y>(f,x1));}
};

//////////////////////// class Bind2<> ////////////////////////////////

template <class X1, class X2, class Y>
class Bind2 { // binding the second parameter
    const X2 x2;
public:
    Bind2(const X2& x2_):x2(x2_){}
    F<X1,Y> operator()(const F<cpmX2<X1,X2>,Y>& f)const
    { return F<X1,Y>(new aux::bind2<X1,X2,Y>(f,x2));}
};

// range and domain conversion

namespace aux{

template <class X1, class X2, class Y>
class convertDomain: public FncObj<X2,Y>{
   // means to transform a F<X,Y> in a F<X',Y> if there is an
   // automatic conversion from X to X'

   const F<X1,Y> f;
public:
   Y operator()(const X2& x)const{ return f(x);}
   convertDomain( const F<X1,Y>& f_):f(f_){}
};

template <class X, class Y1, class Y2>
class convertRange: public FncObj<X,Y2>{
   // means to transform a F<X,Y> in a F<X,Y'> if there is an
   // automatic conversion from Y to Y'

   const F<X,Y1> f;

public:
   Y2 operator()(X const& x)const{ return f(x);}
   convertRange( const F<X,Y1>& f_):f(f_){}
};

} // auxiliatry

////////////////////// class ConvertDomain<> ////////////////////////

template <class X1, class X2, class Y>
```

```
// usage e.g. :
//   F<R,C> f=.....
//   ConvertDomain<R,N,C> converter;
//   F<N,C> f_N=converter(f);
// The same converter object can be used for converting other functions
// of the same domain and range classes.
class ConvertDomain{ // converting the function domain
public:
   ConvertDomain(void){}
   F<X2,Y> operator()(const F<X1,Y>& f)const
      // transforms a F<X1,Y> in a F<X2,Y> if there is an
      // automatic conversion from X1 to X2.
   {
      return F<X2,Y>(new aux::convertDomain<X1,X2,Y>(f));
   }
};

//////////////////////// class ConvertRange ////////////////////////////

template <class X, class Y1, class Y2>
// Usage:
//   F<R,C> f=.....
//   ConvertRange<R,C,Cd> converter;
//   F<R,Cd> f_Cd=converter(f);
// The same converter object can be used for converting other functions
// of the same domain and range classes.
class ConvertRange{ // converting the function range
public:
   ConvertRange(void){}
   F<X,Y2> operator()(const F<X,Y1>& f)const
      // transforms a F<X,Y1> in a F<X,Y2> if there is an
      // automatic conversion from Y1 to Y2.
   {
      return F<X,Y2>(new aux::convertRange<X,Y1,Y2>(f));
   }
};

/// functions of more than one variable

namespace aux{

template <class Y1, class Y2, class Y3>
class Arg2FncObj : public FncObj<cpmX2<Y1,Y2>, Y3>{

   const Y3 (*fp)(const Y1&, const Y2& );

public:
   Y3 operator()(const cpmX2<Y1,Y2>& x)const
   { return fp(x.c1(),x.c2());}
```

```
   Arg2FncObj( Y3 (*g)(const Y1&, const Y2&)):fp(g){}
};

} // aux

//////////////////// class F_2<> ////////////////////////////////////////

template <class Y1, class Y2, class Y3>
class F_2{ // functions of two variables

   const F<cpmX2<Y1,Y2>,Y3> f;

public:

   F_2(Y3 (*f_)(const Y1&, const Y2&)):
      f(new aux::Arg2FncObj<Y1,Y2,Y3>(f_)){}
      // construction from function pointer

   F_2(const F<CpmArrays::X2<Y1,Y2>,Y3>& f_):f(f_){}
      // construction from 'smart function object'

   F<CpmArrays::X2<Y1,Y2>,Y3> operator()(void)const{ return f;}
      // evaluation to 'smart function object'

   Y3 operator()(const Y1& y1, const Y2& y2)const
      // evaluation to value
   { return f(CpmArrays::X2<Y1,Y2>(y1,y2));}

   F<Y2,Y3> c1(const Y1& y1)const
      // evaluation to a 'smart function object'
      // representing a function of one variable;
      // obtained by binding the first of the two variables
   { return Bind1<Y1,Y2,Y3>(y1)(f);}

    F<Y1,Y3> c2(const Y2& y2)const
     // ...  binding the second of the two variables
   { return Bind2<Y1,Y2,Y3>(y2)(f);}

};

// define a function by selecting one function value from a two functions

namespace aux{

template <class X, class Y>
class sel: public FncObj<X,Y>{

   const F<X,Y>  f1;
   const F<X,Y>  f2;
   const F<X,Z>  s;
```

```
public:

   sel(const F<X,Y> & f1_, F<X,Y>  const& f2_, const F<X,Z>& s_):
      f1(f1_),f2(f2_),s(s_){}

   Y operator()(X const& )const;
};

template <class X, class Y>
Y sel<X,Y>::operator()(X const& x)const
{
   Z i=s(x);
   if (i==1) return f1(x);
   else if (i==2) return f2(x);
   else {
      cpmerror
         ("select::operator(): unvalid value of selection function");
      return f1(x); // never done
   }
}

//////////////////// class vselect<> ////////////////////////////////////

// define a function by selecting one function value from a sequence of
// functions

template <class X, class Y>
class vselect: public FncObj<X,Y>{

   const V< F<X,Y> > f;
   const F<X,Z>  sel;

public:

   vselect(const V< F<X,Y> >& f_, const F<X,Z> & sel_):
      f(f_),sel(sel_){}

   Y operator()(X const& )const;

};

template <class X, class Y>
Y vselect<X,Y>::operator()(X const& x)const
{
   Z i=sel(x);
   return (f[i])(x);
}

} // aux
```

```
//////////////////// function select<> ////////////////////////////////////

template <class X, class Y>
F<X,Y> select(F<X,Y>  const& f, F<X,Y>  const& g, const F<X,Z>& h)
// The returned function res has the property res(x)==f(x) if h(x)==1,
// res(x)== g(x) if h(x)==2, cpmerror() else
{
   return F<X,Y>(new aux::sel<X,Y>(f,g,h));
}

template <class X, class Y>
F<X,Y> select(const V<F<X,Y> >& f, const F<X,Z>& h)
// The returned function res has the property res(x)=(f[g(x)])(x);
{
   return F<X,Y>(new aux::vselect<X,Y>(f,h));
}

} // namespace

#endif
```

61 cpmfa.h

```
//? cpmfa.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_FA_H_
#define CPM_FA_H_
//////////////////////////////////////////////////////////////////////////
//
// Purpose: Define classes describing function-like objects with
//   arithmetic operations
//
//////////////////////////////// class FUOBJ<> ////////////////////////////
#include <cpmfo.h>
#include <cpmtypes.h>

namespace CpmFunctions{

   using std::ostream;
   using std::istream;
   using CpmRoot::Z;

   namespace auxiliary { // is within a Cpm... namespace !

//////////////////////// tools for defining class Fa<X,Y> ////////////////////

// Advanced arithmetics of functions based on handles to FncObjs

// Arithmetic operations: Here it is assumed that Y is a ring i.e. that
// Y+Y, Y-Y, Y*Y, -Y are defined. Also division Y/Y is assumed to be
// defined. Also template functions/operators will be implemented which
// do not depend only on X and Y.

// class SUMFUO<>

template <class X, class Y>
class SUMFUO: public BINOPFUO<X,Y>{
   typedef SUMFUO<X,Y> Type;
   CPM_INVAR(SUMFUO)
public:
   Y operator()(const X& x)const{ return f1(x)+f2(x);}
   SUMFUO( const F<X,Y> & f1In, const F<X,Y>& f2In)
   :BINOPFUO<X,Y>(f1In,f2In){}
};

// class DIFFFUO<>

template <class X, class Y>
class DIFFFUO: public BINOPFUO<X,Y>{
```

```
public:
   Y operator()(const X& x)const{ return f1(x)-f2(x);}
   DIFFFUO( const F<X,Y> & f1In, const F<X,Y> & f2In)
   :BINOPFUO<X,Y>(f1In,f2In){}
};

// class NEGFUO<>

template <class X, class Y>
class NEGFUO: public MONOPFUO<X,Y>{

public:
   Y operator()(X const& x)const{ return -f1(x);}
   NEGFUO(const F<X,Y> & f1In):MONOPFUO<X,Y>(f1In){}
};

// class PRODFUO<>

template <class X, class Y>
class PRODFUO: public BINOPFUO<X,Y>{

public:
   Y operator()(const X& x)const{ return f1(x)*f2(x);}
   PRODFUO( const F<X,Y> & f1In, const F<X,Y> & f2In)
   :BINOPFUO<X,Y>(f1In,f2In){}
};

// class DIVIFUO<>

template <class X, class Y>
class DIVIFUO: public BINOPFUO<X,Y>{

public:
   Y operator()(const X& x)const{ return  f1(x)/f2(x);}
   DIVIFUO( const F<X,Y> & f1In, const F<X,Y> & f2In)
   :BINOPFUO<X,Y>(f1In,f2In){}
};

template <class X, class Y>
class INVFUO : public FncObj<X,Y>{
   const F<X,Y> f;
public:
   INVFUO(const F<X,Y>& f_):f(f_){}
   Y operator()(const X& x)const
   { return CpmRoot::inv<Y>(f(x));}
};

// treating operations which need more than 2 template arguments
```

```
template <class X1, class X2, class X3>
class TENSFUO: public FncObj< cpmX2<X1,X2>, X3 >{
   // TENS stands for Tensor Product

   const F<X1,X3>  f1;
   const F<X2,X3>  f2;

public:
   X3 operator()(const cpmX2<X1,X2>& x)const
      { return f1(x.get1())*f2(x.get2());}
   TENSFUO( const F<X1,X3> & f1In, const F<X2,X3> & f2In):
      f1(f1In),f2(f2In){}
};

template <class X1, class X2, class X12, class Y1, class Y2, class Y12>
class GENPRODFUO: public FncObj< X12,Y12 >{
   // GENPROD stands for general product. We assume that X12 is a class
   // which defines functions get1() and get2() such that for each x12 in
   // X12 the values of x12.get1() and x12.get2() allow unique automatic
   // conversion to X1 and X2 respectively. Further, we assume that for
   // each y1 in Y1 and y2 in Y2 the value of Y12(y1,y2) is an element in
   // Y12. Notice that the normal tensor product
   // TENSFUO<X1,X2,X3>(f1,f2)
   // would be obtained as
   // GENPRODFUO<X1,X2,cpmX2<X1,X2>,Y3,Y3,Triv<Y3> > (f1,f2)
   // where
   //    template <class X>
   //    class Triv: public X{
   //    public:
   //       Triv(const X& x):X(x){}
   //       Triv(const X& x1, const X& x2):X(x1*x2){}
   //   };
   //

   const F<X1,Y1>  f1;
   const F<X2,Y2>  f2;

public:
   Y12 operator()(const X12& x)const{ return Y12(f1(x.c1()),f2(x.c2()));}
   GENPRODFUO( const F<X1,Y1> & f1In, const F<X2,Y2>& f2In):
      f1(f1In),f2(f2In){}
};


template <class X, class Y, class S>
   // assumption S*Y defined, in typical situations, Y is a linear space
   // over S

class MULTFUO: public FncObj<X,Y>{
```

```
   const F<X,Y>  f1;
   const S s;

public:

   Y operator()(const X& x)const{ return s*(f1(x));}

   MULTFUO(const S& sIn, const F<X,Y> & f1In):f1(f1In),s(sIn){}

};

template <class X, class Y>
   // assumption Y+Y defined

class Y_SUMFUO: public FncObj<X,Y>{

   const F<X,Y>  f1;
   const Y y;

public:

   Y operator()(const X& x)const{ return (f1(x))+y;}

   Y_SUMFUO(const F<X,Y> & f1In, const Y& yIn):f1(f1In),y(yIn){}

};

template <class X, class Y>
   // assumption Y-Y defined

class Y_DIFFFUO: public FncObj<X,Y>{

   const F<X,Y> f1;
   const Y y;

public:

   Y operator()(const X& x)const{ return f1(x)-y;}

   Y_DIFFFUO(const F<X,Y> & f1In, const Y& yIn):f1(f1In),y(yIn){}

};

template <class X, class Y>
   // assumption Y*Y defined

class Y_PRODFUO: public FncObj<X,Y>{

   const F<X,Y>  f1;
   const Y y;
```

```
public:

   Y operator()(const X& x)const{ return f1(x)*y;}

   Y_PRODFUO(const F<X,Y> & f1In, const Y& yIn):f1(f1In),y(yIn){}

};

template <class X, class Y>

class Y_DIVIFUO: public FncObj<X,Y>{

protected:

   const F<X,Y>  f1;
   const Y y;

public:

   Y operator()(const X& x)const{ return f1(x)/y;}

   Y_DIVIFUO(const F<X,Y> & f1In,const Y& yIn):f1(f1In),y(yIn){}

};

} // auxiliary

//////////////////////////////////// class Fa<X,Y> ////////////////////////

// Arithmetics and some additional functionality for Fo<X,Y>

// Now arithmetics is introduced. Just as in class Va, we assume that
// -Y, Y+Y, Y-Y, Y*Y, Y/Y , Y < Y, Y > Y, are defined
// Note 02-05-24:
// Notice that e.g. addition of functions is not implemented by really
// 'adding something together' but arranging pointers to the code
// of all functions in the sum in a manner that when the sum function is
// asked to be evaluated for some value of the argument, all functions
// needed in the sum are ready for evaluation and addition of the
// output-values. This is sometimes called 'deferred evaluation'.
// I found that code of type:
// Fa<X,Y> f;
// Z i,n=3300;
// V< Fa<X,Y> > vf(n);
// for (i=1;i<=n;i++) vf[i]=...;
// for (i=1;i<=n;i++) f+=vf[i];
// caused stack overflow in the last line

template <class X, class Y>
```

```
class Fa: public Fo<X,Y> { // version of Fo with arithmetics operations

public:

   typedef Fa<X,Y> Type;
   typedef Y ScalarType;

   Fa(void):Fo<X,Y>(){}
      // default constructor

   Fa(const F<X,Y>& g):Fo<X,Y>(g){}
      // downcast constructor

   Fa(const Fo<X,Y>& g):Fo<X,Y>(g){}
      // downcast constructor

   Fa(const Fa<X,Y>& g):Fo<X,Y>(g){}
      // copy constructor

   Fa(FncObj<X,Y>* fop):Fo<X,Y>(fop){}
      // constructor from pointers to FncObj

// Construction from function pointers

   Fa( Y (*f)(const X& )):Fo<X,Y>(f){}
   Fa( Y (*f)(X)):Fo<X,Y>(f){}

// Construtor for constant function

   explicit Fa(const Y& y1):Fo<X,Y>(y1){}

   virtual F<X,Y>* clone()const{ return new Fa(*this);}

   virtual Word nameOf()const
   {
      Word wi="Fa<";
      X x; Y y;
      Word wx=CpmRoot::Name<X>()(x);
      Word wy=CpmRoot::Name<Y>()(y);
      return wi&wx&","&wy&">";
   }

   CPM_SUM_C
   CPM_PRODUCT_C
   CPM_DIFFERENCE
   CPM_DIVISION
   CPM_SCALAR_C
   CPM_IO_V
};
```

```
// functions(operators) outside this class. Here additional template
// arguments are allowed, which creates the flexibility needed for
// treating composition and tensor product.

template <class X1, class X2, class X3>

Fa<cpmX2<X1,X2>,X3> operator || (const Fa<X1,X3>& f13,
         const Fa<X2,X3>& f23)
   // Tensor product of functions: Assumption is that X3*X3 is defined.
   // This creates functions of two variables out of two functions of one
   // variable. Symbol || should remind to the old symbol )( for the
   // 'dyadic product' which is essentially the same construction
{
   using namespace auxiliary;
   return Fa<cpmX2<X1,X2>,X3>(new TENSFUO<X1,X2,X3>(f13,f23));
}

template <class X, class Y, class S>
   // assumption S*Y defined, in typical situations,
   // Y is a linear space over S

Fa<X,Y> mult(const S& sIn, const Fa<X,Y>& fIn)
   // Function values are multiplied by s, operator * instead of
   // mult results in compiler errors caused by mistaking this as
   // another operator *
{
   using namespace auxiliary;
   return Fa<X,Y>(new MULTFUO<X,Y,S>(sIn,fIn));
}

template <class X, class Y>
Fa<X,Y> Fa<X,Y>::neg(void)const
{
   using namespace auxiliary;
   return Fa<X,Y>(new NEGFUO<X,Y>(*this));
}

template <class X, class Y>
Fa<X,Y> Fa<X,Y>::inv(void)const
{
   return Fa<X,Y>(new auxiliary::INVFUO<X,Y>(*this));
}

template <class X, class Y>
Fa<X,Y> Fa<X,Y>::net(CpmRoot::Z i)const
{
   Y y0;
   Y yn=CpmRoot::net<Y>(y0,i);
   return Fa<X,Y>(yn);
}
```

```
template <class X, class Y>
Fa<X,Y> Fa<X,Y>::operator +(const Fa<X,Y>& f )const
{
   using namespace auxiliary;
   return Fa<X,Y>(new SUMFUO<X,Y>(*this,f));
}

template <class X, class Y>
Fa<X,Y> Fa<X,Y>::operator *(const Fa<X,Y>& f )const
{
   using namespace auxiliary;
   return Fa<X,Y>(new PRODFUO<X,Y>(*this,f));
}

template <class X, class Y>
Fa<X,Y> Fa<X,Y>::operator *(const Y& y )const
{
   using namespace auxiliary;
   return Fa<X,Y>(new Y_PRODFUO<X,Y>(*this,y));
}

template <class X, class Y>
Fa<X,Y> Fa<X,Y>::operator +(const Y& y )const
{
   using namespace auxiliary;
   return Fa<X,Y>(new Y_SUMFUO<X,Y>(*this,y));
}


template <class X, class Y>
Fa<X,Y> operator *( const Y& y, const Fa<X,Y>& f1 )
{
//   using namespace auxiliary;
   return (f1*y);
}

// I/O interface trivial. No assumtions on X,Y

template <class X, class Y>
bool Fa<X,Y>::prnOn(ostream& out)const
{
   return CpmRoot::writeTitle(" a function",out);
}

template <class X, class Y>
bool Fa<X,Y>::scanFrom(istream& in)
{
   in;
   return (in!=0);
```

```
}

} // namespace

#endif
```

62 cpmfft.h

```
//? cpmfft.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_FFT_H
#define CPM_FFT_H
/*

   Description: Fourier transform of new effective template
      matrices and tensors. Intended for fourier transformation of
      images. E.g. determination of frequency response of cameras
      by dividing the fourier transform of a camera image by
      the fourier transform of the corresponding areal scene (e.g.
      test pattern). More functionality is provided by
      cpmfftmore.(h,cpp)

*/

#include <cpmc.h>
#include <cpmv.h>

// Implementation based on a modification of
// function fourn in Press et al.,
// Numerical Recipes in C, second edition, p. 523

/*** Definitions concerning the Discrete Fourier Transform *************
We consider complex-valued arrays with indexes running from 1 to some
even number N=2*n. Since we use i and j as array indexes, we have to
use for the imaginary unit something different, namely I.
We define a mapping f : {1,...,N} --> Q  (rationals)
   f(i) := i<=n ? (i-1)/N : (i-1)/N - 1
and note the following properties:
   f(1)   = 0
   f(2)   = 1/N
   ...      ...
   f(n)   = (n-1)/N = 1/2 - f(2)
   f(n+1) =  n/N -1 = -1/2
   f(n+2) = (n+1)/N-1 = (n+1-N)/N = (-n+1)/N = -f(n)
   ...       ...
   f(N)   =  -1/N = f(2)

Thus all values f(i) lie in the intervall [-1/2, 1/2 - 1/N ].
Roughly f(i) is positive in the first half of the interval
and negative in the last one; the graph of |f(i)|
grows linearly towards the center and then falls linearly
to (nearly) zero.

Now we define for a C-valued array x of dimension N=2*n the
```

C-valued array y=F(x) of the same dimension by

```
   y[i] : =
      N^(-1/2) times sum from j=1 to j=N of x[j]*exp[2*pi*I*(j-1)*f(i)]

   =: (Fy)[i]
```

for all i in {1,...,N}. Of course, this defines the discrete Fourier transform. As the definition stands, it involves repetitious computations that can be very effectively streamlined if N is a power of two and thus gives rise to the fast fourier transform (which, as just explained, is the discrete fourier transformation, but implemented efficiently). What are the properties of F(x) if x is real (i.e. x[i]=x[i]~, where the complex conjugate of z \in C is written as z~)? We compute for y := F(x) for general x (not necessarily real)

```
y[i]~ =
   N^(-1/2) times sum from j=1 to j=N of x[j]~*exp[2*pi*I*(j-1)*-f(i)]

 = ...
```

In order to recognize the right-hand side as F(x*) at some other index position we define the mapping

```
 ~ : {1,...,N} --> {1,...,N}, i |--> i~ by

      i~ := i==1 ? 1 : N+2-i
```

which achieves

```
   f(i~) = -f(i) for all i in {1,...,N}
```

and satisfies

```
   i~~ = i
```

This allows to define two idempotent operations C and K on complex arrays:

```
   (Cx)[i]  := x[i]~  (anti-linear operation)

   (Kx)[i]  := x[i~]  (linear operation)
```

Obviously C K = K C.

With these definitions we resume our calculation ...

```
... = N^(-1/2) times sum from j=1 to j=N of x[j]~*exp[2*pi*I*(j-1)*f(i~)]

      = K(F(Cx)).
```

Thus the whole computation shows

```
      C(F(x)) = K(F(C(x)))

or C F = K F C or

   C' F = F C , where C' := K C = C K   (*)

Notice  (C'x)[i] = x[i~]~ and that C' is an anti-linear idempotent
operation. We have
   x==Cx <==> Fx==C'Fx

Due to the pre-factor N^(-1/2) in the definition, the transformation
F is unitary ( <Fx|Fy> == <x|y> ). This is not the case with the
definition in Press et al.

*/

namespace CpmFourier{

   using CpmRoot::Z;
   using CpmRoot::C;
   using CpmArrays::V;
   using CpmArrays::VV;
   using CpmArrays::VVV;

// general fast Fourier Transform of complex 1-dimensional arrays
// See the 2-dimensional analog for explanations

V<C> fft(const V<C>& x, Z exponent=1);
// For real x and y=fft(x) we have
// y[i]~ = y[i~] (*)
// Assume y comes from a real and positive x (e.g. an linear image), and
// was modified by some image processing algorithm and than
// should be converted back to an image. If the algorithm did
// not some magic to ensure that relation (*) is still valid
// after the manipulation, the back-transformed image will no longer
// be real and ---a fortiori--- no longer positive.
// For multiplicative manipulations T (i.e.  (Ty)[i]:=a[i]*y[i] )
// the persistance of (*) can easily be ensured: one has to
// have a[i]~=a[i~], for real a thus simply a[i]=a[i~].
// If this is satisfied, the back transform is real and to
// get a posive function one could cut the negative parts to zero
// or take the absolute value. Both ways are not very natural and
// so it is interesting that there is a general natural prescription
// for unlimited image manipulations in 'Fourier space':
// Write the positive image x[i] as x[i]=|a[i]|^2 (e.g. a[i]=x[i]^(-1/2),
// a[i] could be chosen as complex in principle). Form
// y=F(a) manipulate y as to get a result y'. Transform back to
// a':=F'y' (here F' is the inverse of F, for which one has the
// same formula as for F but with f replaced cby -f so that
// (F'x)[i]=(Fx)[i~]
```

```
// general fast Fourier Transform of complex 2-dimensional arrays

VV<C> fft(const VV<C>& x, Z exponent=1);
   // exponent 1 for fft, -1 for inverse transformation
   // The dimension of x is arbitrary. It will however be filled
   // with zeros up to the next power of two (not necessarily the same
   // in i and j direction)
   // The connection between matrix indexes in and frequencies is
   // in the previous function.
// For real x and y=fft(x) we have
// y[i][j]~ = y[i~][j~]

// general fast Fourier Transform of complex 3-dimensional arrays
// See the 2-dimensional analog for explanations

VVV<C> fft(const VVV<C>& x, Z exponent=1);
// For real x and y=fft(x) we have
// y[i][j][k]~ = y[i~][j~][k~]

} // namespace

#endif
```

63 cpmfft.cpp

```
//? cpmfft.cpp
//? Status of work 2008-10-25.
//?
#include <cpmfft.h>

using CpmRoot::Z;
using CpmRoot::N;
using CpmRoot::R;
using CpmRoot::C;
using CpmRoot::Word;

using CpmArrays::V;
using CpmArrays::VV;
using CpmArrays::VVV;

namespace{

typedef unsigned long Nd; // thus long in all cases

inline void cpmswp(R& a, R& b){R t=a;a=b;b=t;}

void fourn(R* data, Nd* nn, Z ndim, Z isign)
{
   Z idim;
   Nd i1,i2,i3,i2rev,i3rev,ip1,ip2,ip3,ifp1,ifp2;
   Nd ibit,k1,k2,n,nprev,nrem,ntot;
   R tempi,tempr;
   R theta,wi,wpi,wpr,wr,wtemp;

   for (ntot=1,idim=1;idim<=ndim;idim++)
      ntot *= nn[idim];
   nprev=1;
   for (idim=ndim;idim>=1;idim--) {
      n=nn[idim];
      nrem=ntot/(n*nprev);
      ip1=nprev << 1;
      ip2=ip1*n;
      ip3=ip2*nrem;
      i2rev=1;
      for (i2=1;i2<=ip2;i2+=ip1) {
         if (i2 < i2rev) {
            for (i1=i2;i1<=i2+ip1-2;i1+=2) {
               for (i3=i1;i3<=ip3;i3+=ip2) {
                  i3rev=i2rev+i3-i2;
                  cpmswp(data[i3],data[i3rev]);
                  cpmswp(data[i3+1],data[i3rev+1]);
               }
```

```
            }
         }
         ibit=ip2 >> 1;
         while (ibit >= ip1 && i2rev > ibit) {
            i2rev -= ibit;
            ibit >>= 1;
         }
         i2rev += ibit;
      }
      ifp1=ip1;
      while (ifp1 < ip2) {
         ifp2=ifp1 << 1;
         theta=isign*cpm2pi/(ifp2/ip1);
         wtemp=sin(0.5*theta);
         wpr = -2.0*wtemp*wtemp;
         wpi=sin(theta);
         wr=1.0;
         wi=0.0;
         for (i3=1;i3<=ifp1;i3+=ip1) {
            for (i1=i3;i1<=i3+ip1-2;i1+=2) {
               for (i2=i1;i2<=ip3;i2+=ifp2) {
                  k1=i2;
                  k2=k1+ifp1;
                  tempr=wr*data[k2]-wi*data[k2+1];
                  tempi=wr*data[k2+1]+wi*data[k2];
                  data[k2]=data[k1]-tempr;
                  data[k2+1]=data[k1+1]-tempi;
                  data[k1] += tempr;
                  data[k1+1] += tempi;
               }
            }
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
         }
         ifp1=ifp2;
      }
      nprev *= n;
   }
}

// New systematic treatment of fast Fourier transform
// including the inverse transform in a transparent manner.
// Due to the marvellous properties of function fourn of Press et al.
// The generalization to higher dimensions is trivial.
// According to the new style, only function fourn from Press et al. is
// used.
// All the subtilities of function rlft3 are avoided. If we use the
// method for real data, some wasting of storage is to be admitted since
// we have (artificially in this case), to store also imaginary parts 0.
```

```
N po2(N n)
   // simple method to get the power of two needed for re-dimensioning
   // Returns the smallest power of two which is equal or larger than n.
{
   N k=1;
   while (k<n) k*=2;
   return k;
}

R* linearize(const V<C>& x, N& m)
{
   m=x.dim();
   m=po2(m);

   N m0=x.dim();

   Nd d=2;
   d*=m;
   R* res=new R[d+1];
   N i;
   for (i=0;i<=d;i++) res[i]=0.;
   Nd k=1;
   C val;
   C zero(0,0);
   for (i=1;i<=m;i++){
      val=(i<=m0  ? x[i] : zero);
      res[k++]=val[1];
      res[k++]=val[2];
   }
   return res;
}

V<C> deLinearize(R* x, N m, Z exponent)
{
   Z mL=3;
   static Word loc("deLinearize(R*,N,Z)");
   CPM_MA
   N m1=po2(m);
   cpmassert(m1==m,loc);
   V<C> res(m);
   R valRe, valIm;
   N i;
   Nd k=1;
   R norm=m;
   cpmassert(norm>0,loc);
   norm=1./sqrt(norm);
   for (i=1;i<=m;i++){
      valRe=x[k++]*norm;
      valIm=x[k++]*norm;
      res[i]=C(valRe,valIm);
```

```
   }
   delete[] x;
   CPM_MZ
   return res;
}

} // namespace

V<C> CpmFourier::fft(const V<C>& x, Z exponent)
{
   N m;
   R* v=linearize(x,m);
   Nd* nn=new Nd[2];
   nn[1]=m;
   fourn(v,nn,1,exponent);
   delete[] nn;
   return deLinearize(v,m,exponent);
}

namespace{
R* linearize(const VV<C>& x, N& m, N& n)
   // See Press et al. Numerical Recipes in C, second edition
   // p.523, Fig 12.4.1, data arrangement on input.
   // The reference arguments m,n are used to signal the
   // size of the returned allocated linear memory
   // The return value value is given as a pointer in order to
   // directly fit into function fourn which is left in its pointer
   // using form for efficieny.
   // The dimensions of x have not to be powers of 2.
{
   m=x.dim1();
   n=x.dim2();
   m=po2(m);
   n=po2(n);

   N m0=x.dim1();
   N n0=x.dim2();

   Nd d=2;
   d*=m;
   d*=n;
   R* res=new R[d+1];   // no initialization for efficiency
   N i,j;
   Nd k=1;
   C val;
   C zero(0,0);
   for (i=1;i<=m;i++) for (j=1;j<=n;j++){
      val=(i<=m0 && j<=n0 ? x(i,j) : zero);
      res[k++]=val[1];
      res[k++]=val[2];
```

```
   }
   return res;
}

VV<C> deLinearize(R* x, N m, N n, Z exponent)
{
   Z mL=3;
   static Word loc("deLinearize(R*,N,N,Z)");
   CPM_MA
   N m1=po2(m);
   cpmassert(m1==m,loc);
   N n1=po2(n);
   cpmassert(n1==n,loc);
   VV<C> res(m,n);
   R valRe, valIm;
   N i,j;
   Nd k=1;
   R norm=m*n;
   cpmassert(norm>0,loc);
   norm=1./sqrt(norm);
   for (i=1;i<=m;i++) for (j=1;j<=n;j++){
      valRe=x[k++]*norm;
      valIm=x[k++]*norm;
      res(i,j)=C(valRe,valIm);
   }
   delete[] x;
   CPM_MZ
   return res;
}

} // namespace

// general fast Fourier Transform of complex 2-dimensional arrays

VV<C> CpmFourier::fft(const VV<C>& x, Z exponent)
{
   N m,n;
   R* v=linearize(x,m,n);
      // m,n get there right power of 2 values by this call
   Nd* nn=new Nd[3];
   nn[1]=m;
   nn[2]=n;
   fourn(v,nn,2,exponent);
   delete[] nn;
   return deLinearize(v,m,n,exponent);
      // this efficient return method asks for de-allocation of
      // v within the function deLinearize, this is done there
}

// now for homogeneousity 3-dimensional case in the same style
```

```
namespace{
R* linearize(const VVV<C>& x, N& m, N& n, N& o)

{
   m=x.dim1();
   n=x.dim2();
   o=x.dim3();

   m=po2(m);
   n=po2(n);
   o=po2(o);

   N m0=x.dim1();
   N n0=x.dim2();
   N o0=x.dim3();

   Nd d=2;
   d*=m;
   d*=n;
   d*=o;

   R* res=new R[d+1];   // no initialization for efficiency

   N i,j,k;
   Nd kc=1;
   C val;
   C zero(0,0);
   for (i=1;i<=m;i++) for (j=1;j<=n;j++) for (k=1;k<=o;k++){
      val=(i<=m0 && j<=n0 && k<=o0 ? x(i,j,k) : zero);
      res[kc++]=val[1];
      res[kc++]=val[2];
   }
   return res;
}

VVV<C> deLinearize(R* x, N m, N n, N o, Z exponent)
{
   Z mL=3;
   static Word loc("deLinearize(R*,N,N,N,Z)");
   CPM_MA
   N m1=po2(m);
   cpmassert(m1==m,loc);
   N n1=po2(n);
   cpmassert(n1==n,loc);
   N o1=po2(o);
   cpmassert(o1==o,loc);
   VVV<C> res(m,n,o);
   R valRe, valIm;
   N i,j,k;
```

```
   Nd kc=1;
   R norm=m*n*o;
   cpmassert(norm>0,loc);
   norm=1./sqrt(norm);
   for (i=1;i<=m;i++) for (j=1;j<=n;j++) for (k=1;k<=o;k++){
      valRe=x[kc++]*norm;
      valIm=x[kc++]*norm;
      res(i,j,k)=C(valRe,valIm);
   }
   delete[] x;
   CPM_MZ
   return res;
}
} // namespace

VVV<C> CpmFourier::fft(const VVV<C>& x, Z exponent)
{
   N m,n,o;
   R* v=linearize(x,m,n,o);
      // m,n get there right power of 2 values by this call
   Nd* nn=new Nd[4];
   nn[1]=m;
   nn[2]=n;
   nn[3]=o;
   fourn(v,nn,3,exponent);
   delete[] nn;
   return deLinearize(v,m,n,o,exponent);
      // this efficient return method asks for de-allocation of
      // v within the function deLinearize, this is done there
}
```

64 cpmfftmore.h

```
//? cpmfftmore.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_FFTMORE_H
#define CPM_FFTMORE_H
/*

   Description: Fourier transform imbedded into a suitable
      class frame. We use different classes for space domain
      and frequency domain since the property of being real-valued
      has very different relevance in both cases that should
      be reflected in the definition of the member functions.

*/

#include <cpmfft.h>
#include <cpmva.h>

namespace CpmFourier{

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::C;
   using CpmArrays::V;
   using CpmArrays::VV;
   using CpmArrays::VVV;
   using CpmArrays::Va;
   using CpmArrays::VVa;
   using CpmArrays::VVVa;

//////////////////// classes SD1 and FD1 ////////////////////////////////
class FD1;
class SD1{ // space domain 1D
   Va<C> rep;
   SD1(V<C> const& v):rep(v){} // fast but not save
      // thus private
   friend class FD1;
   typedef SD1 Type;
   typedef C ScalarType;
public:
   CPM_LIN // linear algebra, scalar product, and
      // conjugation. Function con() and fft()
      // differ in SDi and FDi (i=1,2,3); all other
      // functions are the same. So the present implementation
      // which does not derive FDi and SDi from a common base class Di
      // has a lot of code doublication but a clean user interface.
      // Notice the following dilemma: The base class would return
```

```
      // its own instances (typed Di) as return values of arithmetical
      // operations.
      // To interprete these as either FDi or SDi (in order to know what
      // kind of conjugation or Fourier transform would have to be
      // chosen) has then to be done by calling some constructor or other
      // conversion operator. This would be tedious and unnatural for the
      // user. As observed many times: inheritance  and arithmetics
      // don't fit together very well. Arithmetics should not be defined
      // before the last level of inheritance  is reached.
   explicit SD1(Z n=2);
      // all components are C(0,0); even if n is not a power of
      // 2, the array length will be of this kind.
   SD1(V<C> const& v, bool careful);
      // using v as the internal value array rep. It will be taken as it
      // is for careful==false, otherwise it will be extended by zeros up
      // to the next power-of-two length if needed
   SD1(Z n, C z, Z& seed);
      // random generator. The internal value array of length
      // powOf2(n) will be filled by number z.ranVal(k++)
      // with k starting with k=seed. After call seed will have
      // a value that was not used so far.
   FD1 fft()const;
      // discrete fourier transform from space domain to frequency
      // domain
   SD1 re()const{ return (*this+con())*C(0.5);}
   SD1 im()const{ return (*this-con())*C(0.5);}
      // For any SD1 x we have x=x.re()+x.im(),
      // (in the complex analogy z=u+iv, im() corresponds
      // to iv and not to v alone)
      // we have x.re().con()=x.re() and
      // x.im().con()=-x.im()
      // Notice that we have no equality operation defined in the
      // class since this would be useless due to numerical errors
      // 'a=b' above means (a-b).absVal() not significantly above the
      // numerical noise level.
   V<C> toV()const{ return rep;}
      // retreiving the internal value array. This allows a simple
      // V<C> x handled by the presnt class. E.g.
      // x=SD1(x).fft().toV() changes x into its fourier transform.
   Z dim()const{ return rep.dim();}
   C& operator()(Z i){return rep[i];}
   C const& operator()(Z i)const{return rep[i];}
};

class FD1{ // frequency domain 1D
   Va<C> rep;
   FD1(V<C> const& v):rep(v){}
   friend class SD1;
   typedef FD1 Type;
   typedef C ScalarType;
```

```
public:
   CPM_LIN
      // conjugation in frequency domain is conceptionally
      // defined as the conjugation in space domain carried
      // over to frequency domain via fourier transform.
      // As an implementation this would be rather inefficient;
      // fortunately there is the explicit representation
      // as CK as explained in cpmfft.h, equation (*)

   explicit FD1(Z n=2);
   FD1(V<C> const&, bool careful);
   FD1(Z n, C z, Z& seed);
   SD1 fft()const;
      // discrete Fourier transform from frequency domain to
      // space domain
   FD1 re()const{ return (*this+con())*C(0.5);}
   FD1 im()const{ return (*this-con())*C(0.5);}
   V<C> toV()const{ return rep;}
   Z dim()const{ return rep.dim();}
   C& operator()(Z i){return rep[i];}
   C const& operator()(Z i)const{return rep[i];}
};

//////////////////// classes SD2 and FD2 ////////////////////////////////
// See classes SD1 and FD1 for explanations to the various functions
class FD2;
class SD2{ // space domain 2D
   VVa<C> rep;
   SD2(VV<C> const& v):rep(v){} // fast but not save
      // thus private
   friend class FD2;
   typedef SD2 Type;
   typedef C ScalarType;
public:
   CPM_LIN
   explicit SD2(Z n1=2,Z n2=2);
   SD2(VV<C> const&, bool careful);
   SD2(Z n1,Z n2,C z,Z& seed);
      // random generator
   FD2 fft()const;//{ return FD2(CpmFourier::fft(rep,1));}
   SD2 re()const{ return (*this+con())*C(0.5);}
   SD2 im()const{ return (*this-con())*C(0.5);}
   VV<C> toVV()const{ return rep;}
   Z dim1()const{ return rep.dim1();}
   Z dim2()const{ return rep.dim2();}
   C& operator()(Z i,Z j){return rep[i][j];}
   C const& operator()(Z i,Z j)const{return rep[i][j];}
};

class FD2{ // frequency domain 2D
```

```
   VVa<C> rep;
   FD2(VV<C> const& v):rep(v){}
   friend class SD2;
   typedef FD2 Type;
   typedef C ScalarType;
public:
   CPM_LIN
   explicit FD2(Z n1=2,Z n2=2);
   FD2(VV<C> const&, bool careful);
   FD2(Z n1,Z n2,C z,Z& seed);
   SD2 fft()const;
   FD2 re()const{ return (*this+con())*C(0.5);}
   FD2 im()const{ return (*this-con())*C(0.5);}
   VV<C> toVV()const{ return rep;}
   Z dim1()const{ return rep.dim1();}
   Z dim2()const{ return rep.dim2();}
   C& operator()(Z i,Z j){return rep[i][j];}
   C const& operator()(Z i,Z j)const{return rep[i][j];}
};

//////////////////// classes SD3 and FD3 ////////////////////////////////
// See classes SD1 and FD1 for explanations to the various functions
class FD3;
class SD3{ // space domain 3D
   VVVa<C> rep;
   SD3(VVV<C> const& v):rep(v){}
   friend class FD3;
   typedef SD3 Type;
   typedef C ScalarType;
public:
   CPM_LIN
   explicit SD3(Z n1=2,Z n2=2,Z n3=2);
   SD3(VVV<C> const&, bool careful);
   SD3(Z n1,Z n2,Z n3,C z,Z& seed);
   FD3 fft()const;
   SD3 re()const{ return (*this+con())*C(0.5);}
   SD3 im()const{ return (*this-con())*C(0.5);}
   VVV<C> toVVV()const{ return rep;}
   Z dim1()const{ return rep.dim1();}
   Z dim2()const{ return rep.dim2();}
   Z dim3()const{ return rep.dim3();}
   C& operator()(Z i,Z j,Z k){return rep[i][j][k];}
   C const& operator()(Z i,Z j,Z k)const{return rep[i][j][k];}
};

class FD3{ // frequency domain 3D
   VVVa<C> rep;
   FD3(VVV<C> const& v):rep(v){}
   friend class SD3;
   typedef FD3 Type;
```

```
   typedef C ScalarType;
public:
   CPM_LIN
   explicit FD3(Z n1=2,Z n2=2, Z n3=2);
   FD3(VVV<C> const&, bool careful);
   FD3(Z n1,Z n2,Z n3,C z,Z& seed);
   SD3 fft()const;
   FD3 re()const{ return (*this+con())*C(0.5);}
   FD3 im()const{ return (*this-con())*C(0.5);}
   VVV<C> toVVV()const{ return rep;}
   Z dim1()const{ return rep.dim1();}
   Z dim2()const{ return rep.dim2();}
   Z dim3()const{ return rep.dim3();}
};

// Functions that should return only numerical noise
// if everything if implementation is OK
// Actual tests not yet systematic and covering
// see implementation for scope of tests
R fftTest1();
R fftTest2();
R fftTest3();

} // namespace

#endif
```

65 cpmfftmore.cpp

```
//? cpmfftmore.cpp
//? Status of work 2008-10-25.
//?
#include <cpmfftmore.h>

using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::C;
using CpmRoot::Word;

using CpmArrays::V;
using CpmArrays::VV;
using CpmArrays::VVV;

using CpmFourier::SD1;
using CpmFourier::FD1;
using CpmFourier::SD2;
using CpmFourier::FD2;
using CpmFourier::SD3;
using CpmFourier::FD3;

namespace{

Z powOf2(Z n) // see N po2(N) in cpmfft.cpp
{
   Z res=1;
   while (res<n) res*=2;
   return res;
}

}// namespace

////////// classes SD1, SD2, SD3, FD1, FD2, FD3 ////////////////////////
// constructors

SD1::SD1(Z n1):rep(powOf2(n1)){}
SD2::SD2(Z n1,Z n2):rep(powOf2(n1),powOf2(n2)){}
SD3::SD3(Z n1,Z n2,Z n3):rep(powOf2(n1),powOf2(n2),powOf2(n3)){}

FD1::FD1(Z n):rep(powOf2(n)){}
FD2::FD2(Z n1,Z n2):rep(powOf2(n1),powOf2(n2)){}
FD3::FD3(Z n1,Z n2,Z n3):rep(powOf2(n1),powOf2(n2),powOf2(n3)){}

// The parallelism of implementation that is apparent from the previous
// lines will be rationalized for the more complex function bodies

// construction from arrays
```

```
#define CPM_SC1 \
{\
   if (!careful) rep=v;\
   else{\
      Z n=v.dim();\
      Z n2=powOf2(n);\
      if (n==n2) rep=v;\
      else{\
         rep=V<C>(n2);\
         for (Z i=1;i<=n;++i) rep[i]=v[i];\
      }\
   }\
}

SD1::SD1(V<C> const& v, bool careful)
CPM_SC1

FD1::FD1(V<C> const& v, bool careful)
CPM_SC1

#undef CPM_SC1

#define CPM_SC2 \
{\
   if (!careful) rep=v;\
   else{\
      Z n1=v.dim1();\
      Z n2=v.dim2();\
      Z d1=powOf2(n1);\
      Z d2=powOf2(n2);\
      Z i,j;\
      if (d1==n1 && d2==n2) rep=v;\
      else{\
         rep=VV<C>(d1,d2);\
         for (i=1;i<=n1;++i) for (j=1;j<=n2;++j){\
            rep[i][j]=v[i][j];\
         }\
      }\
   }\
}

SD2::SD2(VV<C> const& v, bool careful)
CPM_SC2

FD2::FD2(VV<C> const& v, bool careful)
CPM_SC2

#undef CPM_SC2

#define CPM_SC3 \
```

```
{\
   if (!careful) rep=v;\
   else{\
      Z n1=v.dim1();\
      Z n2=v.dim2();\
      Z n3=v.dim3();\
      Z d1=powOf2(n1);\
      Z d2=powOf2(n2);\
      Z d3=powOf2(n3);\
      Z i,j,k;\
      if (d1==n1 && d2==n2 && d3==n3) rep=v;\
      else{\
         rep=VVV<C>(d1,d2,d3);\
         for (i=1;i<=n1;++i) for (j=1;j<=n2;++j) for (k=1;k<=n3;++k){\
            rep[i][j][k]=v[i][j][k];\
         }\
      }\
   }\
}

SD3::SD3(VVV<C> const& v, bool careful)
CPM_SC3

FD3::FD3(VVV<C> const& v, bool careful)
CPM_SC3

#undef CPM_SC3

// random constructor

#define CPM_SC1 \
{\
   Z i,n2=rep.dim();\
   for (i=1;i<=n2;++i){\
      rep[i]=z.ranVal(seed++);\
   }\
}

SD1::SD1(Z n, C z, Z& seed):rep(powOf2(n))
CPM_SC1

FD1::FD1(Z n, C z, Z& seed):rep(powOf2(n))
CPM_SC1

#undef CPM_SC1

#define CPM_SC2 \
{\
   Z i,j,d1=rep.dim1(),d2=rep.dim2();\
   for (i=1;i<=d1;++i) for (j=1;j<=d2;++j){\
```

```
      rep[i][j]=z.ranVal(seed++);\
   }\
}

SD2::SD2(Z n1, Z n2, C z, Z& seed):rep(powOf2(n1),powOf2(n2))
CPM_SC2

FD2::FD2(Z n1, Z n2, C z, Z& seed):rep(powOf2(n1),powOf2(n2))
CPM_SC2

#undef CPM_SC2

#define CPM_SC3 \
{\
   Z i,j,k,d1=rep.dim1(),d2=rep.dim2(),d3=rep.dim3();\
   for (i=1;i<=d1;++i) for (j=1;j<=d2;++j) for (k=1;k<=d3;++k){\
      rep[i][j][k]=z.ranVal(seed++);\
   }\
}

SD3::SD3(Z n1,Z n2,Z n3,C z,Z& seed):rep(powOf2(n1),powOf2(n2),powOf2(n3)\
)
CPM_SC3

FD3::FD3(Z n1,Z n2,Z n3,C z,Z& seed):rep(powOf2(n1),powOf2(n2),powOf2(n3)\
)
CPM_SC3

#undef CPM_SC3

// arithmetics
// sum
SD1 SD1::operator+(SD1 const& fd)const{ return SD1(rep+fd.rep);}
SD2 SD2::operator+(SD2 const& fd)const{ return SD2(rep+fd.rep);}
SD3 SD3::operator+(SD3 const& fd)const{ return SD3(rep+fd.rep);}

FD1 FD1::operator+(FD1 const& fd)const{ return FD1(rep+fd.rep);}
FD2 FD2::operator+(FD2 const& fd)const{ return FD2(rep+fd.rep);}
FD3 FD3::operator+(FD3 const& fd)const{ return FD3(rep+fd.rep);}

// difference
SD1 SD1::operator-(SD1 const& fd)const{ return SD1(rep-fd.rep);}
SD2 SD2::operator-(SD2 const& fd)const{ return SD2(rep-fd.rep);}
SD3 SD3::operator-(SD3 const& fd)const{ return SD3(rep-fd.rep);}

FD1 FD1::operator-(FD1 const& fd)const{ return FD1(rep-fd.rep);}
FD2 FD2::operator-(FD2 const& fd)const{ return FD2(rep-fd.rep);}
FD3 FD3::operator-(FD3 const& fd)const{ return FD3(rep-fd.rep);}

// minus
```

```
SD1 SD1::operator-()const{ return SD1(-rep);}
SD2 SD2::operator-()const{ return SD2(-rep);}
SD3 SD3::operator-()const{ return SD3(-rep);}

FD1 FD1::operator-()const{ return FD1(-rep);}
FD2 FD2::operator-()const{ return FD2(-rep);}
FD3 FD3::operator-()const{ return FD3(-rep);}

// product with C
SD1 SD1::operator*(C const& z)const{ return SD1(rep*z);}
SD2 SD2::operator*(C const& z)const{ return SD2(rep*z);}
SD3 SD3::operator*(C const& z)const{ return SD3(rep*z);}

FD1 FD1::operator*(C const& z)const{ return FD1(rep*z);}
FD2 FD2::operator*(C const& z)const{ return FD2(rep*z);}
FD3 FD3::operator*(C const& z)const{ return FD3(rep*z);}

// scalar product
C SD1::operator |(SD1 const& fd)const{ return (rep|fd.rep);}
C SD2::operator |(SD2 const& fd)const{ return (rep|fd.rep);}
C SD3::operator |(SD3 const& fd)const{ return (rep|fd.rep);}

C FD1::operator |(FD1 const& fd)const{ return (rep|fd.rep);}
C FD2::operator |(FD2 const& fd)const{ return (rep|fd.rep);}
C FD3::operator |(FD3 const& fd)const{ return (rep|fd.rep);}

// discrete fourier transformation
FD1 SD1::fft()const{ return FD1(CpmFourier::fft(rep,1));}
FD2 SD2::fft()const{ return FD2(CpmFourier::fft(rep,1));}
FD3 SD3::fft()const{ return FD3(CpmFourier::fft(rep,1));}

SD1 FD1::fft()const{ return SD1(CpmFourier::fft(rep,-1));}
SD2 FD2::fft()const{ return SD2(CpmFourier::fft(rep,-1));}
SD3 FD3::fft()const{ return SD3(CpmFourier::fft(rep,-1));}

// conjugation
SD1 SD1::con()const{ return SD1(rep.con());}
SD2 SD2::con()const{ return SD2(rep.con());}
SD3 SD3::con()const{ return SD3(rep.con());}

// conjugation in frequency domain is different

FD1 FD1::con()const
{
   Z i,ic,n=rep.dim();
   Z np=n+2;
   V<C> res(n);
   for (i=1;i<=n;++i){
      ic=(i==1 ? 1 : np-i);
         // ic is what in the general explanation is written i~
```

```
      res[i]=rep[ic].con();
   }
   return FD1(res);
}

FD2 FD2::con()const
{
   Z i,j,ic,jc,n1=rep.dim1(),n2=rep.dim2();
   Z n1p=n1+2, n2p=n2+2;
   VV<C> res(n1,n2);
   for (i=1;i<=n1;++i){
      ic=(i==1 ? 1 : n1p-i);
      for (j=1;j<=n2;++j){
         jc=(j==1 ? 1 : n2p-j);
         res[i][j]=rep[ic][jc].con();
      }
   }
   return FD2(res);
}

FD3 FD3::con()const
{
   Z i,j,k,ic,jc,kc,n1=rep.dim1(),n2=rep.dim2(),n3=rep.dim3();
   Z n1p=n1+2, n2p=n2+2, n3p=n3+2;
   VVV<C> res(n1,n2,n3);
   for (i=1;i<=n1;++i){
      ic=(i==1 ? 1 : n1p-i);
      for (j=1;j<=n2;++j){
         jc=(j==1 ? 1 : n2p-j);
         for (k=1;k<=n3;++k){
            kc=(k==1 ? 1 : n3p-k);
            res[i][j][k]=rep[ic][jc][kc].con();
         }
      }
   }
   return FD3(res);
}

R CpmFourier::fftTest1()
// demonstrates unitarity of the fourier transform
{
   Z n=32;
   C z(1,1);
   Z i,j,k=1;
   VVa<C> x1(n,n);
   VVa<C> x2(n,n);
   for (i=1;i<=n;++i) for (j=1;j<=n;++j){
      x1[i][j]=z.ranVal(k++);
      x2[i][j]=z.ranVal(k++);
   }
```

```
   C sp1=(x1|x2);
   Z ex=-1;
   VVa<C> y1=fft(x1,ex);
   VVa<C> y2=fft(x2,ex);
   C sp2=(y1|y2);
   R diff=(sp1-sp2).absVal();
   return diff;
}

R CpmFourier::fftTest2()
// demonstrates linearity of the fourier transform
// and correctness of the inversion
{
   Z n=256;
   C z1(1,1);
   Z seed=1;
   FD1 x1(n,z1,seed);
   FD1 x2(n,z1,seed);
   FD1 x3=(x1+x2)*z1;
   SD1 y1=x1.fft();
   SD1 y2=x2.fft();
   SD1 y3=(y1+y2)*z1;
   SD1 y3_=x3.fft();
   R diff=(y3-y3_).absVal();
   FD1 x3_=y3.fft();
   R diff2=(x3-x3_).absVal();
   return diff+diff2;
}

R CpmFourier::fftTest3()
// demonstrates correct behavior of re
{
   Z n=256;
   C z(1,1);
   Z seed=1;
   SD1 x(n,z,seed);
   SD1 xc=~x;
   FD1 y=x.fft();
   FD1 yc=~y;
   FD1 yc_=xc.fft();
   R diff=yc.disVal(yc_);
   return diff;
}
```

66 cpmfield.h

```
//? cpmfield.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_FIELD_H_
#define CPM_FIELD_H_
/*
Purpose: Define classes describing function-like objects
   with linear algebra arithmetic operations and transformation
   properties under a Group. Most fields in physics fall under this
   catagories. The space X on which the functions are defined is only
   assumed to transform under G. If we assume for G a structure that
   allows to define local translations, we can derive from that
   gradients and other differential operations.
*/
#include <cpmf.h>


namespace CpmFields{

   using CpmFunctions::F;
   using CpmFunctions::F1;
   using CpmFunctions::F2;
   using CpmFunctions::F3;
   using CpmArrays::V;
   using CpmRoot::R;
   using CpmRoot::Z;

//////////////////////////// class ScalarField<> ////////////////////////////

template <class X, class S, class G>
// assumptions X*=G, S+S, S-S, S*S
class ScalarField: public F<X,S> { // scalar fields
   // function space with linear structure and
   // transformation properties under Group G
   typedef F<X,S> Base;
public:

   ScalarField(void):Base(){}
      // default constructor

   ScalarField(const Base& g):Base(g){}
      // downcast constructor

// Construtor for constant function

   ScalarField(const S& s1):Base(s1){}
```

```
   ScalarField operator +(const ScalarField& x)const;
   ScalarField& operator +=(const ScalarField& x){ return *this=*this+x;}

   ScalarField operator -(const ScalarField& x)const;
   ScalarField& operator -=(const ScalarField& x){ return *this=*this-x;}

   ScalarField operator *(const S& s)const;
   ScalarField& operator *=(const S& s){ return *this=*this*s;}

   ScalarField operator *(const G& g)const;
   ScalarField& operator *=(const G& g){ return *this=*this*g;}
   const F<X,S>& toBase()const{ return *this;}

   static ScalarField sum(const V< ScalarField >& vf);
      // adding many fields together by a single operation

private:

   // the algorithmic content of the class should be available in
   // unbiased form
   // notice that these are member functions, so that the arguments need
   // no template arguments <...> which makes the matter quite compact

   static S plus(const X& x, const ScalarField& f1,
      const ScalarField& f2){ return f1(x)+f2(x);}

   static S diff(const X& x, const ScalarField& f1,
      const ScalarField& f2){ return f1(x)-f2(x);}

   static S mult(const X& x, const ScalarField& f, const S& s)
   { return f(x)*s;}

   static S transform(const X& x, const ScalarField& f, const G& g)
   { X xv=x; return f(xv*=!g); }

   static S sumOfArray(const X& x, const V<ScalarField>& vf)
   { S s0=S(); for (Z i=1;i<=vf.dim();i++) s0+=(vf[i])(x); return s0; }

};

template <class X, class S, class G>
ScalarField<X,S,G> ScalarField<X,S,G>::operator +
(const ScalarField& f)const
{
   return F2<X,ScalarField,ScalarField,S>(*this,f)(plus);
}

template <class X, class S, class G>
ScalarField<X,S,G> ScalarField<X,S,G>::operator -
(const ScalarField& f)const
```

```
{
   return F2<X,ScalarField,ScalarField,S>(*this,f)(diff);
}

template <class X, class S, class G>
ScalarField<X,S,G> ScalarField<X,S,G>::operator * (const S& s)const
{
   return F2<X,ScalarField,S,S>(*this,s)(mult);
}

template <class X, class S, class G>
ScalarField<X,S,G> ScalarField<X,S,G>::operator * (const G& g)const
{
   return F2<X,ScalarField,G,S>(*this,g)(transform);
}

template <class X, class S, class G>
ScalarField<X,S,G> ScalarField<X,S,G>::sum(
   const V< ScalarField<X,S,G> >& vf)
{
   return F1<X,V<ScalarField>,S>(vf)(sumOfArray);
}

//////////////////////////////// class Field<> ////////////////////////

template <class X, class Y, class S, class G>
// assumptions: X*=G, Y*=G, S Y[Z], Y+Y, Y-Y, Y*S
class Field: public F<X,Y> { // fields on which a group acts
// see ScalarField; now the group acts also
// on the values of the field (vector field, spinor field, ..

   typedef F<X,Y> Base;
   typedef ScalarField<X,S,G> FieldS;

public:

   Field(void):Base(){}
      // default constructor

   Field(const F<X,Y>& g):Base(g){}
      // downcast constructor

// Construtor for constant function

   Field(const Y& y1):F<X,Y>(y1){}

   Field operator +(const Field& x)const;
   Field& operator +=(const Field& x){ return *this=*this+x;}

   Field operator -(const Field& x)const;
```

```
   Field& operator -=(const Field& x){ return *this=*this-x;}

   Field operator *(const S& s)const;
   Field& operator *=(const S& s){ return *this=*this*s;}

   Field operator *(const G& g)const;
   Field& operator *=(const G& g){ return *this=*this*g;}

   void setComponent(const FieldS& fs, Z i);
      // makes the ith component of *this equal to the scalar field fs

   FieldS getComponent(Z i)const;
      // returns a selected field component as a scalar field

   static Field sum(const V< Field >& vf);
      // adding many fields together by a single operation

private:  // algorithmic content, see ScalarField
   static Y plus(const X& x, const Field& f1, const Field& f2)
   { return (Y)(f1(x)+f2(x));}

   static Y diff(const X& x, const Field& f1, const Field& f2)
   { return (Y)(f1(x)-f2(x));}

   static Y mult(const X& x, const Field& f, const S& s)
   { return (Y)(f(x)*s);}

   static Y transform(const X& x, const Field& f, const G& g)
   { X xv=x; return f(xv*=!g)*=g; }
      // see UM/Ariton 99-7-22/1 for a proof that this is the correct
      // operation of a group on a field

   static Y setcomp(const X& x, const Field& f, const FieldS& fs,
      const Z& i ){ Y y=f(x); y[i]=fs(x); return y;}

   static R getcomp(const X& x, const Field& f, const Z& i )
      {return f(x)[i];}

   static Y sumOfArray(const X& x, const V<Field>& vf)
   { Y y0=Y(); for (Z i=1;i<=vf.dim();i++) y0+=(vf[i])(x); return y0; }
};

template <class X, class Y, class S, class G>
void Field<X,Y,S,G>::setComponent(const FieldS& fs, Z i)
  // makes the ith component of *this equal to the scalar field fs
{
   (*this)=F3<X,Field,FieldS,Z,Y>(*this,fs,i)(setcomp);
}

template <class X, class Y, class S, class G>
```

```
ScalarField<X,S,G> Field<X,Y,S,G>::getComponent(Z i)const
{
   return FieldS(F2<X,Field,Z,R>(*this,i)(getcomp));
}

template <class X, class Y, class S, class G>
Field<X,Y,S,G> Field<X,Y,S,G>::operator + (const Field& f)const
{
   return F2<X,Field,Field,Y>(*this,f)(plus);
}

template <class X, class Y, class S, class G>
Field<X,Y,S,G> Field<X,Y,S,G>::operator - (const Field& f)const
{
   return F2<X,Field,Field,Y>(*this,f)(diff);
}

template <class X, class Y, class S, class G>
Field<X,Y,S,G> Field<X,Y,S,G>::operator * (const S& s)const
{
   return F2<X,Field,S,Y>(*this,s)(mult);
}

template <class X, class Y, class S, class G>
Field<X,Y,S,G> Field<X,Y,S,G>::operator * (const G& g)const
{
   return F2<X,Field,G,Y>(*this,g)(transform);
}

template <class X, class Y, class S, class G>
Field<X,Y,S,G> Field<X,Y,S,G>::sum(const V< Field<X,Y,S,G> >& vf)
{
   return F1<X,V<Field>,Y>(vf)(sumOfArray);
}

} // namespace
#endif
```

67 cpmfieldoperator.h

```
//? cpmfieldoperator.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_FIELDOPERATOR_H_
#define CPM_FIELDOPERATOR_H_
/*
   Description: objects for describing quantum mechanical operators
      as functions from FieldSpace to FieldSpace
*/

#include <cpmfieldstate.h>
#include <cpmoperator.h>

namespace CpmQuantumMechanics{

   using namespace CpmArrays;
   using namespace CpmFunctions;
   using namespace CpmGraphics;
   using CpmAlgorithms::PeriodicAlarm;

//////////////////////// class FieldOperator ////////////////////////////////

class FieldOperator: public QMBase{
   // field operators as functions FieldState x R -->  FieldState
   typedef FieldOperator Type;
   typedef X2<FieldState,R> XFR;
      // pair of a quantum state and a point in time
   Z d_;
      // number of discrete points on which the one particle
      // states are defined as complex-valued functions.
   F< XFR ,FieldState> f_;
      // we prefer not to rely on Fa<> for implementing arithmetics since
      // we don't need all the stuff implemented there. The direct and
      // lean implementation of arithmetics in cpmfieldobservable.cpp
      // is considered instructive.

   FieldOperator(Z d, F<XFR,FieldState> const& f):d_(d),f_(f){}

   static FieldState conInTime(XFR const& xsr,
      F<FieldState,FieldState> const& f){ return f(xsr.c1());}
      //: constant in time
      // auxiliar device for implementing the constructor
      // FieldOperator(Z,F<FieldState,FieldState>);

   static FieldState mul2(XFR const& xsr, FieldOperator const& op1,
      FieldOperator const& op2)
   {  return op1.f_(XFR(op2.f_(xsr),xsr.c2()));}
```

```
   static FieldState add2(XFR const& xsr, FieldOperator const& op1,
      FieldOperator const& op2)
   { return op1.f_(xsr)+op2.f_(xsr);}

   static FieldState dif2(XFR const& xsr, FieldOperator const& op1,
      FieldOperator const& op2)
   { return op1.f_(xsr)-op2.f_(xsr);}

   static FieldState mulC(XFR const& xsr, FieldOperator const& op,
      C const& z){ return op.f_(xsr)*z;}

   static FieldState mulR(XFR const& xsr, FieldOperator const& op,
      R const&r){  return op.f_(xsr)*r;}

public:

   CPM_PURE_VECTOR_SPACE_INTERFACE(Type,C)
   CPM_PURE_SCALAR_INTERFACE(Type,R)

   FieldOperator(){}

   FieldOperator(R2_Func const& pot,Z d);
      // many-particle Hamiltonian with pot as time-dependent
      // pair potential

   FieldOperator(Z d, F<FieldState,FieldState> const& f);
      // constructor from function

   explicit FieldOperator(Operator const& op);

   FieldOperator operator*(FieldOperator const& fop)const;

   FieldState actOn(FieldState const& st, R t)const
      { return f_(XFR(st,t));}
      // Operator acts on state at time t

   FieldState operator*(XFR const& xsr)const{ return f_(xsr);}
      // another form of actOn;

// generators for creation and destruction operators as static members

   static FieldOperator crt(Z i);
   static FieldOperator des(Z i);
   static FieldOperator crt(State const& st);
   static FieldOperator des(State const& st);
   static FieldOperator psi(Z i);
   static FieldOperator phi(Z i);
   static FieldOperator psi(State const& st);
   static FieldOperator phi(State const& st);
```

```
   static FieldOperator cook1(Observable const& ob);
   static FieldOperator cook2(Observable const& ob);

   static FieldOperator cook1(Operator const& op);
      // trivial implementation so far
   static FieldOperator cook2(Operator const& op);
       // trivial implementation so far

   static FieldOperator manParHam(R2_Func const& pot1,
      R2_Func const& pot2, Z d);
   // Many particle Hamiltonian for one type of particles.
   // pot1 is a potential in one particle space
   // and pot2 a pair potential which, together with d
   // and the data from QMBase define a Hamiltonian.
   static FieldOperator hub(R a, R b, Z d);
   // Hubbard model Hamiltonian
};

////////////////////// class TrjDMI_FilOpr ////////////////////////////
// trajectory for time-dependent Hamiltonians.
// similar to TrjTD but replaces Operators by
// FieldOperators

class TrjDMI_FilOpr: public TrjDMI<FieldState>{//trajectory method: DMI
   PeriodicAlarm pal_;
   FieldOperator op_;
   FieldState op(FieldState const& x, R t)const
      { return op_.actOn(x,t);}
public:
   typedef TrjDMI_FilOpr Type;
   virtual TrjBase* clone()const{ return new TrjDMI_FilOpr(*this);}
   TrjDMI_FilOpr(){}
   TrjDMI_FilOpr(FieldOperator const& h, FieldState const& x,
      TrjCon const& tc):
   TrjDMI<FieldState>(x,tc),op_(h){}
      // creating a trajectory from hamiltonian and initial state
   virtual void showAsArea(Graph& g)const;
   virtual Word text()const{ return Word("TrjDMI_FilOpr");}
};

} // namespace

#endif
```

68 cpmfieldoperator.cpp

```
//? cpmfieldoperator.cpp
//? Status of work 2008-10-25.
//?
#include <cpmoperator.h>
#include <cpmfieldoperator.h>

using CpmQuantumMechanics::QMBase;
using CpmQuantumMechanics::State;
using CpmQuantumMechanics::FieldState;
using CpmQuantumMechanics::Operator;
using CpmQuantumMechanics::FieldOperator;
using CpmQuantumMechanics::Observable;
using CpmQuantumMechanics::TrjCon;
using CpmQuantumMechanics::TrjDMI_FilOpr;
using CpmFunctions::F1;
using CpmFunctions::F2;
using CpmFunctions::R_Func;
using CpmFunctions::R2_Func;

///////////////////////// class FieldOperator //////////////////////////

namespace {
   FieldState cook1Func(X2<FieldState,R> const& fs, Observable const& ob)
   { return fs.c1().cook1(ob);}

   FieldState cook2Func(X2<FieldState,R> const& fs, Observable const& ob)
   { return fs.c1().cook2(ob);}

   FieldState manParHamFunc(X2<FieldState,R> const& fs,
      R2_Func const& pot1, R2_Func const& pot2)
   { return fs.c1().manParHam(pot1,pot2,fs.c2());}
      // calls FieldState::manParHam(...) see cpmfieldstate.h
   FieldState hubFunc(X2<FieldState,R> const& fs,
      R const& a, R const& b){ return fs.c1().hub(a,b,fs.c2());}
}

FieldOperator::FieldOperator(Z d, F<FieldState,FieldState> const& f):d_(d\
)
{
   f_=F1<XFR,F<FieldState,FieldState>,FieldState>(f)(conInTime);
}

FieldOperator FieldOperator::operator*(FieldOperator const& op)const
{
   return
   FieldOperator(d_,
      F2<XFR,FieldOperator,FieldOperator,FieldState>(*this,op)(mul2));
```

```
}

FieldOperator FieldOperator::operator+(FieldOperator const& op)const
{
   return
   FieldOperator(d_,
      F2<XFR,FieldOperator,FieldOperator,FieldState>(*this,op)(add2));
}

FieldOperator FieldOperator::operator-(FieldOperator const& op)const
{
   return
   FieldOperator(d_,
      F2<XFR,FieldOperator,FieldOperator,FieldState>(*this,op)(dif2));
}

FieldOperator FieldOperator::operator*(C z)const
{
   return
   FieldOperator(d_,F2<XFR,FieldOperator,C,FieldState>(*this,z)(mulC));  \

}

FieldOperator FieldOperator::operator*(R r)const
{
   return
   FieldOperator(d_,F2<XFR,FieldOperator,R,FieldState>(*this,r)(mulR));
}

FieldOperator& FieldOperator::operator+=(FieldOperator const& fop)
{
   return *this=operator+(fop);
}

FieldOperator& FieldOperator::operator-=(FieldOperator const& fop)
{
   return *this=operator-(fop);
}

FieldOperator& FieldOperator::operator*=(C z)
{
   return *this=operator*(z);
}

FieldOperator& FieldOperator::operator*=(R r)
{
   return *this=operator*(r);
}

FieldOperator FieldOperator::cook1(Observable const& ob)
```

```
{
   return
   FieldOperator(ob.dim(),F1<XFR,Observable,FieldState>(ob)(cook1Func));
}

FieldOperator FieldOperator::cook2(Observable const& ob)
{
   return
   FieldOperator(ob.dim(),F1<XFR,Observable,FieldState>(ob)(cook2Func));
}

FieldOperator FieldOperator::manParHam(R2_Func const& pot1,
   R2_Func const& pot2, Z d)
{
   return
   FieldOperator(d,F2<XFR,R2_Func,R2_Func,FieldState>
      (pot1,pot2)(manParHamFunc));
   // thus finally uses FieldState::manParHam(...), see function
   // manParHamFunc above.
}

FieldOperator FieldOperator::hub(R a, R b, Z d)
{
   return
   FieldOperator(d,F2<XFR,R,R,FieldState>(a,b)(hubFunc));
   // thus finally uses FieldState::hub(...), see function
   // hubFunc above.
}

FieldOperator FieldOperator::cook1(Operator const& op)
{
   return FieldOperator();
}

FieldOperator FieldOperator::cook2(Operator const& op)
{
   return FieldOperator();
}

//////////////////// class TrjDMI_FilOpr ////////////////////////////////

void TrjDMI_FilOpr::showAsArea(Graph& g)const
{
   Z mL=2;
   static Word loc("TrjDMI_FilOpr::showAsArea(Graph)");
   CPM_MA
/*
   Z ipolMeth=tc_.disMode_-3;
   x_.toObservable(tc_.ignPhs_).showImg(g,ipolMeth);
*/
```

```
   x_.mark(g,tc_.intMet_,tc_.gam_);
   g.display();
   CPM_MZ
}
```

69 cpmfieldstate.h

```
//? cpmfieldstate.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_FIELDSTATE_H_
#define CPM_FIELDSTATE_H_
/*
   Description: Development and test of quantized fields
      in one spatial dimension
*/
//#define CPM_M
// if this define is active, CpmArrays::M<> is used instead of
// std::map<>.
// Much slower, unfortunately: presently the factor is ~ 16

#if !defined(CPM_M)
   #include <map>
#endif

#include <cpmx.h>
#include <cpmoperator.h>

using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::C;
using CpmRootX::Z1;
using CpmRootX::R1;
using CpmGraphics::Frame;
using CpmGraphics::Graph;
using namespace std;
using namespace CpmArrays;
using namespace CpmFunctions;
using CpmArrays::X3;

namespace CpmQuantumMechanics{

//////////////////////////////// class Ky ////////////////////////////////

class Ky{ // index set for an orthonormal base of FieldSpace
   // auxiliary class needed as an index set for an orthonormal
   // base of FieldSpace.
   // The name Key would be inconvenient to use
   // since there is already a class CpmRootX::Key in use.
   // I prefer short names anyway.
   // Actually the instances of Ky are multi-indexes i.e.
   // lists of integers. List length is s_ and list values are
   // not negative in our present case, due to their interpretation
   // as occupation numbers.
```

```
   Z s_;
      // number of space points in the basic space lattice
   Vl<Z> v_;
      // Z-valued array such that v_.dim()==s_.
      // is a list of occupation numbers in the lattice points
   R sigma(Z i)const;
      // function needed to deal with fermions;
   static Z lim_;
      // limit to the occupation numbers, 1 for fermions,
      // large for bosons
   static Z eps_;
      // +1 for fermions , -1 for bosons, set depending on
      // lim_=1 (for fermions) or larger (for bosons)
   static void setEps(void){ if (lim_==1) eps_=1; else eps_=-1;}
      // setting eps_ consistently
   typedef Ky Type;

public:
   CPM_IO
   CPM_ORDER
   CPM_NAM(Ky)
   explicit Ky(Z s=0):s_(CpmRoot::posVal(s)),v_(s_,Z(0)){}
      // Basic constructor
      // no exception for negative s, absolute value used!

   Z dim()const{ return s_;}
     //: dimension

   Z sup()const;
      //: supremum
      // returns the value of the largest component for s_>0
      // and -1 else

   Z inf()const;
      //: infimum
      // returns the value of the smallest component for s_>0
      // and -1 else

   void sup_(Ky const& k);
      //: supremum
      // changes *this into the component-wise maximum
      // of *this and k.
      // Error if both have not the the same dimension

// basic tools for defining creation operators
// (see code defining crt(Z), crt(State)
   bool up_(Z i);
   R up(Z i)const;
// basic tools for defining destruction operators
// (see code defining des(Z), des(State)
```

```
   bool down_(Z i);
   R down(Z i)const;
   Z operator[](Z i)const{ return v_.cui(i);} // needed for crt and des
      // in up_,up,down_,down, and operator[] it is now assumed that the
      // index is in proper range. The destruction and creation operators
      // use this functions in a manner which takes care.

// setting the boson-fermion alternative
   static void setLim(Z i){ lim_=CpmRoot::posVal(i); setEps();}
   static Z getLim(){ return lim_;}
   static Z getEps(){ return eps_;}
   static void setBosons(Z lim=1024){ lim_=lim; setEps();}
   static void setFermions(){ lim_=1;setEps();}
};

class FieldOperator;

//////////////////////////// class FieldState ////////////////////////////

class FieldState:public QMBase{
   // state space in occupation number representation
   // General considerations on finite-dimensional Hilbert spaces.
   // Let an orthonormal basis of Hilbert space H be represented as a
   // class, say B. This means that there is an interpretation according
   // to which each instance of B is an element of an orthonormal basis of
   // H and vice versa. Then each element psi \in H determines a
   // function B --> C, b |--> <b|psi> ( where <|> denotes the scalar
   // product in H and C are the complex numbers) and this mapping
   // H --> (B --> C) is linear and an isomorphism of Hilbert spaces if
   // the class B --> C is made a Hilbert space in a natural manner.
   // The class B-->C is naturally represented as CpmArrays::M<B,C> in C+-
   // or as std::map<B,C> in the C++ standard library (former STL).
   // If the cardinality |B|=:d is large, this is much more economic
   // than the class of all psi \in V<C> with psi.dim()=d.
   // In the present case we deal with a Hilbert space of states for which
   // the dimension d is written as a product of two natural numbers
   // which are the data members s_ and lim_ of the present class. For
   // convenience we refer to these numbers as s and l in the present
   // discussion. There are two natural interpretations of this space:
   //
   // 1. Tensor product of s single-particle spaces, each of dimension l.
   // Here we  consider the single-particle spaces as attached to s space
   // points, which may (or may not) form a linear lattice, the biotope of
   // the system. As mentioned already, these 'localized' state spaces all
   // have the same dimension l, and we consider them as identical to
   // some space l-dimensional Hilbert space H_0 which has an orthonormal
   // base B_0.
   // The physical interpretation is that at each of the s space points
   // (sites) x_i, i=1,...,s,  there is a particle p_i (or other kind of
   // quantum system), which is a fermion for l=1 and a boson else. This
```

```
// is not important for the state space as a Hilbert space but only for
// the definition of operators.
// A basis of H is given by the set of lists (b_1,...b_s) where b_i
// \in B_0. Put differently, a natural basis of H has the form (B_0)^s
// so that the dimension of H is d = l^s and thus grows exponentially
// with s. If, however, l=1 we have d=1 for all s so that this case is
// not interesting for fermions, so lets consider bosons only.
// For bosons it is natural to consider for each site i the 'local'
// creation and destruction operators c_i and d_i satisfying
// [d_i,c_i]=1. Since operators referring to different sites commute
// we have the general canonical relations [d_i,c_j]=\delta_{i,j},
// [d_i,d_j]=0, [c_i,c_j]=0 . Since operators referring directly to
// different sites never anti-commute, a correspondingly trivial
// realization of canonical anti-commutation relations is not possible.
// In the bosonic case, it is a consitent physical picture that at each
// site there is placed a boson (oscillator). The state of the whole
// system is, however, not given by assigning to each site a local
// state. This would only lead to pure product states (nuclear states)
// in the tensor product state. The generic state is a linear
// combination of nuclear states. Notice that this is exactly what
// holds for the state space of s distinguishable bosons. If these
// bosons have a two-dimensional state space (qu-bits), e. g.
// neutral atoms in electronic ground-state with respect to the
// normal fine structure but showing a two-level hyperfine splitting.
// For the states of this bosonic system there is also a representation
// in terms of fermions via the Jordan-Wigner transformation, which we
// do not consider here.

// 2. Fock space over a s-dimensional single-particle space.
// This works for bosons and fermions in nice parallelity and is the
// case to which the creation and destruction operators defined here
// directly refer. It is instructive to look at the single particle
// space as the space of complex functions on B_0 = {x_1,x_2,...,x_s}.
// For each i \in B_0 there is a creation operator c_i and its
// Hermitean conjugate d_i (destruction operator) and a
// orthonormal base of H is of the form
// (n_1! n_2!...n_s!)^{-1/2} {c_1}^{n_1} {c_2}^{n_2}...{c_s}^{n_s} |0>
// =: |n_1,n_2,...,n_s> where n_i \in {0,...s-1}, notice 0!=1
// see e.g. Messiah I, p. 453, (XII.68) For fermions n_i \in {0,1}
// and all pre-factors are 1.
// On the other hand, simply declaring the set B :=
// { (n_1,n_2,...n_s)| n_i \in {0,...s-1}}
// as an orthonormal base can be implemented by considering the
// Hilberspace M<B,C> as mentioned above. In this 'Hilbertspace in
// occupation number representation' creation and destruction operators
// can be defined by the formulas
// d_i |n_1,...n_i,...n_s> = {n_i}^{1/1} |n_1,...,n_i - 1,...n_s>
// c_i |n_1,...n_i,...n_s> = {n_i + 1}^{1/2} |n_1,...,n_i + 1,...n_s>
// (see Schweber, p. 130 (47), p. 131 (51))
// for bosons and
```

```
   // d_i |n_1,...n_i,...n_s> = (-1)^{q_i} n_i |n_1,...,n_i - 1,...n_s>
   // c_i |n_1,...n_i,...n_s> = (-1)^{q_i}(1-n_i) |n_1,...,n_i + 1,...n_s>
   // where q_i := n_1 + n_2 + ... + n_(i-1)
   // (see Schweber, p. 138 (87), p. 139 (89),(90))
   // for fermions
   // As an effect of this definition, the we have
   // |n_1,n_2,...,n_s> =
   // (n_1!n_2!...n_s!)^{-1/2} {c_1}^n_1 {c_2}^n_2...{c_s}^n_s |0,0,...,0>

   Z s_;
      // number of space points, or dimension of single-particle
      // space
   Z lim_;
      // limit for occupation number, 1 for fermions.
      // Should be large enough not to have an influence
      // on results if >1 (then we have bosons). So the type
      // of statistics is a state attribute here.
   bool pbc_;
      // periodic boundary condition: normally true
      // false for reflecting boundary conditions.
      // Influences function b_1(Z&)
#if defined(CPM_M)
   typedef CpmArrays::M<Ky,C> Mkc;
#else
   typedef std::map<Ky,C> Mkc;
#endif
   // associative array either from the standard library or from C+-
   Mkc m_;
   FieldState(Z s, Mkc const& m, Z lim, bool pbc=true):
      s_(s>=0 ? s : -s ), lim_(lim),pbc_(pbc),m_(m){}

   FieldState mod(Mkc const& m)const{return FieldState(s_,m,lim_,pbc_);}
      // returns a modified version of *this

   typedef FieldState Type;
   void b_1(Z& i)const
      // changes j into the periodic or reflected position
   {
      if(i<0){ if (pbc_) i+=s_; else i=-i;}
      else if(i>=s_){ if (pbc_) i-=s_; else i=-i+2*(s_-1);}
      else return;
   }
public:
   CPM_IO
   CPM_SCALAR_PRODUCT_INTERFACE(Type,C)
   CPM_PURE_VECTOR_SPACE_INTERFACE(Type,C)
   CPM_PURE_SCALAR_INTERFACE(Type,R)
   CPM_NAM(FieldState)

   FieldState():s_(1),lim_(1),pbc_(true){}
```

```
FieldState(Z s, Z kind, Z lim, bool pbc=true);
   // Creating the normalized ground state on a lattice of s points
   // (Fock vacuum state)
   // for kind==1 and the zero-vector on the same lattice else

FieldState zero()const{ return FieldState(s_,0,lim_,pbc_);}
   //: zero
   // zero-vector with the same general data as *this

FieldState vac()const{ return FieldState(s_,1,lim_,pbc_);}
   //: vacuum
   // Fock vacuum with the same general data as *this

FieldState(Ky const& k, C z, Z lim, bool pbc=true):
   s_(k.dim()),lim_(lim),pbc_(pbc){ m_[k]=z; }

FieldState(State const& st,Z lim, bool pbc=true);
   // takes the lattice size from n=st.dim(). Creates a
   // FieldState which is obtained by letting the
   // operator sum from i=0 to i=n-1 of crt(i)*st[i] act on
   // the ground state FieldState(n,1). The actual implementation
   // is more direct and more efficient.

FieldState(V<State> const& vs,Z lim, bool pbc=true);
   // Assumes that the components of vs are States on the same
   // lattice.
   // Let m=vs.dim(). Then the generated space is a m-particle
   // state.

void set_(Ky const& k, C z){ m_[k]=z;}
   // notice that mutating functions end in _ by convention

Z size()const{ return s_;}
   // cardinality of the underlying point lattice

Z dim()const{ return s_;}
   // not actually the dimension of FieldState (which is unlimited)
   // but the dimension of the derived State given by function
   // toState.
   // More generally it is the dimension of the one-particle space
   // over which FieldState happens to be the Fock space

C get(Ky const& k)const;
   // gets the value of the wave function at 'occupation index'
   // i.e. at a list of discrete positions given by k.

C operator[](Ky const& k)const{ return get(k);}
   // notice that return m_[k] would not define a constant operator
   // since it sets m_[k]=0 in cases (i.e. for values of k) in which
```

```
   // m_[k] was not given a value earlier.

C operator[](Z i)const;
   // returns a a sum of all those values m_[k] for which k[i]>0.
   // This is OK for rough visualization of the state. If one would
   // use the probability density instead of the amplitude density
   // ('the wave function') this would under the name marginal
   // distribution of the probability density of all particles.
   // Proper index range is 0,1,...s_-1

State toState(bool ignorePhases)const;
   // If ignorePhases==0, this returns a State s such that
   // s.dim()=size() and s[i]=(*this)[i] for all i \in {0,...s_-1}
   // with operator[] just introduced. If ignorePhases!=0 the
   // resulting State res describes the 'marginal probability density'
   // i.e. res[i] is the probability to find a particle at place i
   // irrespective of other particles being arround elsewhere.

State den()const;
   //: density
   // probability density as a distribution over the s_ discrete
   // points.
   // The square roout could be interpreted as a real state that
   // is conveniently to display.
   // Meaning has to be checked, clarify connection to Bethe-Salpeter
   // amplitudes.

Observable toObservable(bool ignorePhases)const;

Type crt(Z i)const;
   // crt =creator = creation operator
   // 'creates an oscillator quantum at lattice position i.
   // Notice that for i out of range (with respect to the state
   // to which the operator is being applied) effectively a lattice
   // point within the proper range is used (considering the
   // index range a torus)


Type des(Z i)const;
   // destructor (not in the sense of C++) = destruction operator.
   // The basic property of crt(i) and des(j) is:
   //    psi.crt(i).des(j)-psi.des(j).crt(i)==psi*krn(i,j)
   // which corresponds to
   //    [a(i),a*(j)]=[des(i),crt(j)]=krn(i,j)*1
   // taking into account the order of application given by
   // postfix notation.
   // Notice that for i out of range (with respect to the state
   // to which the operator is being applied) effectively a lattice
   // point within the proper range is used (considering the
   // index range a torus).
```

```
   // Old explanation which holds for bosons only,
   // see des(State const& st) for the inclusion of fermions.


Type crt(State const& st)const;
   // The natural situation is st.dim()==size().
   // equals sum of crt(i)*st(i)

Type des(State const& st)const;
   // The natural situation is st.dim()==size().
   // equals sum of des(i)*st(i).con()

   // For R eps=Ky::getEps();
   // and State s1,s2; FieldState fs; s1.dim()==s2.dim()==fs.size();
   // fs.crt(s2).des(s1)+fs.des(s1).crt(s2)*eps-fs*(s1|s2)==0

 Type phi(Z i)const{  return (crt(i)+des(i))*facPhi;}
    // The couple (phi,psi) is the natural self-adjoint
    // derivate of the mutually adjoint couple (crt,des)
    // of operators.
    // [phi(i),psi(j)]=C(0,krn(i,j))*1

Type psi(Z i)const{ return (crt(i)-des(i))*facPsi;}
   // see comment to previous function phi

Type phi(State const& st)const{  return (crt(st)+des(st))*facPhi;}
   // Going from from crt(i),des(j) to crt(s1), des(s2) is
   // analogous to going from psi(i),phi(j) to psi(s1),phi(s2)

Type psi(State const& st)const{ return (crt(st)-des(st))*facPsi;}
   // For R eps=Ky::getEps();
   // and State s1,s2; FieldState fs; s1.dim()==s2.dim()==fs.size();
   // C sp12=((s1|s2)-(s1|s2).con()*eps)*C(0,0.5);
   // fs.psi(s2).phi(s1)+fs.phi(s1).psi(s2)*eps-fs*sp12==0;

Type cook1(Observable const& ob)const;
   // Lets the observable ob (assuming ob.dim()==size()) acting
   // on *this by the Fock-Cook construction. Since the matrix
   // elements of ob get automatically periodically continued,
   // it is not necessary that ob.dim()==size() holds.

Type cook2(Observable const& ob)const;
   // Lets the observable ob (assuming ob.dim()==size()) acting
   // on *this by the Fock-Cook construction. Since the matrix
   // elements of ob get automatically periodically continued, it is
   // not necessary that ob.dim()==size() holds.

Type manParHam(R2_Func const& pot1, R2_Func const& pot2, R t)const;
   // 'many particle Hamiltonian'
   // returns a FieldState which results from application of
```

```
   // a many-body hamiltonian with position dependent single particle
   // potential pot1 and pair potential pot2. Both potentials are
   // functions of two variables (instances of R2_Func), where the
   // second argument is time. The first argument is an absolute
   // particle position in case of pot1 and a relative position of
   // a particle pair in the case of pot2.
   // Positions are always given by QMBase::posNor(d) and thus are in
   // the range [-1/2,+1/2]

Type hub(R a, R b, R t)const;
     // Hamiltonian of the Hubbard model

R nor_(R r=1);
   // returns the norm of *this and changes *this into
   // a FieldState of norm r by multiplying with a real factor.
   // If *this was the zero FieldState, it will be turned into
   // a 'standard State' of norm r

R sup()const;
   // max of absolute value of components

Type reduce(Z d=32)const;
   // returns a state res that no occupation number higher
   // than d occurs

void show(Frame& f, Z mult_i=8, Z mult_j=4,
   Z iShift=0, Z jShift=0, bool ignorePhases=false)const;
   //: show
   // This is the comment to the obsolete version. A valid
   // implementation of graphical representation is still missing.
   // notice that the state space has a very high dimension even
   // if s_ and Ky::lim_ have moderate values.
   // one component of rep will be represented as a rectangle of
   // mult_i*mult_j pixels. The whole image will be shifted by
   // (iShift,jShift) with respect to the left upper corner
   // of the graphics window f.
   // Actually the components of rep are not converted to a screen
   // color directly but before get renormalized by a real factor
   // chosen such that the dominant component of rep gets an absolute
   // value of 1. If the true values of the components of a state s
   // are to be shown use
   // Graph g;
   // g.show(s.toRealForm())
   // If ignorePhases differs from 0, what is represented is the
   // probability density instead of the complex-valued wave function.

   void mark(Graph& f, Z ipolMeth=1, R gamma=0.5, R secFac=1.1)const;
      //: mark
      // See State2::mark in cpmstate2.h for the meaning of the
      // arguments.
```

```
   X3<Ky,Z1,R1> dgn()const;
      // diagnosis for state *this. The first component of the result
      // gives the maximum occupation number found at each lattice point,
      // the second component gives the number of non-zero components,
      // and the third component gives the norm of the state.

   Type makeReal()const;
      // returns a FieldState with all components of *this
      // replaced by their absolute value

   static R test(Z com);
      //: test
      // test of the most important relations.
      // returns the sum of errors (also the unavoidable
      // small numeric ones, so that only large values
      // indicate malfunction.
};

} // namespace

#endif
```

70 cpmfieldstate.cpp

```
//? cpmfieldstate.cpp
//? Status of work 2008-10-25.
//?
#include <cpmfieldstate.h>
#include <cpmimagingtools.h>
#include <cpmfft.h>
#include <cpmgraph.h>
#include <cpmx.h>
#include <cpmrvector.h>
#include <cpmfieldoperator.h>
#include <cpmstate2.h> // for using a graphics function from there

using namespace CpmQuantumMechanics;
using namespace CpmRoot;
using namespace CpmArrays;

/************************* class Ky *********************************/

Z Ky::lim_=1;
   // the default value should come into action only for unmanagable
   // high 'occupation numbers' such as 1000

Z Ky::eps_=1;

// not relying verbally on the implementation in Vo<Z>
// for utmost efficiency

Z Ky::com(const Ky& s)const
{
   if (s_<s.s_) return 1;
   else if (s_>s.s_) return -1;
   else{
      for (Z i=0;i<s_;i++){
         if (v_.cui(i)<s.v_.cui(i))   return 1;
         if (v_.cui(i)>s.v_.cui(i))   return -1;
      }
      return 0;
   }
}

bool Ky::prnOn(ostream& str)const
{
   if (!CpmRoot::writeTitle("Ky",str)) return false;
   cpmp(s_);
   for (Z i=0;i<s_;i++) cpmp(v_.cui(i));
   return true;
}
```

```
bool Ky::scanFrom(istream& str)
{
// reading a Ky with s_=0 will be used as a stopping trigger
   Z mL=3;
   static Word loc="Ky::scanFrom(istream)";
   CPM_MA
   cpms(s_);
   if (s_<0 || s_>CpmArrays::dimMax){
      cpmmessage(loc&": s_ found as "&cpm(s_));
      return false;
   }
   v_=Vl<Z>(s_);
   // here s_==0 causes no trouble
   for (Z i=0;i<s_;i++) cpms(v_.cui(i));
   CPM_MZ
   return true;
}

Z Ky::sup()const
{
   if (s_<1) return -1;
   Z res=v_.cui(0);
   for (Z i=1;i<s_;i++){
      Z vi=v_.cui(i);
      if (vi>res) res=vi;
   }
   return res;
}

Z Ky::inf()const
{
   if (s_<1) return -1;
   Z res=v_.cui(0);
   for (Z i=1;i<s_;i++){
      Z vi=v_.cui(i);
      if (vi<res) res=vi;
   }
   return res;
}

void Ky::sup_(Ky const& k)
{
   if (s_!=k.s_) cpmerror("dimension mismatch in Ky::sup_(Ky)");
   for (Z i=0;i<s_;++i){
      Z n1=v_.cui(i);
      Z n2=k.v_.cui(i);
      v_.cui(i)=CpmRootX::sup<Z>(n1,n2);
   }
}
```

```
bool Ky::up_(Z i)
{
   Z n=v_.cui(i);
   if (n<lim_){
      v_.cui(i)+=1;
      return true;
   }
   else return false;
}

bool Ky::down_(Z i)
{
   Z n=v_.cui(i);
   if (n>0){
      v_.cui(i)-=1;
      return true;
   }
   else{
      return false;
   }
}

R Ky::sigma(Z i)const
{
   Z sum=0;
   for (Z k=0;k<i;++k) sum+=v_.cui(k);
   return CpmRootX::parity(sum);
}

R Ky::up(Z i)const
{
   Z ni=v_.cui(i);
   if (eps_==-1)
      return ::sqrt(1.+ni);
   else{
      if (ni>1) cpmerror("ni>1: not a fermion state");
      return sigma(i)*(1-ni);
   }
}

R Ky::down(Z i)const
{
   Z ni=v_.cui(i);
   if (eps_==-1){
      return ::sqrt(R(ni));
   }
   else{
      if (ni>1) cpmerror("ni>1: not a fermion state");
      return sigma(i)*ni;
```

```
   }
}

/******************* class FieldState *********************************/

bool FieldState::prnOn(ostream& str)const
{
   if (!CpmRoot::writeTitle("FieldState",str)) return false;
   cpmp(s_);
   cpmp(lim_);
   cpmp(pbc_);
#if !defined(CPM_M)
   std::map<Ky,C>::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i){
      cpmp(i->first);
      cpmp(i->second);
   }
   cpmp(Ky(0));
   C zClosing;
   cpmp(zClosing);
#else
   cpmp(m_);
#endif
 //  cpmp(C()); // closing entry that will not be read
   // back from stream
   return true;
}

bool FieldState::scanFrom(istream& str)
{
   Z mL=3;
   static Word loc="FieldState::scanFrom(istream)";
   CPM_MA
   cpms(s_);
   cpms(lim_);
   cpms(pbc_);
#if defined(CPM_M)
   cpms(m_);
   CPM_MZ
   return true;
#else
   m_=Mkc();
 //  bool active=true;
   bool res=true;
   Ky k;
   C val;
   while (true){
      bool succ=k.scanFrom(str);
      if (!succ){
         cpmwarning("can't read a Ky!");
```

```
         res=false;
         break;
      }
      succ=val.scanFrom(str);
      if (!succ){
         cpmwarning("can't read a C!");
         res=false;
         break;
      }
      if (k.dim()==0){
         cpmmessage(mL,loc&": stopping trigger found");
         break;
         // then the function should return true since
         // one is open to read the next FieldState from file
      }
      m_[k]=val;
         // only done if k and val were found as valid
         // and k was not a stopping trigger
   }
   CPM_MZ
   return res;
#endif
}

FieldState::FieldState(Z s, Z kind, Z lim, bool pbc):s_(s>0 ? s : 1),
   lim_(lim),pbc_(pbc),m_()
{
   Ky::setLim(lim_);
   if (kind==1) m_[Ky(s_)]=C::one;
}

FieldState::FieldState(State const& st, Z lim, bool pbc):s_(st.dim()),
   lim_(lim),pbc_(pbc)
{
   Ky::setLim(lim_);
   Ky k0(s_);
   for (Z i=0;i<s_;++i){
      Ky k(k0);
      k.up_(i);
      m_[k]=st[i];
   }
}

FieldState::FieldState(V<State> const& vs, Z lim, bool pbc):
   lim_(lim),pbc_(pbc)
{
   Z mL=3;
   Word loc("FieldState::FieldState(V<State> const& vs)");
   CPM_MA
   Ky::setLim(lim_);
```

```
   Z d=vs.dim();
   Z s=(d>0 ? vs[1].dim() : 0);
   FieldState fs(s,1,lim,pbc_); // vacuum state
   for ( Z i=1;i<=d;++i) fs=fs.crt(vs[i]);
   fs.nor_();
   s_=fs.size();
   m_=fs.m_;
   CPM_MZ
}

C FieldState::get(Ky const& k)const
// notice that {return m_[k];} would not be allowed
// for a const function get
{
#if !defined(CPM_M)
   std::map<Ky,C>::const_iterator i;
   i=m_.find(k);
   return i!=m_.end() ? i->second : C();
#else
   return m_[k];
#endif
}

// notice the beautyful symmetry and structure in the
// implementation of crt and des.

FieldState FieldState::crt(Z j0)const
{
   Z j=j0;
   b_1(j);
   Ky::setLim(lim_);
   Mkc res;
#if !defined(CPM_M)
   std::map<Ky,C>::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i){
      Ky k=i->first;
      Ky kj(k);
      if (kj.up_(j)) res[kj]=(i->second)*k.up(j);
   }
#else
   for (Z i=m_.b();i<=m_.e();++i){
      Ky k=m_.x(i);
      Ky kj(k);
      if (kj.up_(j)) res[kj]=m_.y(i)*k.up(j);
   }
#endif
   return mod(res);
}

FieldState FieldState::crt(State const& st)const
```

```
{
   Ky::setLim(lim_);
   Mkc res;
#if !defined(CPM_M)
   std::map<Ky,C>::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i){
      Ky k=i->first;
      for (Z j=0;j<s_;++j){
         Ky kj(k);
         if (kj.up_(j)) res[kj]+=(i->second)*k.up(j)*st(j);
            // notice that all indexes of st are defined (by
            // periodic continuation)
      }
   }
#else
   for (Z i=m_.b();i<=m_.e();++i){
      Ky k=m_.x(i);
      for (Z j=0;j<s_;++j){
         Ky kj(k);
         if (kj.up_(j)) res[kj]+=m_.y(i)*k.up(j)*st(j);
            // notice that all indexes of st are defined (by
            // periodic continuation)
      }
   }
#endif
   return mod(res);
}

/********* less efficient, but shows the logic nicely
FieldState FieldState::crt(State const& st)const
{
   FieldState res(s_);
   for (Z i=0;i<s_;++i) res+=crt(i)*st(i);
   return res;
}
*/

FieldState FieldState::des(Z j0)const
{
   Z j=j0;
   b_1(j);
   Ky::setLim(lim_);
   Mkc res;
#if !defined(CPM_M)
   std::map<Ky,C>::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i){
      Ky k=i->first;
      Ky kj(k);
      if (kj.down_(j)) res[kj]=(i->second)*k.down(j);
   }
```

```
#else
   for (Z i=m_.b();i<=m_.e();++i){
      Ky k=m_.x(i);
      Ky kj(k);
      if (kj.down_(j)) res[kj]=m_.y(i)*k.down(j);
   }
#endif
   return mod(res);
}

/***************** less efficient, but shows the logic nicely
FieldState FieldState::des(State const& st)const
{
   FieldState res(s_);
   for (Z i=0;i<s_;++i) res+=des(i)*st(i).con();
   return res;
}
*/

FieldState FieldState::des(State const& st)const
{
   Ky::setLim(lim_);
   Mkc res;
#if !defined(CPM_M)
   std::map<Ky,C>::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i){
      Ky k=i->first;
      for (Z j=0;j<s_;++j){
         Ky kj(k);
         if (kj.down_(j)) res[kj]+=(i->second)*k.down(j)*st(j).con();
      }
   }
#else
   for (Z i=m_.b();i<=m_.e();++i){
      Ky k=m_.x(i);
      for (Z j=0;j<s_;++j){
         Ky kj(k);
         if (kj.down_(j)) res[kj]+=m_.y(i)*k.down(j)*st(j).con();
      }
   }
#endif
   return mod(res);
}

FieldState FieldState::cook1(Observable const& ob)const
{
   Z mL=3;
   Word loc("FieldState::cook1(Observable)");
   CPM_MA
   FieldState res=zero();
```

```
   Z i,j;
   for (i=0;i<s_;++i)
     for (j=0;j<s_;++j)
        res+=des(i).crt(j)*ob(i,j);
   CPM_MZ
   return res;
}

FieldState FieldState::cook2(Observable const& ob)const
{
   Z mL=3;
   Word loc("FieldState::cook2(Observable)");
   CPM_MA
   FieldState res=zero();
   Z i,j;
   for (i=0;i<s_;++i)
     for (j=0;j<s_;++j)
       res+=des(j).des(i).crt(i).crt(j)*(ob(i,j)*0.5);
   CPM_MZ
   return res;
}

FieldState FieldState::manParHam(R2_Func const& pot1,
   R2_Func const& pot2, R t)const
{
   Z mL=3;
   Word loc("FieldState::manParHam(R2_Func,R2_Func,R,bool)");
   CPM_MA
   static R hD=QMBase::get_hDirac();
   static R hD2=hD*hD;
   Z i,j,d=s_;
   R dxNorm=1./d;
   R dxInv2=d/QMBase::getSize(); // (1/dx)^2, dx=size/d
   dxInv2*=dxInv2;
   R fac=-hD2*dxInv2/(2*QMBase::getMass());
      // since the - is here, we have to take the second derivative and
      // n o t  its negative. So we take the coefficients directly from
      // Bronstein ... p. 768 Tabelle 7.13
   FieldState res=zero();
   Vl<R> x0=posNor(d);
   for (i=0;i<d;++i){
      FieldState fsi=des(i);
      fsi=(fsi.crt(i-1)+fsi.crt(i+1)-fsi.crt(i)*2.)*fac;
      res+=fsi;
   }
   // now res=Kinetic energy operator applied to *this.
   // What follows are the potential contributions.
   for (i=0;i<d;++i){
      R xi=x0[i];
      C xit(xi,t);
```

```
      R pxit=pot1(xit);
      if (pxit!=0) res+=des(i).crt(i)*pxit; // efficiency !
      for (j=0;j<d;++j){
         R xij=xi-x0[j];
         if (xij>0.5)  xij-=1;
         else if (xij<-0.5) xij+=1;
         else ;
         C xijt(xij,t);
         R pxijt=pot2(xijt)*0.5;
         if (pxijt!=0) res+=des(j).des(i).crt(i).crt(j)*pxijt;
            // efficiency !
      }
   }
   CPM_MZ
   return res;
}

/*
FieldState FieldState::hub(R a, R b, R t)const
{
   Z mL=3;
   Word loc("FieldState::hub(R,R,R)");
   CPM_MA
   FieldState res(s_,0,lim_);
   FieldState psi=*this;
   for (Z i=0;i<s_;++i){
      FieldState fi=des(i);
      res+=des(i-1).crt(i);
      res+=fi.crt(i+1);
      res+=des(i+1).crt(i);
      res+=fi.crt(i-1);
      res-=fi.crt(i)*4;
   }
   CPM_MZ
   return res;
}
*/
/******************* works for fermions and bosons why? *****
FieldState FieldState::hub(R a, R b, R t)const
{
   Z mL=3;
   Word loc("FieldState::hub(R,R,R)");
   CPM_MA
   FieldState res(s_,0,lim_);
   Z i;
   for (i=0;i<s_;++i){
      FieldState fsi=des(i);
      fsi=(fsi.crt(i-1)+fsi.crt(i+1)-fsi.crt(i)*2.);
      res+=fsi;
   }
```

```
   // now res=Kinetic energy operator applied to *this.
   // What follows are the potential contributions.
   CPM_MZ
   return res;
}
*/

/**********************/
FieldState FieldState::hub(R a, R b, R t)const
{
   Z mL=3;
   Word loc("FieldState::hub(R,R,R)");
   CPM_MA
   t; // no role so far
   FieldState res=zero();
   for (Z i=0;i<s_;++i){
      Z i1=i+1;
      FieldState f1i=des(i1).crt(i);
      FieldState f2i=des(i).crt(i1);
      res+=(f1i+f2i)*a;
      res+=des(i).des(i).crt(i).crt(i)*b;
   }
   CPM_MZ
   return res;
}

FieldState FieldState::reduce(Z d)const
{
   Mkc res;
#if !defined(CPM_M)
   std::map<Ky,C>::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i){
      Ky k=i->first;
      Z max=k.sup();
      if (max<=d) res[k]=i->second;
   }
#else
   for (Z i=m_.b();i<=m_.e();++i){
      Ky k=m_.x(i);
      Z max=k.sup();
      if (max<=d) res[k]=m_.y(i);
   }
#endif
   return mod(res);
}

C FieldState::operator[](Z j)const
{
   C res;
#if !defined(CPM_M)
```

```
   std::map<Ky,C>::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i){
      Ky k=i->first;
      if (k[j]>0) res+=i->second;
   }
#else
   for (Z i=m_.b();i<=m_.e();++i){
      Ky k=m_.x(i);
      if (k[j]>0) res+=m_.y(i);
   }
#endif
   return res;
}

/*
State FieldState::toState(bool ignorePhases)const
{
   State res(s_);
   std::map<Ky,C>::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i){
      Ky k=i->first;
      C val=i->second;
      for (Z j=0;j<s_;++j){
         if (ignorePhases){
            if (k[j]>0) res[j]+=val.absSqr();
         }
         else{
            if (k[j]>0) res[j]+=val;
         }
      }
   }
   return res;
}
*/

State FieldState::toState(bool ignorePhases)const
{
   ignorePhases;
   return den();
}

State FieldState::den()const
{
   State res(s_);
   for (Z i=0;i<s_;++i){
      FieldState fsi=des(i);
      res[i]=C(fsi.absSqr());
   }
   return res;
}
```

```
Observable FieldState::toObservable(bool ignorePhases)const
{
   FieldState vac(s_,1,lim_);
   Observable res(s_);
   for (Z i=0;i<s_;++i){
      for (Z j=0;j<s_;++j){
         FieldState sij=des(i).des(j);
         C z=(vac|sij);
         if (ignorePhases) z=C(z.absVal());
         res.setLean(i,j,z);
      }
   }
   return res;
}

X3<Ky,Z1,R1> FieldState::dgn()const
{
   Z1 res;
   Ky ks(s_);
#if !defined(CPM_M)
   std::map<Ky,C>::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i){
      Ky k=i->first;
      ks.sup_(k);
      res+=1;
   }
#else
   for (Z i=m_.b();i<=m_.e();++i){
      Ky k=m_.x(i);
      ks.sup_(k);
      res+=1;
   }
#endif
   R1 n(norm());
   return X3<Ky,Z1,R1>(ks,res,n);
}

C FieldState::operator|(FieldState const& fs)const
// see Musser,Saini: STL Tutorial and Reference Guide,
// Addison-Wesley 1996, Example 7-6, p. 169,170
// for the pattern of this implementation.
{
   C res;
#if !defined(CPM_M)
   std::map<Ky,C>::const_iterator i,i2;
   for (i=m_.begin();i!=m_.end();++i){
      Ky k=i->first;
      i2=fs.m_.find(k);
      if (i2!=fs.m_.end()){
```

```
         C z=i->second;
         C z2=i2->second;
         res+=z.con()*z2;
      }
   }
#else
   for (Z i=m_.b();i<=m_.e();++i){
      Ky k=m_.x(i);
      if (fs.m_.defined(k)){
         C z=m_.y(i);
         C z2=fs[k];
         res+=z.con()*z2;
      }
   }
#endif
   return res;
}

FieldState FieldState::operator+(FieldState const& fs)const
{

   Mkc res(m_);
#if !defined(CPM_M)
   std::map<Ky,C>::const_iterator i;
   for (i=fs.m_.begin();i!=fs.m_.end();++i){
      Ky k=i->first;
      res[k]+=(i->second);
   }
#else
   for (Z i=fs.m_.b();i<=fs.m_.e();++i){
      Ky k=fs.m_.x(i);
      res[k]+=fs.m_.y(i);
   }
#endif
   return mod(res);
}

FieldState FieldState::operator-(FieldState const& fs)const
{
   Mkc res(m_);
#if !defined(CPM_M)
   std::map<Ky,C>::const_iterator i;
   for (i=fs.m_.begin();i!=fs.m_.end();++i){
      Ky k=i->first;
      res[k]-=(i->second);
   }
#else
   for (Z i=fs.m_.b();i<=fs.m_.e();++i){
      Ky k=fs.m_.x(i);
      res[k]-=fs.m_.y(i);
```

```
   }
#endif
   return mod(res);
}

FieldState FieldState::operator*(C z)const
{
   Mkc res(m_);
#if !defined(CPM_M)
   std::map<Ky,C>::iterator i;
   for (i=res.begin();i!=res.end();++i) (i->second)*=z;
#else
   for (Z i=res.b();i<=res.e();++i) res[m_.x(i)]*=z;
#endif
   return mod(res);
}

FieldState FieldState::operator*(R r)const
{
   Z mL=3;
   Word loc("FieldState::operator*(R r="&cpm(r)&")");
   CPM_MA
   Mkc res(m_);
#if !defined(CPM_M)
   std::map<Ky,C>::iterator i;
   for (i=res.begin();i!=res.end();++i) (i->second)*=r;
#else
   for (Z i=res.b();i<=res.e();++i) res[m_.x(i)]*=r;
#endif
   CPM_MZ
   return mod(res);
}

FieldState& FieldState::operator+=(FieldState const& fs)
{
   return *this=operator+(fs);
}

FieldState& FieldState::operator-=(FieldState const& fs)
{
   return *this=operator-(fs);
}

FieldState& FieldState::operator*=(C z)
{
   return *this=operator*(z);
}

FieldState& FieldState::operator*=(R r)
{
```

```
   return *this=operator*(r);
}

R FieldState::sup(void)const
{
   R res=0;
#if !defined(CPM_M)
   std::map<Ky,C>::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i){
      R val=(i->second).absVal();
      if (val>res) res=val;
   }
#else
   for (Z i=m_.b();i<=m_.e();++i){
      R val=m_.y(i).absVal();
      if (val>res) res=val;
   }
#endif
   return res;
}

FieldState FieldState::con(void)const
{
   Mkc res;
#if !defined(CPM_M)
   std::map<Ky,C>::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i){
      Ky k=i->first;
      res[k]=(i->second).con();
   }
#else
   for (Z i=m_.b();i<=m_.e();++i){
      Ky k=m_.x(i);
      res[k]=m_.y(i).con();
   }
#endif
   return mod(res);
}

FieldState FieldState::operator-(void)const
{
   Mkc res;
#if !defined(CPM_M)
   std::map<Ky,C>::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i){
      Ky k=i->first;
      res[k]=-(i->second);
   }
#else
   for (Z i=m_.b();i<=m_.e();++i){
```

```
      Ky k=m_.x(i);
      res[k]=-m_.y(i);
   }
#endif
   return mod(res);
}

R FieldState::nor_(R r)
{
   Z d=dim();
   if (d==0) return 0.;

   R r0=norm();
   if (r0==0){
      FieldState aux=vac();
         // ground state
      aux*=r;
      *this=aux;
   }
   else{
      *this*=(r/r0);
   }
   return r0;
}

void FieldState::show(Frame& f, Z mult_i, Z mult_j,
      Z iShift, Z jShift, bool ignorePhases)const
{
   toState(ignorePhases).show(f,mult_i,mult_j,iShift,jShift,ignorePhases);
}

void FieldState::mark(Graph& f, Z ipolMeth, R gamma, R secFac)const
{
   Vl<FieldState> vf(s_);
   Z i,j;
   for (i=0;i<s_;++i){
      vf[i]=des(i);
   }
   State s(s_);
   State2 s2(s,s);
   for (i=0;i<s_;++i){
      for (j=i;j<s_;++j){
         C cij=(vf[i]|vf[j]);
         s2.put_(i,j,cij);
         if (i!=j) s2.put_(j,i,cij.con());
      }
   }
   s2.mark(f,ipolMeth,gamma,secFac);
}
```

```
R FieldState::test(Z com)
{
   Z mL=1;
   Word loc("FieldState::test(Z)");
   CPM_MA
   R tol=1e-12;
   Z d=com;
   State s0(d);
   Z i;
   Z lim=1;
   R errTot=0;
back:
   cpmmessage(loc&": doing test for lim="&cpm(lim));
   State s1=s0.ranVal();
   V<State> v2(2);
   for (i=v2.b();i<=v2.e();++i) v2[i]=s0.ranVal();
   V<State> v3(3);
   for (i=v3.b();i<=v3.e();++i) v3[i]=s0.ranVal();
   FieldState fs1(s1,lim);
   FieldState fs2(v2,lim);
   FieldState fs3(v3,lim);
   V<R> c(6);
   for (i=c.b();i<=c.e();++i) c[i]=CpmRoot::randomR();

   FieldState sr=fs1*c[1]+fs2*c[2]+fs3*c[3];
   FieldState sl=fs1*c[4]+fs2*c[5]+fs3*c[6];
   R nr=sr.absVal();
   R nl=sl.absVal();
   cpmmessage(loc&": nr="&cpm(nr)&", nl="&cpm(nl));
   R res=0;
   R val=0;
   for (i=0;i<d;++i){
      FieldState pr=sr.des(i);
      FieldState pl=sl.crt(i);
      C c1=(sl|pr);
      C c2=(pl|sr);
      val+=c1.absVal();
      res+=(c1-c2).absVal();
   }
   errTot+=res;
   cpmmessage(loc&": error sum was "&cpm(res)&
      ", value sum was "&cpm(val));
   if (lim==1){
      lim+=3;
      goto back; // same for bosons
   }
   CPM_MZ
   if (errTot>tol) cpmerror(loc&": total error="&cpm(errTot));
   return errTot;
}
```

71 cpmfile.h

```
//? cpmfile.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_FILE_H_
#define CPM_FILE_H_
/*

Description:
   Making a class File that satisfies the value interface
   along using smart pointers in close analogy to F<X,Y>
   Assignments between File instances
   don't yet behave as the strict value interface requires
   (not a very natural requirement for files)
   Saying
   File f1(name);
   File f2=f1;
   cout<<f1.readLine();
   cout<<f2.readLine();
   will output the first and the second line of f1 and not twice
   the first line as would be the requirement from the
   strict value interface.
   This effect comes from hidden data in files: the 'access-mark'
   is not accessible to me, so I cant make the copy of a file
   that also copies the position of this mark.
   More important:
      File f1(name);
      File f2(name);
      cout<<f1.readLine();
      cout<<f2.readLine();
   will output the first and the second line twice
   since f1 and f2 are completely independent, although
   they are fed by the same data on file.

  However one hardly will need files in a context that would
  require the strict value interface.

  The main benfit from the exercise so far is that we can
  use data of type File in classes in a manner that no additional
  default initialisations and destructors have to be given.
  Also the effective initialization has not to be done in a constructor
  initialization list (where the information for proper
  initialization may not yet be available) but when needed
  e.g.
         file=IFile("prog.ini");
  where file is a data member of the class the constructor of
  which we are just writing
```

```
  class CpmEP..::Extractor uses this style so far.

  All definitions are short and inline, thus no cpmfile.cpp
  exists.

*/
#include <cpmv.h>

namespace CpmSystem{

   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmRoot::Word;
   using CpmArrays::P;
   using CpmArrays::V;

//////////////////////// class IFile ////////////////////////

   class IFile:public P<IFileStream>{ // input file
      Word name;
      typedef P<IFileStream> Base;
   public:
      Word nameOf()const{ return "IFile";}
      IFile(){}
      explicit IFile(const Word& w, bool safe=true, bool binary=true):
         Base(new IFileStream(w,safe,binary)){}
      ifstream& operator()(void){ return p->operator()();}
      bool isVal()const{return p->isVal();}
         //: is valid
      char readChar(){return p->readChar();}
      string readWord(){ return p->readWord();}
      string readLine(){ return p->readLine();}
      Word getName()const{ return name;}
      bool getChar(char& ch)
         { p->operator()().get(ch); return p->isVal();}
   };

//////////////////////// class OFile ////////////////////////

   class OFile:public P<OFileStream>{ // output file
      Word name;
      typedef P<OFileStream> Base;
   public:
      Word nameOf()const{ return "OFile";}
      OFile(){}
      explicit OFile(const Word& w, bool safe=true, bool binary=true):
         Base(new OFileStream(w,safe,binary)){}
      ofstream& operator()(void){ return p->operator()();}
      bool isVal()const{ return p->isVal();}
         //: is valid
```

```
      Word getName()const{ return name;}
      bool putChar(char& ch)
         { p->operator()().put(ch); return p->isVal();}
   };

} // namespace

#endif
```

72 cpmfl.h

```
//? cpmfl.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_FL_H_
#define CPM_FL_H_
/*
Purpose: Define class F<X,Y> describing 'functions' from
   X to Y. Later a derived class Fa<X,Y> will define arithmetic
   operations. This description also has in mind this class.
   The classes have default constructor, copy constructor, and assignment
   operator. Thus these 'functions' behave like the most conventional
   arithmetic quantities (they have value semantics, moreover they
   satisfy the strict value interface, see preamble of cpmv.h)

   Example:

      F<R,R> f1(cos);
      F<R,R> f2(sin);
      F<R,R> f3=f1&f2;
      F<R,R> f4=f1+f2;
      R x=1.2345;
      R eps1=f3(x)-sin(cos(x));
      R eps2=f4(x)-sin(x)-cos(x);

   will result in eps1=eps2=0. I.e. the concatenation of functions sin()
   and cos() can be created as a F object simply by a binary operator
   acting on the F objects corresponding to sin() and cos() (notice the
   order convention for & which is not intuitive to everybody).
   F<>-instances are allowed as arguments and return values of functions:

      F<C,C> power3(const F<C,C>& g)
      {
         return g*g*g;
      }

   creates the third power of a complex function, i.e.
      power3(g)(x)=g(x)*g(x)*g(x).

   In a nutshell, these new functions behave just like normal variables.
   They can also serve as data members in class definitions. This is
   important in physics and engineering applications where e.g.
   electrical fields now can be directly represented as functions instead
   of discrete value samples. They also can form components of vectors.

   For instance, a family of functions consisting of the first 100
   Chebyshev polynomials
         T0(x),....T100(x)
```

can be defined by verbally coding the (numerically not optimal) definition

```
      Tn(x):=cos(n*arccos(x)):
```

```
Vl<Fa<R,R> > Chebyshev(100);
Fa<R,R> ACos(acos);
Fa<R,R> Cos(cos);
for (int i=0; i<=n; i++) Chebyshev[i]=(ACos*i)&Cos ;
```

The value of the 27-th Chebyshev polynomial for x=0.5 then will be written as Chebyshev[27](0.5)

In defining new classes, data members of type F<X,Y> are allowed and for the classes defined this way the default copy constructor and default assignment operator are OK.

Compared to all other classes that I dealt with F<X,Y> shows the particularity that it is not possible to implement CPM_IO and CPM_ORDER (see cpminterfaces.h) in a useful manner. Later Fr<X,Y> tries to do the best one can in this respect.

Remarks on 'functions as values':

Consider

```
   R sum=0;
   for (Z i=1;i<=1000;++i) sum+=(result of some complicated
      computation with i);
   R x=sum;
```

and the similiar construct

```
   F<R,R> f;
   for (Z i=1;i<=1000;++i) f+=(Sin*i); // (*)
   F<R,R> g=f;
```

There is a remarkable difference between the two:
Whenever x will be used in subsequent parts of the program it is simply a number and the potential pain of getting to it is forgotten. On the contrary, when using g by evaluating it for specific values of the argument, e.g.

```
   R y=g(137)-g(1./137);
```

each evaluation has to go through the huge expression which g is defined in (*) to be. What was achieved by defining g compactly was to compose the code defining the Sin*i into the code defining g. But this code was composed verbatim without any intent (and chance) of simplification or compactification. So the normal idea associated with evaluation is misleading in this case.
In a time-stepping simulation one could be tempted

```
   to replace in each step a data member F<X,X> f of the evolving system
   by f&g with some transformation g characteristic for the step under
   consideration. This would make f more expensive to compute with each
   step (actually we would run into stack overflow sometime).
   Here the right way is to parameterize the mappings and express their
   composition in terms of a computation done with these parameters.
   Sophus Lie has forseen the importance of this kind of computational
   treatment of the combination of transformations. With group theory,
   Lie groups of transformations, and their linear representations, there
   has a huge body of knowledge grown on this topic.

*/

#include <cpmuc.h>
   // for reference counting handle template CpmArrays::P<X>
#include <cpmword.h>

namespace CpmFunctions{

   using CpmRoot::Z; // for order relation
   using CpmRoot::Word;
   using CpmArrays::P;

//////////////////////////////// class FncObj<> ////////////////////////

template <class X, class Y>
class  FncObj{ // function objects
   // FncObj stands for function object
   // A class which is derived from some FncObj<X,Y> is said to be a
   // function class and all instances of function classes are called
   // function objects. A function object can do everything (and more)
   // than a function can do. Unlike a function, a function class can be
   // defined within a function bloc. Let G be a function class and g an
   // object belonging to it. The value of g at x can be written as
   // g(x) (see. operator()). If in deriving G from
   // FncObj<X,Y>, no further non-static data members are added (then
   // G is said to be a pure function class), G has just one instance
   // g; this is selected by a statement as simple as
   //      G g;
   // Despite this close relationship between function objects and their
   // classes, there is an important syntactic difference between these
   // entities. As already mentioned, function values are accessed via
   // instances, e.g. Y y=g(x). Function arithmetics can be implemented
   // via template classes, among the arguments of which are
   // function classes. We will use such templates only as a transitory
   // step in implementing value-like function objects in F<X,Y>,
   // Fa<X,Y>.

   // Notice that FncObj<X,Y> is an abstract class. This is a usefull
   // prevention against invalid attempts to re-define operator() (e.g.
```

```
   // by forgetting the final 'const' attribute.
   //
   // Example: function class describing the sin function:
   //
   //    class Sin: public FncObj<R,R>{
   //       public:
   //       R operator()(const R& x)const{ return sin(x);}
   //    };
   // classes derived from FncObj may implement a more flexible and
   // effective ievaluation scheme than mere function: certain
   // preparatory
   // actions (e.g. computation of auxiliary quantities that are held
   // as data members) can be done by constructor code, so that
   // evaluation
   // of operator()(X const&) gets simpler.

   typedef FncObj<X,Y> Type;
   CPM_INVAR(FncObj)
      // makes sure that no client class will use FncObj in a way that
      // makes use of copy or assignment

public:

   FncObj(void){}

   virtual ~FncObj(void){}

   virtual   Y operator()(X const& x)const=0;

};

namespace aux{
// prevention against polluting namespace CpmFunctions
// The classes to be defined in this namespace aux are not intended
// for direct usage. They will be used internally for building
// classes which have a more expressive interface: the
// classes F<>,F1<>, .... F6<>

///////////////// ParFncObj ////////////////////////////////////////////

template <class X, class Y>
class ParFncObj : public FncObj<X,Y>{

   Y (* const fp)(X);
public:
   Y operator()(X const& x)const{ return (*fp)(x);}
   ParFncObj( Y (*g)(X)):fp(g){}
};

///////////////// Par0FncObj ///////////////////////////////////////////
```

```
template <class X, class Y>
class Par0FncObj : public FncObj<X,Y>{

   Y (* const fp)(X const&);

public:
   Y operator()(X const& x)const{ return (*fp)(x);}
   Par0FncObj( Y (*g)(X const&)):fp(g){}
};

///////////////// Par1FncObj ////////////////////////////////////////////
template <class X, class P1, class Y>
class Par1FncObj : public FncObj<X,Y>{

   const P1 p;

   Y (* const fp)(X const&, P1 const& );

public:
   Y operator()(X const& x)const{ return (*fp)(x,p);}
   P1 const& getP(void)const{return p();} // experiment
   Par1FncObj( Y (*g)(X const&, P1 const&), P1 const& pIn):fp(g),p(pIn){}
};

template <class X, class P1, class P2, class Y>
class Par2FncObj : public FncObj<X,Y>{

   const P1 p1;
   const P2 p2;
   Y (* const fp)(X const&, P1 const&, P2 const& );

public:
   Y operator()(X const& x)const{ return (*fp)(x,p1,p2);}
   Par2FncObj(Y (*g)(X const&, P1 const&, P2 const&), P1 const& p1I,
      P2 const& p2I)
   :fp(g),p1(p1I),p2(p2I){}
};

template <class X, class P1, class P2, class P3, class Y>
class Par3FncObj : public FncObj<X,Y>{

   const P1 p1;
   const P2 p2;
   const P3 p3;
   Y (* const fp)(X const&,P1 const&,P2 const&,P3 const&);

public:
   Y operator()(X const& x)const{ return (*fp)(x,p1,p2,p3);}
   Par3FncObj(Y (*g)(X const&,P1 const&,P2 const&,P3 const&),
   P1 const& q1, P2 const& q2, P3 const& q3)
```

```
   :fp(g),p1(q1),p2(q2),p3(q3){}
};

template <class X, class P1, class P2, class P3, class P4, class Y>
class Par4FncObj : public FncObj<X,Y>{

   const P1 p1;
   const P2 p2;
   const P3 p3;
   const P4 p4;
   Y (* const fp)(X const&, P1 const&, P2 const& ,P3 const& ,P4 const&);

public:
   Y operator()(X const& x)const{ return (*fp)(x,p1,p2,p3,p4);}
   Par4FncObj(Y (*g)(X const&,P1 const&,P2 const&,P3 const&,P4 const&),
   P1 const& q1,P2 const& q2,P3 const& q3,P4 const& q4)
   :fp(g),p1(q1),p2(q2),p3(q3),p4(q4){}
};

template <class X1, class X2, class X3> class comp;

} // aux

//////////////////////////// class F<X,Y> ////////////////////////////////

// Defining a template class implementing the value interface out of the
// 'poor' class FncObj<X,Y> via the handle template CpmArrays::P<>

// No assumption on X
// Y has to define Y()

template <class X, class Y>
class F: public P< FncObj<X,Y> >{ // functions as a class
   // F stands for function

   typedef F<X,Y> Type;

   typedef P< FncObj<X,Y> > base;

   static Y f_const(X const& x, const Y& y0){x; return y0;}
   static Y f_default(X const& x){ x; return Y();}

public:
   virtual F<X,Y>* clone()const{ return new F(*this);}
   F<X,Y> toClnBase()const{ return *this;}
   CPM_ORDER
      // order is here a simple bookkeeping device
      // (based on position in memory space not on
      // function values, so no order relations in X
      // or Y are needed. This is sufficient for the set
```

```
      // template  S< F<X,Y> > to work.
      // Operations that deal with order and equality in Y
      // will be introduced in Fo and Fr.

// destructor, and assignment inherited from P<...>
// constructors
   F(FncObj<X,Y>* fop):base(fop){}
      // constructor from pointers to FncObj's. All deeper constructions
      // utilize this way. F<X,Y>(new DC(...)) where DC is a class
      // derived from FncObj<X,Y> and (...) are suitable constructor
      // arguments, is thus a correct construction of a F<X,Y> object.
      // The class DC should be immutable, i.e. all its
      // data are declared const
      // That this is also directly possible for classes derived from
      // F<X,Y> it is essential to have this as automatic conversion
      // i.e. without an 'explicit' in front

   F(base const& g):base(g){}
      // upcast constructor
      // conversion needed that F<X,Y> is just a new name for
      // P< FncObj<X,Y> >

   F(void):base(new aux::Par0FncObj<X,Y>(f_default)){}
      // default constructor

   explicit F(Y const& y1):base(
      new aux::Par1FncObj<X,Y,Y>(f_const,y1)){}
      // constructor for constant function

// Construction from function pointers

   explicit F( Y (*f)(X const& )):
   base(new aux::Par0FncObj<X,Y>(f)){}

   explicit F( Y (*f)(X)):
   base(new aux::ParFncObj<X,Y>(f)){}

   //F<X,Y>& operator=( F<X,Y> const& f){ return *this=f;}
      // should not be needed
   F<X,Y>& operator=( Y (*f)(X) ){ return *this=F<X,Y>(f);}
      // added 2005-04-16. Now we can write also
      // F<R,R> sinCpm; sinCpm=sin;
      // instead of F<R,R> sinCpm(sin);
      // The = syntax is particularly convenient in arrays:
      // V< F<R,R> > fList(3);
      // fList[1]=sin; fList[2]=cos; fList[3]=exp;
      // Beware of F<R,R> sinCpm=sin; which can't be valid in one's
      // right mind
   F<X,Y>& operator=( Y (*f)(X const&)){ return *this=F<X,Y>(f);}
      // see previous function
```

```
// enabling 'function value taking' via '()' for the underlying FncObj
   virtual Y operator()(X const& x)const
   {
      return (base::operator()())(x);
      // this complicated-looking syntax is the price for using
      // operator()(void) for 'de-referencing' in 'pointer'-class P<>,
      // since the function to be defined itself is operator(), a fully
      // scope resolved name is to be used for an expression which in a
      // different scope could be simply (*this)()(x)
   }

   Y evl(X const& x)const{ return (this->base::operator()())(x);}
      // evl stands for evaluation

// turning *this into a constant-valued approximation (the most simple
// easy-to-evaluate approximating function)

   void makeConstant(X const& x0){ Y y0=(*this)(x0);*this=F<X,Y>(y0);}
      // the modified object is a constant-valued function taking
      // everywhere the value y0, which the un-modified one takes at
      // x=x0.

// concatenation (composition) of functions.

   template< class T>
   F<X,T> operator&(F<Y,T> const& g)const
      // x|-->y=(*this)(x)|-->t=g(y)=g((*this)(x))
      // thus (f&g)(x)=g(f(x))
      // notice that f & g : x |--> f(x) |--> g(f(x))
      // first action by f, second action by g thus n o t
      // (f&g)(x) = f(g(x))    wrong  !!!!
      // but
      // (f&g)(x) = g(f(x))     right !!!!
      // If one would write x.f for f(x) one had
      // x.(f&g) = x.f.g which looks much better
      // In the literature one finds f;g for f&g
   { return F<X,T>(new aux::comp<X,Y,T>(*this,g));}

   F<X,Y>& operator&=(F<Y,Y> const& g)
   { *this=operator&(g); return *this;}
      // mutating form of &, but type should not change, thus
      // T == Y needed

   template< class T>
   F<T,Y> circ(F<T,X> const& g)const
      // t|-->x=g(t)|-->y=(*this)(x)=(*this)(g(t))
      // thus (f.circ(g))(x)=f(g(x))
      // \circ is the LATEX name for the usual function
      // composition symbol '°'
```

```
   { return F<T,Y>(new aux::comp<T,X,Y>(g,*this));}


   F<X,Y>& circ_(F<X,X> const& g){ *this=circ(g); return *this;}
      // mutating form of circ, but type should not change, thus
      // T == X needed

   void leftCombine(F<X,X> const& fL){ *this=fL&(*this);}
      // *this is changed into fL & *this

   void rightCombine(F<Y,Y> const& fR){ *this=operator&(fR);}
      // *this is changed into *this & fR

   virtual Word nameOf()const
   {
      Word wi="F<";
      X x;Y y;
      Word wx=CpmRoot::Name<X>()(x);
      Word wy=CpmRoot::Name<Y>()(y);
      return wi&wx&","&wy&">";
   }
};

template <class X, class Y>
Z F<X,Y>::com(F<X,Y> const& f)const
{
   Z a=P< FncObj<X,Y> >::loc();
   Z b=f.loc();
   if (a<b) return 1;
   else if (a>b) return -1;
   else return 0;
}

namespace aux{

template <class X1, class X2, class X3>
class comp: public FncObj<X1,X3>{
   const F<X1,X2> f1;
   const F<X2,X3> f2;
public:
   comp(const F<X1,X2>& f1_, const F<X2,X3>& f2_):f1(f1_),f2(f2_){}
   X3 operator ()(const X1& x1)const{ return f2(f1(x1));}
};

} // aux

//////////////////////// class F1<> ////////////////////////////////////////

template <class X, class P1, class Y>
class F1{ // functions with one parameter
```

```
    const P1 p1;
    typedef F1<X,P1,Y> Type;
    CPM_INVAR(F1)
public:
explicit F1( P1 const& p):p1(p){}
    F<X,Y> operator()( Y (*f)(X const&, P1 const&) )const;
};

// We explain the usage of F2<X,P1,P2,Y> which is representative
// for F1,F2,F3,..., F6 (see cpmf.h for F5 and F6).
// F2<X,P1,P2,Y> is a tool for constructing instances of
// F<X,Y> in a flexible manner from classical C-functions
// of prototype Y (*f)(X const&, P1 const&, P2 const&). The
// syntax is as follows:
// Assume definitions
// P1 p1(...);
// P2 p2(...);
// Y fc(X const& x, P1 const& p1, P2 const& p2){....}
// Then
// F<X,Y> f=F2<X,P1,P2,Y>(p1,p2)(fc);
// ( or F<X,Y> f(F2<X,P1,P2,Y>(p1,p2)(fc); )
// defines a function f for which
// f(x)==fc(x,p1,p2)
// Thus f is 'the function fc with the parameters of types
// P1 and P2 given fixed values'.
// If the fixed values p1, p2 of these parameters are to be used for
// several classical functions of identical prototype,
// say fc, gc, hc, the the following syntax is economic:
// F2<X,P1,P2,Y> f2(p1,p2);
// F<X,Y> f=f2(fc);
// F<X,Y> g=f2(gc);
// F<X,Y> h=f2(hc);

template <class X, class P1, class Y>
F<X,Y> F1<X,P1,Y>::operator()( Y (*f)(X const&, P1 const&) )const
{
   return F<X,Y>(new aux::Par1FncObj<X,P1,Y>(f,p1));
}

//////////////////////// class F2<> ////////////////////////////////////////

template <class X, class P1, class P2, class Y>
class F2{ // functions with two parameters
    const P1 p1;
    const P2 p2;
    typedef F2<X,P1,P2,Y> Type;
    CPM_INVAR(F2)
public:
    F2( P1 const& p1_, P2 const& p2_):p1(p1_),p2(p2_){}
    F<X,Y> operator()(Y (*f)(X const&, P1 const&, P2 const&))const;
```

```
};

template <class X, class P1, class P2, class Y>
F<X,Y> F2<X,P1,P2,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&) )const
{
   using namespace aux;
   return F<X,Y>(new Par2FncObj<X,P1,P2,Y>(f,p1,p2));
}

//////////////////////// class F3<> ////////////////////////////////////////

template <class X, class P1, class P2, class P3, class Y>
class F3{ // functions with three parameters
    const P1 p1;
    const P2 p2;
    const P3 p3;
    typedef F3<X,P1,P2,P3,Y> Type;
    CPM_INVAR(F3)
public:
    F3( P1 const& p1_, P2 const& p2_, P3 const& p3_):
    p1(p1_),p2(p2_),p3(p3_){}
    F<X,Y> operator()(Y (*f)(X const&, P1 const&, P2 const&,
       P3 const&) )const;
};

template <class X, class P1, class P2, class P3, class Y>
F<X,Y> F3<X,P1,P2,P3,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&, P3 const&) )const
{
   using namespace aux;
   return F<X,Y>(new Par3FncObj<X,P1,P2,P3,Y>(f,p1,p2,p3));
}

//////////////////////// class F4<> ////////////////////////////////////////

template <class X, class P1, class P2, class P3, class P4, class Y>
class F4{ // functions with four parameters
    const P1 p1;
    const P2 p2;
    const P3 p3;
    const P4 p4;
    typedef F4<X,P1,P2,P3,P4,Y> Type;
    CPM_INVAR(F4)
public:
    F4( P1 const& p1_, P2 const& p2_, P3 const& p3_, P4 const& p4_):
    p1(p1_),p2(p2_),p3(p3_),p4(p4_){}
    F<X,Y> operator()(Y (*f)(X const&, P1 const&,
       P2 const&, P3 const&, P4 const&))const;
};
```

```
template <class X, class P1, class P2, class P3, class P4,
     class Y>

F<X,Y> F4<X,P1,P2,P3,P4,Y>::operator()(Y (*f)(X const&, P1 const&,
   P2 const&, P3 const&, P4 const&))const
{
   using namespace aux;
   return
   F<X,Y>(new Par4FncObj<X,P1,P2,P3,P4,Y>(f,p1,p2,p3,p4));
}

} // namespace

#endif
```

73 cpmfo.h

```
//? cpmfo.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_FO_H_
#define CPM_FO_H_
/*

Purpose: Define classes describing function-like objects with
   arithmetic operations

*/
#include <cpmf.h>

namespace CpmFunctions{

// Base class for classes describing binary operations

namespace auxiliary{

template <class X, class Y>
class BINOPFUO: public FncObj<X,Y>{ // BINOP stands for Binary
// Operation
protected:
   const F<X,Y>  f1;
   const F<X,Y>  f2;

public:
   BINOPFUO(  const F<X,Y> & f1In, const F<X,Y> & f2In ):
   f1(f1In),f2(f2In){}

};

// Base class for classes describing monary operations

template <class X, class Y>
class MONOPFUO: public FncObj<X,Y>{

protected:
   const F<X,Y>  f1;

public:

    MONOPFUO( const F<X,Y> &  f1In): f1(f1In){}
};


// order-related operations
```

```
// Now it will be assumed that Y is ordered ( Y < Y and Y > Y)

// class MINFUO<>

template <class X, class Y>
class MINFUO: public BINOPFUO<X,Y>{

public:
   Y operator()(const X& x)const
   {
      Y y1=f1(x);
      Y y2=f2(x);
      return (y1 < y2 ? y1 : y2);
   }

   MINFUO(const F<X,Y> & f1In, const F<X,Y> & f2In):
      BINOPFUO<X,Y>(f1In,f2In){}
};

// class MAXFUO<>

template <class X, class Y>
class MAXFUO: public BINOPFUO<X,Y>{

public:
   Y operator()(const X& x)const
   {
      Y y1=f1(x);
      Y y2=f2(x);
      return (y1 > y2 ? y1 : y2);
   }

   MAXFUO(const F<X,Y>& f1In, const F<X,Y>& f2In):
      BINOPFUO<X,Y>(f1In,f2In){}
};

// setting function values

template <class X, class Y>
class SETVALUEFUO : public FncObj<X,Y>{   // changing function values
   // assumes that '==' is defined in X

   const F<X,Y>  f1;
   const X xs;
   const Y ys;

public:
   Y operator()(const X& x)const
   {
      if (x==xs) return ys; else return f1(x);
```

```
   }

   SETVALUEFUO(const F<X,Y>& g1, const X& x1, const Y& y1):
   f1(g1),xs(x1),ys(y1){}
};

} // auxiliary

////////////////////////////// class Fo<X,Y> ////////////////////////////////

template <class X, class Y>
class Fo: public F<X,Y> { // version of F with order-related operations

   typedef Fo<X,Y> Type;

public:

   Fo(const F<X,Y>& g):F<X,Y>(g){}
      // upcast constructor

   Fo(const Fo<X,Y>& g):F<X,Y>(g){}
      // copy constructor

   Fo(void):F<X,Y>(){}
      // default constructor

   Fo(FncObj<X,Y>* fop):F<X,Y>(fop){}
      // constructor from pointers to FncObj

// Construction from function pointers

   Fo( Y (*f)(const X& )):F<X,Y>(f){}
   Fo( Y (*f)(X)):F<X,Y>(f){}

// Construtor for constant function

   explicit Fo(const Y& y1):F<X,Y>(y1){}
      // Functionality:
      // Y y=...; F<X,Y> f(y); X x=...; f(x)==y;
      // last statement always evaluates to 1, independent of x

   void setValue(const X& x0, const Y& y0);
      // non-constant member function which modifies the function value
      // at x=x0 to become y0
   virtual F<X,Y>* clone()const{ return new Fo(*this);}
   virtual Word nameOf()const
   {
      Word wi="Fo<";
      X x; Y y;
      Word wx=CpmRoot::Name<X>()(x);
```

```
      Word wy=CpmRoot::Name<Y>()(y);
      return wi&wx&","&wy&">";
   }
   CPM_ORDER
};

template <class X, class Y>
   // assumption Y < Y defined

Fo<X,Y> inf( const Fo<X,Y>& f1, const Fo<X,Y>& f2)
{
   using namespace auxiliary;
   return Fo<X,Y>(new MINFUO<X,Y>(f1,f2));
}

template <class X, class Y>
   // assumption Y < Y defined

Fo<X,Y> sup( const Fo<X,Y>& f1, const Fo<X,Y>& f2)
{
   using namespace auxiliary;
   return Fo<X,Y>(new MAXFUO<X,Y>(f1,f2));
}

// order interface consistent but trivial. No assumtions on X,Y

template <class X, class Y>
Z Fo<X,Y>::com(const Fo<X,Y>& f)const
{
   f;
   return 0;
}

} // namespace

#endif
```

74 cpmfontgeneration.h

```
//? cpmfontgeneration.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_FONTGENERATION_H_
#define CPM_FONTGENERATION_H_
/*

    Description:
      classes for extracting a consistent font from a
      set of non-uniformely scaled bitmap patterns
      available in ppm format. It is assumed, however that
      the orientation is correct so that no rotation of the
      originals is needed

*/

#include <cpmimg24.h>
#include <cpmm.h>

namespace CpmFonts{

   using namespace CpmStd;

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::L;
   using CpmRoot::Word;

   using CpmRootX::Z1;
   using CpmRootX::R1;

   using CpmArrays::Vl;
   using CpmArrays::IvZ;
   using CpmArrays::M;

   using CpmGraphics::Img24;
   using CpmGraphics::xy;
   using CpmGraphics::ColRef;

////////////////// class PrlLet ////////////////////////////////////////

class PrlLet{ // preliminary concept of a letter
   // preliminary letter, where letter means a printable symbol with
   // emphasis on its shape and proper placement in relation to a
   // 'printing box'.
   // Preliminarity refers to the meaning of h_,w_,d_ which do not yet
   // refer to a consistently defined printing box.
```

```
   Vl< Vl<IvZ> > r_; // representation
      // as an array of arrays of horizontal intervals. The main (outer)
      // index refers to the vertical direction. That there is a need for
      // a second (inner) index is clear from letters like A or B where a
      // horicontal cut results in either one or two segments.
   Z1 h_; // Total actual height of the symbol (this is
      // a shift-invariant characteristic of the symbol).
      // It is very small for e.g. the minus sign '-' and
      // large for 'Q', particularly if it is given a descending part.
   Z1 w_;
      // width of the data
   Z1 d_;
      // descender (Unterlänge), says how many pixels of the total
      // hight are to be placed below the text baseline.
      // Initialized as 0, which is convenient since most
      // letters don't descend
   L c_;
      // character, meaning of the letter. This comes into action if a
      // character string such as "x=10; y=20;" is to be translated into
      // letters.
   PrlLet(Vl< Vl<IvZ> > const& r, Z h, Z w):
      r_(r),h_(h),w_(w),c_('x'){}
      // private since internal consistency of data is not assured.
      // From h and w one can't give a meaningful setting of pw_.
      // So we let it stay as zero
   typedef PrlLet Type;
public:
   CPM_IO
   PrlLet():c_('x'){} // since all other members are of class type
      // this is OK
   PrlLet(Word const& fileName);
      // fileName is the name of a ppm-image
      // complete name whith ".ppm" at the end

   PrlLet(Img24 const& imr, ColRef sep=ColRef(Z(127),Z(127),Z(127)));
      // generation from a versatile image format that can be
      // get by transformation from ppm image files which, in turn,
      // may be made from electronic camera exposures.

   Vl<IvZ> const& operator[](Z i)const{return r_[i]; }

   PrlLet min()const; // minimize
      // returns a version of *this that has no white
      // rims arround the letter
   void setDes(Z d){ d_=d;}
      // set descender
   void setMen(L c){ c_=c;}
      // set meaning
   void setMen(Z n){ c_=(L)n;}
      // input of c by its decimal code
```

```
   void setWid(R w){ w_=(Z)w;}

   Img24 toImg24()const;
      // returns a image of type Img24 based on the data of *this
   Img24 toImg24(Z hl, Z hu, Z r)const;
      // Returns an image of height=hl+hu and a width that is larger
      // at both the left and the right side by a rim of width r.
      // In the vertical direction the image is placed such that its
      // lowest line is at hl-d_ above the lower rim of the image,
      // or, put differently, the upper line of the character is
      // line hu-(h_-d_) (counted - as is standard - from the upper
      // end)

   Z h()const{ return h_;}
   Z w()const{ return w_;}
   Z d()const{ return d_;}
   L c()const{ return c_;}
   void print(Z x, Z y, Img24& im)const;
      // prints the letter *this to im such that
      // the 'origin' of the letter is placed on (x,y)
   Word nameOf()const{ return "PrlLet";}
};

//////////////////// class PrlFont ////////////////////////////////////
class PrlFont{
// preliminary font is a list of preliminary letters
   Vl<PrlLet> l_;
      // letters
   Vl<Img24> im_;
      // properly placed maximum resolution images
   Z h_; // maximum of l_[i].h()
   Z hu_;
      // 'upper hight': maximum extend above the main line
   Z hl_;
      // 'lower hight': maximum extend below the main line
   typedef PrlFont Type;
   Z getIntNum(L c)const;
      // get internal number, returns for c an
      // index i such that l_[i].getMen()==c
   void makeReady();
      // setting all descenders and printing with
public:
   CPM_IO
   PrlFont(){}
   PrlFont( Vl<PrlLet> const& l);
   PrlFont(Word const& fileName);
      // fileName is the complete name
      // (including extension) of the file
      // from which to read the preliminary
```

```
      // font data
   Z dim()const{ return l_.dim();}
   Z h()const{ return h_;}
   Z hu()const{ return hu_;}
   Z hl()const{ return hl_;}

   const Img24& getImg(Z i)const{ return im_[i];}
   const PrlLet& getLet(Z i)const{ return l_[i];}
   L getMen(Z i)const{ return l_[i].c();}

   const PrlLet& getLet(L c)const{ return l_[getIntNum(c)];}
      // access to the letters by meaning

   static Vl<PrlLet> fromPPM();
      // It is assumed that there is a directory in which the raw
      // information for font creation is placed in the form of a
      // collection of ppm-images named in a way that lets deduce
      // from each file name an ASCII symbol which is represented
      // by the image encoded in this file. All symbols are assumed
      // to be represented according to the same (otherwise arbitrary
      // scale) so that they have to be demagnified all by a common
      // factor for creation of a scaled font. Presently the directory
      // and the file names are coded within this function and don't
      // enter as arguments.
};

//////////////////// class Let ////////////////////////////////////////

class Let{ // letter, where letter means a printable symbol
   // with emphasis on its shape and proper placement
   // in relation to a 'printing box'
   Img24 r_; // representation
   L c_;
      // character, meaning of the letter
   typedef Let Type;
public:
   CPM_IO

   Let(){} // since all members are of class type this is OK

   Let(PrlLet const& pl, Img24 const& im, R frac=0.1);
      // creates a letter from a preliminary letter and
      // a de-magnification factor (typically < 1)

   Img24 toImg24()const{ return r_;}
      // returns a image of type Img24 based on the data of *this

   Z h()const{ return r_.h();}
   Z w()const{ return r_.w();}
   L c()const{ return c_;}
```

```
   void print(Z x, Z y, Img24& im , ColRef cr=ColRef())const;
   Word nameOf()const{ return "Let";}
};

class Font{ // a font is a list of letters
   Vl<Let> l_;
      // letters
   Z1 h_;
      // uniform height of letters
   IvZ w_;
      // width range
   R1 mw_;
      // mean width
   M<L,Z> st_;
      // assigns to a letter its index in list l_
      // see function getLet
   typedef Font Type;
 public:
   CPM_IO
   Font(){}
   Font(Word const& fileName);
      // fileName is the complete name
      // (including extension) of the file
      // from which to read the final font data
      // (one file for one size !)
   Font(Word const& fileName, Z h);
      // fileName is the complete name
      // (including extension) of the file
      // from which to read the preliminary
      // font data. From these a font of
      // height of h pixels gets created
   Font(PrlFont const& pf, Z h);
      // same as previous function with the preliminary
      // font data as direct input instead of read from file
   Z dim()const{ return l_.dim();}
      // number of letters
   Z h()const{ return h_;}
      // hight of each letter box
   IvZ w()const{ return w_;}
      // minimum and maximum printing width
   R mw()const{ return mw_;}
      // mean printing width

   const Let& getLet(Z i)const{ return l_[i];}
   L c(Z i)const{ return l_[i].c();}
   const Let& getLet(L c)const{ return l_[st_[c]];}
      // access to the letters by meaning
   void show(Img24&, ColRef )const;
      // write the font *this in a line
   void wrt(Word const& w, xy& p, Img24& img,
```

```
      ColRef cr=ColRef())const;
      // write
      // p acts as a writing cursor: Consider
      // Word w1=..., w2=...; xy p=...; Img24 img=...; Font f;
      // Then
      // f.wrt(w1,p,img); f.wrt(w2,p,img);
      // lets the Word w2 be written in direct succession to w1.
      // p is the left-upper corner of the rectangular 'box'
      // associated with the first letter of w. Notice that all
      // such boxes have the same height (not width!) within
      // one font.
   Z len(Word const& w)const;
      //: length
      // returns the actual length of w, when written in font *this.
   Word nameOf()const{ return "CpmFonts::Font";}
   static Font factory(Z h);
      // creates a font of hight f from data in presently valid
      // location

   static Vl<Font> massFactory(Word const& fn=Word("fonts/cpmfont1.txt"));
      // returns a family of fonts al differing only in
      // height and which all are made from a single set
      // of letter shapes by interpolation.
      // Presently there are 6 fonts of heights
      // 10, 12, 14, 16, 18, 20 pixels
      // The function also creates a file with name fn to which
      // it writes the result.
};

}

#endif
```

75 cpmfontgeneration.cpp

```
//? cpmfontgeneration.cpp
//? Status of work 2008-10-25.
//?
#include <cpmfontgeneration.h>

using CpmFonts::PrlLet;
using CpmFonts::Let;
using CpmFonts::PrlFont;
using CpmFonts::Font;

// other helpers:

using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::Word;
using CpmRoot::L;

using CpmSystem::IFileStream;
using CpmSystem::OFileStream;

using CpmRootX::Z1;

using CpmLinAlg::R2;

using CpmArrays::IvZ;
using CpmArrays::Vl;
using CpmArrays::M;

using CpmFunctions::F;
using CpmFunctions::F1;

using CpmGraphics::Viewport;
using CpmGraphics::Img24;
using CpmGraphics::ColRef;
using CpmGraphics::rgb;
using CpmGraphics::xy;
using namespace std;

namespace{
bool printFunc(ColRef const& cr, ColRef const& sep)
{
   if (cr.r>sep.r && cr.g>sep.g && cr.b>sep.b) return false;
   else return true;
}
} // namespace

//////////////////////// class PrlLet ////////////////////////////////////
```

```
PrlLet::PrlLet(Img24 const& imr, ColRef sep)
{
   Z mL=3;
   static Word loc("PrlLet(Image<R>)");
   CPM_MA
   w_=imr.w();
   h_=imr.h();
   r_=Vl< Vl<IvZ> >(h_, Vl<IvZ>(0));
// We allow for the case that the image was taken with an
// electronic camera (with camera aligned properly) but
// sufficient resolution that allows proper bi-level
// binarization of the values.

   F<ColRef,bool> f=F1<ColRef,ColRef,bool>(sep)(printFunc);
   for (Z i=0;i<h_;++i){
       r_[i]=imr[i].valOn(f);
         // this is the new member functon Vl<>::valOn(...)
   }
   CPM_MZ
}

namespace{
bool printFunc(ColRef const& cr, R const& sep)
{
   if (cr.getR()>sep) return false; // only very dark parts
      // will be printed
   if (cr.getG()>sep) return false;
   if (cr.getB()>sep) return false;
   return true;
}
} // namespace

PrlLet::PrlLet(Word const& fn)
{
   Z mL=3;
   static Word loc("PrlLet(Word)");
   CPM_MA
   Img24 img(fn);
   if (cpmverbose>mL) img.display();
   cpmwait(4);
   w_=img.w();
   h_=img.h();
   r_=Vl< Vl<IvZ> >(h_, Vl<IvZ>(0));
// We allow for the case that the image was taken with an
// electronic camera (with camera aligned properly) but
// sufficient resolution that allows proper bi-level
// binarization of the values.
   R sep=127;
   F<ColRef,bool> f=F1<ColRef,R,bool>(sep)(printFunc);
```

```
   for (Z i=0;i<h_;++i){
       r_[i]=img[i].valOn(f);
         // this is the new member functon Vl<>::valOn(...)
   }
   CPM_MZ
}

namespace{
bool dimFunc(Z const& i){ return i>0;}
}

Vl<Z> minMax(Vl<Z> const& v)
{
   Z n=v.dim();
   if (n==0) return Vl<Z>(0);
   Z vMin=v[0];
   Z vMax=vMin;
   for (Z i=1;i<n;++i){
      Z vi=v[i];
      if (vi<vMin) vMin=vi;
      else if (vi>vMax) vMax=vi;
      else ;
   }
   Vl<Z> res(2);
   res[0]=vMin;
   res[1]=vMax;
   return res;
}

PrlLet PrlLet::min()const
{
   Z mL=2;
   static Word loc("PrlLet::min()");
   CPM_MA
   if (h_==0) return *this;
   Vl<Z> dims(h_);
   Z i;
   for (i=0;i<h_;++i) dims[i]=r_[i].dim();
   Vl<IvZ> roi=dims.valOn(F<Z,bool>(dimFunc));
   IvZ iv1=roi.first();
   Z iFirst=iv1.first();
   IvZ iv2=roi.last();
   Z iLast=iv2.last();
   Z h=1+iLast-iFirst;
   Vl< Vl<IvZ> > r(h);
   Z iRead=iFirst;
   for (i=0;i<h;++i) r[i]=r_[iRead++];
   Vl<Z> firsts(0);
   Vl<Z> lasts(0);
   for (i=0;i<h;++i){
```

```
      Vl<IvZ> ri=r[i];
      Z di=ri.dim();
      if (di>0){
         firsts&=ri.first().first();
         lasts&=ri.last().last();
      }
   }
   Vl<Z> mmf=minMax(firsts); // min is needed
   Vl<Z> mml=minMax(lasts); // max is needed
   Z f=mmf[0];
   Z l=mml[1];
   Z j,w=1+l-f;
   for (i=0;i<h;++i){
      Z di=r[i].dim();
      for (j=0;j<di;++j) r[i][j]-=f;
   }
   CPM_MZ
   return PrlLet(r,h,w);
}


Img24 PrlLet::toImg24()const
{
   Z mL=3;
   Word loc("PrlLet::toImg24()");
   CPM_MA
   rgb cw(255,255,255);
   rgb cb(0,0,0);

   Img24 res(w_,h_,cw);
   Z i,j;
   for (i=0;i<h_;++i){
      Vl<IvZ> ri=r_[i];
      Z di=ri.dim();
      for (j=0;j<di;++j){
         IvZ vij=ri[j];
         if (!vij.isVoid()){
            Z x1=vij.first();
            Z x2=vij.last();
            Z y1=i,y2=y1;
            res.line(x1,y1,x2,y2,cb);
         }
      }
   }
   CPM_MZ
   return res;
}

Img24 PrlLet::toImg24(Z hl, Z hu, Z r)const
{
```

```
   Z mL=3;
   Word loc("PrlLet::toImg24(Z,Z,Z)");
   CPM_MA
   rgb cw(255,255,255);
   rgb cb(0,0,0);
   Z wf=w_+2*r;
   Z hf=hl+hu;
   Img24 im=toImg24();
   Img24 res(wf,hf,cw);
   xy p(r, hu-h_+d_);
   res.insert(p,im);
   CPM_MZ
   return res;
}

void PrlLet::print(Z x, Z y, Img24& im)const
{
   Z i,j;
   rgb cb(0,100,0);
   Z xShift=x;
   Z yShift=y-h_+d_;
   for (i=0;i<h_;++i){
      Vl<IvZ> ri=r_[i];
      Z di=ri.dim();
      for (j=0;j<di;++j){
         IvZ vij=ri[j];
         Z x1=vij.first()+xShift;
         Z x2=vij.last()+xShift;
         Z y1=i+yShift,y2=y1;
         im.line(x1,y1,x2,y2,cb);
      }
   }
}

bool PrlLet::prnOn(ostream& str)const
{
   cpmwt("PrlLet");
   cpmp(r_);
   cpmp(h_);
   cpmp(w_);
   cpmp(d_);
   cpmp(c_);
   return true;
}

bool PrlLet::scanFrom(istream& str)
{
   cpms(r_);
   cpms(h_);
   cpms(w_);
```

```
   cpms(d_);
   cpms(c_);
   return true;
}

////////////////////////// class Let ////////////////////////////////////

Let::Let(PrlLet const& pl, Img24 const& im, R frac)
{
   Z mL=3;
   Word loc("Let(PrlLet,Img24,R)");
   CPM_MA
   c_=pl.c();
   Z hi=im.h();
   cpmassert(hi>0,loc);
   Z wi=im.w();
   hi=cpmround(hi*frac);
   wi=cpmround(wi*frac);
   r_=im.interpolate(wi,hi,1);
   CPM_MZ
}

void Let::print(Z x, Z y, Img24& im , ColRef cr)const
{
   xy p(x,y);
   im.insert(p,r_,cr);
}

bool Let::prnOn(ostream& str)const
{
   cpmwt("Let begin");
   cpmp(r_);
   cpmp(c_);
   cpmwt("Let end");
   return true;
}

bool Let::scanFrom(istream& str)
{
   cpms(r_);
   cpms(c_);
   return true;
}

////////////////////////// class PrlFont ////////////////////////////////

Vl<PrlLet> PrlFont::fromPPM()
{
#if defined(CPM_LINUX)
   Word dirBase="/windows/D/";
```

```
#else
   Word dirBase="D:/";
#endif
   Word dir=dirBase&"cpm/FontPrj/converted/";

   Z df=91;
   Vl<Word> vw(df);
   vw[0]="0";
   vw[1]="1";
   vw[2]="2";
   vw[3]="3";
   vw[4]="4";
   vw[5]="5";
   vw[6]="6";
   vw[7]="7";
   vw[8]="8";
   vw[9]="9";

   vw[10]="a";
   vw[11]="b";
   vw[12]="c";
   vw[13]="d";
   vw[14]="e";
   vw[15]="f";
   vw[16]="g";
   vw[17]="h";
   vw[18]="i";
   vw[19]="j";
   vw[20]="k";
   vw[21]="l";
   vw[22]="m";
   vw[23]="n";
   vw[24]="o";
   vw[25]="p";
   vw[26]="q";
   vw[27]="r";
   vw[28]="s";
   vw[29]="t";
   vw[30]="u";
   vw[31]="v";
   vw[32]="w";
   vw[33]="x";
   vw[34]="y";
   vw[35]="z";

   Z i,s0=26;
   for (i=36;i<62;++i) vw[i]=vw[i-s0]+"2";
      // the corresponding capital letters get a '2'
      // appended
```

```
vw[62]="20";
vw[63]="21";
vw[64]="23";
vw[65]="24";
vw[66]="25";
vw[67]="26";
vw[68]="28";
vw[69]="29";
vw[70]="2a";
vw[71]="2b";
vw[72]="2c";
vw[73]="2d";
vw[74]="2e";
vw[75]="2f";
vw[76]="3a";
vw[77]="3b";
vw[78]="3c";
vw[79]="3d";
vw[80]="3e";
vw[81]="3f";
vw[82]="40";
vw[83]="5b";
vw[84]="5c";
vw[85]="5d";
vw[86]="7b";
vw[87]="7c";
vw[88]="7d";
vw[89]="7e";
vw[90]="2ds";

Vl<L> c(df);
for (i=0;i<36;++i) c[i]=vw[i][1]; // first character of word
c[36]='A';
c[37]='B';
c[38]='C';
c[39]='D';
c[40]='E';
c[41]='F';
c[42]='G';
c[43]='H';
c[44]='I';
c[45]='J';
c[46]='K';
c[47]='L';
c[48]='M';
c[49]='N';
c[50]='O';
c[51]='P';
c[52]='Q';
c[53]='R';
```

```
c[54]='S';
c[55]='T';
c[56]='U';
c[57]='V';
c[58]='W';
c[59]='X';
c[60]='Y';
c[61]='Z';

c[62]='\x20';
c[63]='\x21';
c[64]='\x23';
c[65]='\x24';
c[66]='\x25';
c[67]='\x26';
c[68]='\x28';
c[69]='\x29';
c[70]='\x2a';
c[71]='\x2b';
c[72]='\x2c';
c[73]='\x2d';
c[74]='\x2e';
c[75]='\x2f';
c[76]='\x3a';
c[77]='\x3b';
c[78]='\x3c';
c[79]='\x3d';
c[80]='\x3e';
c[81]='\x3f';
c[82]='\x40';
c[83]='\x5b';
c[84]='\x5c';
c[85]='\x5d';
c[86]='\x7b';
c[87]='\x7c';
c[88]='\x7d';
c[89]='\x7e';
c[90]='_';
Z n=c.dim();
Vl<PrlLet> res(n);
 for (i=0;i<n;++i){
   L ci=c[i];
   PrlLet pi; // trivial
   if (ci != '\x20'){ // not the blank character
      Word fni(dir&"f"&vw[i]&".ppm");
      pi=PrlLet(fni);
      pi=pi.min();
   }
   pi.setMen(c[i]);
   res[i]=pi;
```

```
   }
   return res;
}


PrlFont::PrlFont( Vl<PrlLet> const& l):
   l_(l),im_(l.dim()),h_(0),hl_(0),hu_(0)
{
   Z mL=1;
   Word loc("PrlFont(Vl<PrlLet>)");
   CPM_MA
   makeReady();
      // now all size descriptors of the PrlLets have been set
   Z d=l_.dim(),hi,di,hui,hli;
   for (Z i=0;i<d;++i){
      hi=l_[i].h();
      di=l_[i].d();
      hli=di;
      hui=hi-di;
      if (hi>h_) h_=hi;
      if (hli>hl_) hl_=hli;
      if (hui>hu_) hu_=hui;
   }
   R r=cpmround(0.05*h_);
   for (Z i=0;i<d;++i){
      cpmmessage("starting i="&cpm(i));
      L ci=l_[i].c();
      if (ci=='\x20')
         im_[i]=Img24(l_[i].w(),(Z)(hl_+hu_),rgb(255,255,255));
      else
         im_[i]=l_[i].toImg24((Z)hl_,(Z)hu_,(Z)r);
   }
   CPM_MZ
}

PrlFont::PrlFont(Word const& fileName)
{
   IFileStream in(fileName);
   scanFrom(in());
}

bool PrlFont::prnOn(ostream& str)const
{
   cpmwt("PrlFont");
   cpmp(l_);
   cpmp(im_);
   cpmp(h_);
   cpmp(hl_);
   cpmp(hu_);
   return true;
```

```
}

bool PrlFont::scanFrom(istream& str)
{
   cpms(l_);
   cpms(im_);
   cpms(h_);
   cpms(hl_);
   cpms(hu_);
   return true;
}

void PrlFont::makeReady()
      // setting all descenders consistently
{
   Z mL=1;
   static Word loc("PrlFont::makeReady()");
   CPM_MA
   Z j=getIntNum('j');
   Z g=getIntNum('g');
   Z p=getIntNum('p');
   Z q=getIntNum('q');
   Z y=getIntNum('y');
   Z quc=getIntNum('Q');

   Z comma=getIntNum('\x2c');
   Z blank=getIntNum('\x20');
   Z minus=getIntNum('\x2d');
   Z mult=getIntNum('\x2a');
   Z semicol=getIntNum('\x3b');

   l_[j].setDes(90); // more like y
   l_[g].setDes(82);
   l_[p].setDes(80);
   l_[q].setDes(80);
   l_[y].setDes(92);
   l_[quc].setDes(74);
   l_[comma].setDes(82); // surprising but true
   l_[minus].setDes(-100);
   l_[mult].setDes(-40);
   l_[semicol].setDes(82);
   l_[blank].setWid(110); // set width of blank
   l_[getIntNum('\x24')].setDes(87); // $ otherwise much too high
   l_[getIntNum('\x28')].setDes(60); // brackets
   l_[getIntNum('\x29')].setDes(60);
   l_[getIntNum('\x5b')].setDes(60);
   l_[getIntNum('\x5d')].setDes(60);
   l_[getIntNum('\x7b')].setDes(60);
   l_[getIntNum('\x7d')].setDes(60);
   l_[getIntNum('\x7e')].setDes(-100); // tilde ~
```

```
   l_[getIntNum('\x3d')].setDes(-80); // =
   Z d=l_.dim();
   for (Z i=0;i<d;++i){
     Z wi=l_[i].w();
   }
   CPM_MZ
}

Z PrlFont::getIntNum(L c)const
{
   static bool firstrun=true;
   static M<L,Z> st; // search table
   Z d=l_.dim();
   if (firstrun){
      for (Z i=0;i<d;++i){
         PrlLet pi=l_[i];
         L ci=pi.c();
         st[ci]=i;
      }
      firstrun =false;
   }
   return st[c];
}

//////////////////////////// class Font ////////////////////////////////////

bool Font::prnOn(ostream& str)const
{
   Z mL=3;
   static Word loc("Font::prnOn(ostream)");
   CPM_MA
   cpmwt("Font");
   cpmp(l_);
   cpmp(h_);
   cpmp(w_);
   cpmp(mw_);
   cpmp(st_);
   CPM_MZ
   return true;
}

bool Font::scanFrom(istream& str)
{
   Z mL=3;
   static Word loc("Font::scanFrom(istream)");
   CPM_MA
   cpms(l_);
   cpms(h_);
   cpms(w_);
   cpms(mw_);
```

```
   cpms(st_);
   CPM_MZ
   return true;
}

Font::Font( PrlFont const& pf, Z hAim)
{
   Z mL=2;
   static Word loc("Font(PrlFont,Z)");
   cpmstatus("making font "&cpm(hAim),2);
   CPM_MA
   Z i,pd=pf.dim();
   if (pd<1) return;
   l_=Vl<Let>(pd);
   Z ph=pf.h();
   cpmassert(ph>0,loc);
   R hInv=1./ph;
   R frac=hAim*hInv;

   Z wMin,wMax;
   R mw=0;
   for (i=0;i<pd;++i){
      cpmmessage("loop: i="&cpm(i));
      PrlLet pli=pf.getLet(i);
      l_[i]=Let(pli,pf.getImg(i),frac);
      L ci=l_[i].c();
      st_[ci]=i;
      Z wi=l_[i].w();
      if (i==0){
         h_=l_[i].h(); // all li.h() are equal by construction
         wMin=wi;
         wMax=wi;
         mw=wi;
      }
      else{
         mw+=wi;
         if (wi>wMax) wMax=wi;
         if (wi<wMin) wMin=wi;
      }
   }
   w_=IvZ(wMin,wMax);
   mw_=mw/pd;
   cpmstatus("font "&cpm(hAim)&" ready ",2);
   CPM_MZ
}

Font::Font(Word const& fn)
{
   Z mL=3;
   static Word loc("Font(Word)");
```

```
   CPM_MA
   IFileStream in(fn);
   if (!in()) cpmerror("opening file "&fn&" failed");
   Font fTemp;
   bool suc=fTemp.scanFrom(in());
   if (!suc) cpmerror("could not read Font from "&fn);
   *this=fTemp;
   CPM_MZ
}

void Font::show(Img24& im , ColRef cr)const // write a font on im
{
   Z mL=2;
   static Word loc("Font::show(Img24&)");
   CPM_MA
   IvZ wf=w();
   Z hf=h();
   Z xStart=wf.inf();
   Z yStart=hf;
   Z pd=dim();
   Z sx=xStart;
   Z sy=yStart;
   Z yShift=hf+hf/2;
   Z wi=im.w()-wf.sup();
   for (Z i=0;i<pd;++i){
      Let li=l_[i];
      li.print(sx,sy,im,cr);
      sx+=li.w();
      if (sx>=wi){
         sx=xStart;
         sy+=yShift;
      }
      im.display();
   }
   CPM_MZ
}

void Font::wrt(Word const& w, xy& p, Img24& img, ColRef cr)const
{
   Z x=p.x, y=p.y;
   for (Z i=w.b();i<=w.e();++i){
      L ci=w[i];
      Let li=getLet(ci);
      li.print(x,y,img,cr);
      x+=li.w();
   }
   p=xy(x,y);
}

Z Font::len(Word const& w)const
```

```
{
   Z res=0;
   for (Z i=w.b();i<=w.e();++i){
      L ci=w[i];
      Let li=getLet(ci);
      res+=li.w();
   }
   return res;
}

Font Font::factory(Z fHeight)
{
   Z mL=2;
   Word loc("Font::factory(Z)");
   CPM_MA
   Word fLoc="fonts/fontshapes1.txt"; // relative path OK
   IFileStream in(fLoc);
   Vl<PrlLet> vpl;
   vpl.scanFrom(in());
   PrlFont pft(vpl);
   CPM_MZ
   return Font(pft,fHeight);
}

Vl<Font> Font::massFactory(Word const& fn)
{
   Z mL=1;
   Word loc1="Font::massFactory(";
   Word loc2=fn&")";
   Word loc=loc1&loc2;
   CPM_MA
   Word fLoc="fonts/fontshapes1.txt"; // relative path OK
   IFileStream in(fLoc);
   Vl<PrlLet> vpl;
   bool suc=vpl.scanFrom(in());
   if (suc) cpmmessage(mL, loc&": reading "&fLoc&" done");
   PrlFont pft(vpl);
   Z n=6;
   Vl<Font> res(n);
   Vl<Z> heights(n);
   heights[0]=10;
   heights[1]=12;
   heights[2]=14;
   heights[3]=16;
   heights[4]=18;
   heights[5]=24;
   for (Z i=0;i<n;++i) res[i]=factory(heights[i]);
   OFileStream ofs(fn);
   res.prnOn(ofs());
   CPM_MZ
```

```
   return res;
}
```

76 cpmforces.h

```
//? cpmforces.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_FORCES_H_
#define CPM_FORCES_H_
/*

   Description: classes for sources of forces and related physical
   quantities. Here the decision is made whether geometry is treated
   in 2 dimensions or three dimensions (2-dimensional case i.e.
   cpmdim2.h, cpmdim2.cpp not yet available)
   See also cpmforcesx.h
*/

#include <cpmconstphys.h>
#include <cpmfield.h>
#include <cpmproperties.h>
#include <cpmdim.h>
#include <cpmcamera.h>
#include <cpmdimdef.h>

#if defined(CPM_DIMDEF2)

#define CPM_DIM CpmDim2
#define CPM_CAMERA CpmCamera2
#define CPM_PHYSICS CpmPhysics2

namespace CPM_PHYSICS{
   #include <cpmforcesx.h>
}

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS

#endif

//////////////////////////////////////////////////////////////////////
#if defined(CPM_DIMDEF3)

#define CPM_DIM CpmDim3
#define CPM_CAMERA CpmCamera3
#define CPM_PHYSICS CpmPhysics3

namespace CPM_PHYSICS{
   #include <cpmforcesx.h>
}
```

```
#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS

#endif

#endif
```

77 cpmforces.cpp

```
//? cpmforces.cpp
//? Status of work 2008-10-25.
//?
#include <cpmforces.h>
#include <cpmimage.h>
#include <cpmproperties.h>
#include <cpmangle.h>

#if defined(CPM_DIMDEF2)

#define CPM_DIM CpmDim2
#define CPM_CAMERA CpmCamera2
#define CPM_PHYSICS CpmPhysics2

namespace CPM_PHYSICS{
   #include <cpmforcescpp.h>
}

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS

#endif

//////////////////////////////////////////////////////////////////////
#if defined(CPM_DIMDEF3)

#define CPM_DIM CpmDim3
#define CPM_CAMERA CpmCamera3
#define CPM_PHYSICS CpmPhysics3

namespace CPM_PHYSICS{
   #include <cpmforcescpp.h>
}

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS

#endif
```

78 cpmforcescpp.h

```
//? cpmforcescpp.h
//? Status of work 2008-10-25.
//?
using namespace CpmLinAlg;
using namespace CpmGraphics;
using namespace CpmRoot;
using namespace CpmFunctions;
using namespace CpmArrays;
using namespace CpmPhysics;
   // dimension-independent part
using namespace CPM_DIM;

using CPM_DIM::dimVec;
using CPM_DIM::Rdim_;
using CPM_DIM::Skeleton;

   using CpmProperties::SysControl;
   using CpmImaging::Color;
   using CpmImaging::Image;
   using CpmImaging::Palette;
   using CpmImaging::INVISIBLE;
   using CpmImaging::YELLOW;
   using CPM_CAMERA::Camera;
   using CpmProperties::transitionFunction;

namespace{
   const R mu=CpmPhysics::my0_PHYS;
   const R muInv=1./mu;
   const R kel=CpmPhysics::influence_PHYS;
   const R kmag=CpmPhysics::mag_influence_PHYS;
   const R kgrav=-CpmPhysics::G_PHYS;
   inline void regularize(R& x, R r){ x=sqrt(x*x+r*r);}
      // regularization device
}

//////////////////////// class VecBH ////////////////////////////////////////

void VecBH::ini(R3 const& a, Word const& ind)
{
   Z mL=3;
   Word loc("VecBH::ini(R3,Word)");
   CPM_MA
   if (Spc::dimension()==3){
      // both av_ and v_ have 3 components
      if (ind=="H"){ // v_ initialized correctly
         // av_ needs definition
         v_[1]=a[1];
```

```
         av_[1]=a[1]*mu;

         v_[2]=a[2];
         av_[2]=a[2]*mu;

         v_[3]=a[3];
         av_[3]=a[3]*mu;
      }
      else if (ind=="B"){
         av_[1]=a[1];
         v_[1]=a[1]*muInv;

         av_[2]=a[2];
         v_[2]=a[2]*muInv;

         av_[3]=a[3];
         v_[3]=a[3]*muInv;
      }
      else{
         cpmerror(loc&": indicator should be B or H not "&ind);
      }
   }
   else{ // then Spc::dimension()==2 and av_ has 1 component
      // and v_ two. av_ wil be initialized by a[3]
      // v_ by a[1], a[2]
      if (ind=="H"){
         v_[1]=a[1];
         v_[2]=a[2];
         av_[3]=a[3]*mu;
      }
      else if (ind=="B"){
         v_[1]=a[1]*muInv;
         v_[2]=a[2]*muInv;
         av_[3]=a[3];
      }
      else{
         cpmerror(loc&": indicator should be B or H not "&ind);
      }
   }
   CPM_MZ
}

VecBH::VecBH(const Vec& v, Word const& ind)
{
   R3 a; // all components 0, in particular a[3]
   a[1]=v[1];
   a[2]=v[2];
   if (Spc::dimension()==3) a[3]=v[3];
   ini(a,ind);
}
```

```
bool VecBH::prnOn(ostream& str)const
{
   cpmwt("VecBH");
   cpmp(av_);
   cpmp(v_);
   return true;
}

bool VecBH::scanFrom(istream& str)
{
   cpms(av_);
   cpms(v_);
   return true;
}

///////////////// class ForTor ////////////////////////////////////////

Z ForTor::com(const ForTor& a)const
{
   if (f<a.f) return 1;
   if (f>a.f) return -1;
   if (t<a.t) return 1;
   if (t>a.t) return -1;
   return 0;
}

ForTor::ForTor(const V< X2<Force,Spc> >& vx, Spc const& sp):f(),t()
{
   Z i,n=vx.dim();
   for (i=1;i<=n;i++){
      Vec vi=vx[i].c2()-sp;
      operator +=(ForTor(vx[i].c1(),vi));
   }
}

///////  class PointCharge ////////////////////////////////////////

bool PointCharge::prnOn( ostream& str )const
{
   CpmRoot::writeTitle("PointCharge",str);
   cpmp(q);
   cpmp(r);
   cpmp(x);
   return true;
}

bool PointCharge::scanFrom( istream& str )
{
   cpms(q);
```

```
   cpms(r);
   cpms(x);
   return true;
}

PointCharge PointCharge::mirror(const Plane& pl)const
{
   return PointCharge(-q,pl.reflect(x),r);
}

PointCharge PointCharge::mirror(const Plane& pl, R eps)const
{
   R qm=q*(1-eps)/(1+eps); // Jackson: Classical Electrodynamics,
      // Wiley 1962, p. 112, (4.51)
   return PointCharge(qm,pl.reflect(x),r);
}

Force PointCharge::force(VecBH const& BH, Vec const& v)const
{
   return Force(BH.cross(v)*(-q));
}

Force PointCharge::force( const PointCharge& pc )const
// this is the exact gradient of the following potential
// so that energy conservation holds for a system with forces
// and potential energy given by force and pot
{
   if (SysControl::noInteractionChargeToCharge) return Force();
   Vec xr=x-pc.x; // points from pc to *this
   R rr=xr.absVal();
   if (rr>SysControl::intactRange) return Vec();
   R Fabs=q*pc.q*kel*pow(r*r+rr*rr,R(-1.5));
   return Force(xr*Fabs);
}

Force PointCharge::forceMag(Vec const& v, PointCharge const& pc,
   Vec const& vpc )const
{
   if (SysControl::noMagneticInteractionChargeToCharge) return Force();
   VecBH vbhx=PointCharge::magneticField(x,pc,vpc);
   Vec temp=vbhx.cross(v);
   return Force(temp*(-q));
}

R PointCharge::pot( const PointCharge& pc )const
{
   if(SysControl::noInteractionChargeToCharge) return 0;
   Vec xr=x-pc.x; // points from pc to *this
   R rr=xr.absVal();
   if (rr>SysControl::intactRange) return 0;
```

```
   return q*pc.q*kel*pow(r*r+rr*rr,R(-0.5));
}

Force PointCharge::force( const V<PointCharge>& vpc )const
{
   Force f;
   Z n=vpc.dim();
   for (Z i=1;i<=n;i++) f+=force(vpc[i]);
   return f;
}

Force PointCharge::force(const Plane& pl, R eps)const
{
   if(SysControl::noInteractionChargeToWall) return Vec();
   R h=pl.dist(x);
   if (h<0) h=0.5*r;
   Vec n=pl.n();
   Spc x1=x-n*(2*h);
   R qAux=-q/(1+eps);
   R q1=(eps-1)*qAux;
   return force(PointCharge(q1,x1));
}

Force PointCharge::force(const Plane& pl, R eps, R d)const
{
   if(SysControl::noInteractionChargeToWall) return Vec();
   R h=pl.dist(x);
   if (h<0) h=0.5*r;
   Vec n=pl.n();
   Spc x1=x-n*(2*h);
   Spc x2=x1-n*(2*d);
   R qAux=-q/(1+eps);
   R q1=(eps-1)*qAux;
   R q2=2*qAux;
   return force(PointCharge(q1,x1))+force(PointCharge(q2,x2));
}

VecE PointCharge::electrostaticField(Spc const& x, const PointCharge& pc)
{
   if (pc.q==0.) return VecE();
   Vec xr=x-pc.x;
   R rr=xr.absVal();
   regularize(rr,pc.r);
   return VecE(xr*(pc.q*kel*pow(rr,-3)));
      // OK for pc.q>0, this has to point away from pc.x
      // (like the force on a positive charge probe.
}

VecBH PointCharge::magneticField(Spc const& x, const PointCharge& pc,
         const Vec& v)
```

```
{
   if (pc.q==0.) return VecBH();
   Vec xr=x-pc.x;
   R rr=xr.normalize();
   R nom=1./((rr*rr+pc.r*pc.r)*4*CpmGeo::Pi);
   Vec dI=v*pc.q;
   AxVec dH=dI.cross(xr)*nom;
   // my source: Weizel I, p. 371, (27) where I*ds  corresponds
   // to my dI
   R3 res; // all components are 0
   if (Spc::dimension()==3){

      res[1]=dH[1];
      res[2]=dH[2];
      res[3]=dH[3];

   }
   else{
      res[3]=dH[3];
   }
   return VecBH(res,"H");
}

VecE PointCharge::electrostaticField(Spc const& x,
   const V<PointCharge>& vpc)
{
   VecE res;
   Z n=vpc.dim();
   for (Z i=1;i<=n;i++) res+=electrostaticField(x,vpc[i]);
   return res;
}

R PointCharge::electrostaticPotential(Spc const& x,
   const PointCharge& pc)
   //
{
   if (pc.q==0.) return 0.;
   Vec xr=x-pc.x;
   R rr=xr.absVal();
   regularize(rr,pc.r);
   return pc.q*kel/rr;
}

R PointCharge::electrostaticPotential(Spc const& x,
         const V<PointCharge>& vpc)
{
   R res=0;
   Z n=vpc.dim();
   for (Z i=1;i<=n;i++) res+=electrostaticPotential(x,vpc[i]);
   return res;
```

```
}

FieldE PointCharge::field()const
         // returns the electric field generated by the source *this
{
   return FieldE(F1<Spc,PointCharge,VecE>(*this)(electrostaticField));
}

FieldBH PointCharge::fieldBH(const Vec& v)const
{
   return FieldBH(F2<Spc,PointCharge,Vec,VecBH>(*this,v)(magneticField));
}

ScalarField PointCharge::fieldU()const
         // returns the electric field generated by the source *this
{
   return F1<Spc,PointCharge,R>(*this)(electrostaticPotential);
}

//////////////////////// class PatchCharge ////////////////////////

namespace{//anon

R reg(R y, R r) // keep away from y==0
{
   Z sig=(y>=0 ? 1 :-1);
   return sig*sqrt(y*y+r*r);
}

bool indFunc(const Rdim_& x, const R& w, const R& depth)
{
   bool b1=cpmabs(x[1])<=w*0.5;
   bool b2=true;
   if (dimVec==3){
      b2=cpmabs(x[3])<=depth*0.5;
   }
   return b1&&b2;
}

// implementations based on the anti-derivation method
// (unbestimmtes Integral). See maintheorem.nb for methodological
// considerations on the connection of indefinite integrals
// (anti-derivations) and definite integrals over rectangles.
// Unfortunately this implementation was delayed by not
// realizing for 3 days that Mathematicas ArcTan(x,y)
// is not C's atan2(x,y) but atan2(y,x) even in the creation
// of C-code. (Solved 03-04-09)

inline void f3(R& fx, R& fy, R& fz, R x, R y, R z)
// name appendix 3 says that the first 3 arguments are of ref type
```

```
// The reason for this argument structure is that
// one needs to compute r only once
{
   R r=sqrt(x*x+y*y+z*z);
   fx=log(y+r);
   fy=log(x+r);
   fz=-atan2(x*y/z,r);
}

R f(R x, R y, R z)
{
   R fx,fy,fz;
   f3(fx,fy,fz,x,y,z);
   return x*fx+y*fy+z*fz;
}
   // f this is one of the many functions
   // AntiDer(x) AntiDer(y) (x^2+y^2+z^2)^(-1/2)
   // see my Mathematica file potential_and_field_of_rectangle.nb
   // a surprisingly simple function.
   // fx,fy,fz are the corresponding partial derivatives of f.
} //anon

B PatchCharge::makeFieldFromFieldU(false);
   // false is by far faster (and more precise) so
   // true is only acceptable for debugging.

const R facSqr=400;
   // factor 20 in distance
   // controls the decision from what distance from the patch on,
   // the patch should be replaced by a point charge

R PatchCharge::electrostaticPotential(Spc const& p,
   const PatchCharge& sc)
{
   if (sc.sigma==0.) return 0;
   R a=0.5*sc.w;
   R b=0.5*sc.depth;
   Rdim xr=sc.rs(p);
   R y=0;
   R x=xr[1];
   R z=reg(xr[2],sc.r);
   if (dimVec==3) y=-xr[3];

   R a2=a*a;
   R b2=b*b;
   R disSqr=x*x+y*y+z*z;
   R sizeSqr=a2+b2;
   if (disSqr>facSqr*sizeSqr){ // use point charge approximation
      PointCharge pc(sc.sigma*4*a*b,sc.get_x());
      return PointCharge::electrostaticPotential(p,pc);
```

```
   }
   else{
      R xp=x+a, xm=x-a, yp=y+b, ym=y-b;
      R res=f(xp,yp,z)-f(xp,ym,z)-f(xm,yp,z)+f(xm,ym,z);
      res*=influence_PHYS*sc.sigma;
      return res;
   }
}

VecE PatchCharge::electrostaticField(Spc const& p,
   const PatchCharge& sc)
// see potential_and_field_of_rectangle.nb by UM
{
   if (sc.sigma==0.) return VecE();
   R a=0.5*sc.w;
   R b=0.5*sc.depth;
   Rdim xr=sc.rs(p);
   R y=0;
   R x=xr[1];
   R z=reg(xr[2],sc.r);
   if (dimVec==3) y=-xr[3];

   R a2=a*a;
   R b2=b*b;
   R disSqr=x*x+y*y+z*z;
   R sizeSqr=a2+b2;
   if (disSqr>facSqr*sizeSqr){ // use point charge approximation
      PointCharge pc(sc.sigma*4*a*b,sc.get_x());
      return PointCharge::electrostaticField(p,pc);
   }
   if (makeFieldFromFieldU){
      Vec res=CPM_DIM::grad(p, F1<Spc,PatchCharge,R>(sc)
         (electrostaticPotential), sc.r);
         // uses fast and simplistic gradient formula
      return VecE(-res);
   }
   else{
      // direct field formula, which is numerically less complex
      // and significantly faster

      R fx1,fy1,fz1, fx2,fy2,fz2, fx3,fy3,fz3, fx4,fy4,fz4;
      R xp=x+a, xm=x-a, yp=y+b, ym=y-b; // p: plus, m: minus

      f3(fx1,fy1,fz1,xp,yp,z);
      f3(fx2,fy2,fz2,xp,ym,z);
      f3(fx3,fy3,fz3,xm,yp,z);
      f3(fx4,fy4,fz4,xm,ym,z);

      R Ex=fx1-fx2-fx3+fx4;
      R Ey=(dimVec==3 ? fy1-fy2-fy3+fy4 : 0);
```

```
      R Ez=fz1-fz2-fz3+fz4;

      Vec ex=sc.rs.getVectors()[1];
      Vec ey=(dimVec==3 ? -sc.rs.getVectors()[3] : Vec() );
      Vec ez=sc.rs.getVectors()[2];

      Vec res= (dimVec==3 ? ex*Ex+ey*Ey+ez*Ez : ex*Ex+ez*Ez);
      res*=(-influence_PHYS*sc.sigma);
      return VecE(res);
   }
}

ScalarField PatchCharge::fieldU()const
{
   return F1<Spc,PatchCharge,R>(*this)(electrostaticPotential);
}

FieldE PatchCharge::field()const
{
   return FieldE(F1<Spc,PatchCharge,VecE>(*this)(electrostaticField));
}

SubsetOfPlane PatchCharge::getPatch()const
{
   return SubsetOfPlane(rs,F2<Rdim_,R,R,bool>(w,depth)(indFunc));
}

V<Spc> PatchCharge::range()const
{
   Rdim c1,c2;
   R wh=0.5*w;
   c1[1]=-wh;
   c2[1]=wh;
   V<Spc> res(2);
   res[1]=rs(c1);
   res[2]=rs(c2);
   return res;
}

void PatchCharge::mark(const Camera& ca, Color c, bool shAx)const
// shAx = show axes
{
   R wh=0.5*w;
   R dh=0.5*depth;

   if (Spc::dimension()==2){
      Rdim p1,p2;
      p1[1]=wh;
      p2[1]=-wh;
      V<Spc> p(2); // was 4 till 2004-06-28
```

```
      p[1]=rs(p1);
      p[2]=rs(p2);
      Skeleton sk(p,true);
      ca.mark(sk,c);
   }
   else{

      Rdim p1,p2,p3,p4;

      p1[1]=wh;
      p1[3]=dh;

      p2[1]=wh;
      p2[3]=-dh;

      p3[1]=-wh;
      p3[3]=-dh;

      p4[1]=-wh;
      p4[3]=dh;

      V<Spc> p(4);

      p[1]=rs(p1);
      p[2]=rs(p2);
      p[3]=rs(p3);
      p[4]=rs(p4);

      Skeleton sk(p,true);
      sk.clsPath_();
      ca.mark(sk,c);
   }
   if (shAx) ca.mark(rs,c,wh);
}

V<PatchCharge> PatchCharge::increase(const V<Vec>& vv)const
{
   Z i,vvd=vv.dim();
   V<PatchCharge> res(vvd,*this);
   for (i=1;i<=vvd;i++) res[i]+=vv[i];
   return res;
}

bool PatchCharge::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("PatchCharge",str);
   cpmp(rs);
   cpmp(w);
   cpmp(sigma);
   cpmp(r);
```

```
   return true;
}

bool PatchCharge::scanFrom(istream& str)
{
   cpms(rs);
   cpms(w);
   cpms(sigma);
   cpms(r);
   return true;
}


R PatchCharge::test()
{
   if (dimVec==2){
      cpmerror("R PatchCharge::test() not implemented for 2D");
      return 0;
   }
   const Z iStart=3;
   const R range=13.7;
   const Z n=128;

   B makeFieldFromFieldU_mem=makeFieldFromFieldU;

   RefSys rfs;
   Z ir=iStart;
   Iv ivAngle(-180,180); // intervals are very convenient
      // input data for shaping the random generation process.
   R w1=ivAngle.ranVal(ir++);
   R w2=ivAngle.ranVal(ir++);
   R w3=ivAngle.ranVal(ir++);
   Angle alpha1(w1);
   Angle alpha2(w2);
   Angle alpha3(w3);
   AxVec a1(alpha1,1);
   AxVec a2(alpha2,2);
   AxVec a3(alpha3,3);
   Group g1(a1);
   Group g2(a2);
   Group g3(a3);
   rfs*=g1;
   rfs*=g2;
   rfs*=g3;
   Iv ivRange(-range,range);
   Vec sh;
   sh[1]=ivRange.ranVal(ir++);
   sh[2]=ivRange.ranVal(ir++);
   sh[3]=ivRange.ranVal(ir++);
   Group g4(sh);
```

```
   rfs*=g4;
   Iv ivSize(0,0.5*range);
   R width=ivSize.ranVal(ir++);
   R depth=ivSize.ranVal(ir++);
   R sigma=1.;
   R r=CpmRootX::inf<R>(width,depth)*1e-6;
   PatchCharge pc1(rfs,width,depth,sigma,r);
   V<Spc> tp(n); // tp = test points
   Z i;
   for (i=1;i<=n;i++){
      R x=ivRange.ranVal(ir++);
      R y=ivRange.ranVal(ir++);
      R z=ivRange.ranVal(ir++);
      Rdim xyz;
      xyz[1]=x;
      xyz[2]=y;
      xyz[3]=z;
      tp[i]=rfs(xyz);
   }
   R res=0;
   V<VecE> res1(n);
   makeFieldFromFieldU=false;
   FieldE f1=pc1.field();
   for (i=1;i<=n;i++){
      res1[i]=f1(tp[i]);
   }
   RefSys rs0;
// different style of creating and using the 'same' object
   PatchCharge pc2(rs0,width,depth,sigma,r);
   pc2*=g1;
   pc2*=g2;
   pc2*=g3;
   pc2*=g4;
   V<VecE> res2(n);
   makeFieldFromFieldU=true;
   for (i=1;i<=n;i++){
      res2[i]=pc2.eval(tp[i]);
   }
   for (i=1;i<=n;i++) res+=res1[i].disVal(res2[i]);
      // since the res[i] may have very large or small 'values'
      // summing up (res1[i]-res2[i]).absVal() would
      // be misleading
   res/=n;
   makeFieldFromFieldU=makeFieldFromFieldU_mem;
   ostringstream ost;
   ost<<"Class test for PatchCharge(): mean error was "<<res<<"."<<endl<<
   "   This should be well below 1e-5"<<endl<<
   "   Some data of the random patch: width="<<width<<" depth="<<depth<<
      " r="<<r<<endl<<
   "   Number of field points where the two methods of computing the\
```

```
 field were compared: "<<n<<endl;
   cpmmessage(ost);

   return res;
}
/// class ElcDipole ////////////////////////////////////////////

AxVec ElcDipole::torque(const VecE& E)const
{
   return d.cross(E);
}

bool ElcDipole::prnOn( ostream& str )const
{
   CpmRoot::writeTitle("ElcDipole",str);
   cpmp(d);
   cpmp(r);
   cpmp(x);
   return true;
}

bool ElcDipole::scanFrom( istream& str )
{
   cpms(d);
   cpms(r);
   cpms(x);
   return true;
}

ForTor ElcDipole::forceAndTorque( const F<Spc,VecE>& fE, R r)const
// number of fE-evaluations minimized
{
   if(SysControl::noInteractionElectricField ||
      SysControl::noInteractionElectricDipole) return ForTor();
   X2<R,Rdim> dd=d.polDec();
   R  dabs=dd.first();
   Rdim n=dd.second();
   Vec a=r*n;
   Spc x1=x+a;
   Vec E1=fE(x1);
   Spc x2=x-a;
   Vec E2=fE(x2);
   R Qel=dabs*CpmRoot::inv(2*r);
      // see Weizel I, p. 321, Abb 103, p. 322 first formula
   Vec F=Qel*(E1-E2);
   Vec E0=0.5*(E1+E2); // in order to avoid a third evaluation at the
   // center
   AxVec T=d.cross(E0);
   return ForTor(F,T);
}
```

```
ForTor ElcDipole::forceAndTorque( const ElcDipole& ed )const
{
   if(SysControl::noInteractionElectricDipole) return ForTor();
   Vec xr=x-ed.x;
   X2<R,Rdim> xrpd=xr.polDec();
   R  rr=xrpd.first();
   if (rr>SysControl::intactRange) return ForTor();
   Rdim n=xrpd.second();
   regularize(rr,r);
   R fac1=kel*pow(rr,-3);
   R fac2=3*fac1/rr;
   R dn=d|n;
   R dd=d|ed.d;
   R ddn=ed.d|n;
   Vec E=n*3*ddn-ed.d;
   E*=fac1;
   AxVec T=d.cross(E);
   Vec F=ed.d*dn+d*ddn+n*(dd-5*dn*ddn);
   F*=fac2;
   return ForTor(F,T);
}

VecE ElcDipole::electrostaticDipoleField(Spc const& x,
   const ElcDipole& md)
{
   if(SysControl::noInteractionElectricDipole) return VecE();
   Vec xr=x-md.x;
   X2<R,Rdim> xrpd=xr.polDec();
   R  rr=xrpd.c1();
   regularize(rr,md.r);
   Rdim n=xrpd.c2();
   R fac=kel*pow(rr,-3);
   R dn=md.d|n;
   Vec E=n*(3.*dn)-md.d;
   E*=fac;
   return VecE(E);
}

FieldE ElcDipole::field()const
   // returns the magnetic field generated by the source *this
{
   return FieldE(F1<Spc,ElcDipole,VecE>(*this)(electrostaticDipoleField));
}

R ElcDipole::electrostaticDipoleFieldU(Spc const& x,
   const ElcDipole& md)
{
   if(SysControl::noInteractionElectricDipole) return 0.;
   Vec xr=x-md.x;
```

```
   X2<R,Rdim> xrpd=xr.polDec();
   R  rr=xrpd.c1();
   regularize(rr,md.r);
   Rdim n=xrpd.c2();
   R fac=kel*pow(rr,-2);
   R dn=md.d|n;
   return fac*dn;
}

ScalarField ElcDipole::fieldU()const
         // returns the magnetic field generated by the source *this
{
   return F1<Spc,ElcDipole,R>(*this)(electrostaticDipoleFieldU);
}

///////  class MagPole ////////////////////////////////////////

Force MagPole::force( const MagPole& pc )const
{
   if(SysControl::noInteractionMagnetToMagnet) return Vec();
   Vec xr=x-pc.x; // points from pc to *this
   R rr=xr.absVal();
   if (rr>SysControl::intactRange) return Vec();
   regularize(rr,r);
   R fac=kmag*q*pc.q*pow(rr,-3); // positive if charges have equal sign
   return Force(xr*fac); // repulsive if fac is positive, OK
}

VecBH MagPole::magnetostaticField(Spc const& x, const MagPole& mp)
{
   Vec xr=x-mp.x;
   R rr=xr.absVal();
   rr=(rr<mp.r ? mp.r : rr);
   return VecBH(xr*(mp.q*kmag*pow(rr,-3)),"H");
}

FieldBH MagPole::field()const
         // returns the electric field generated by the source *this
{
   return F1<Spc,MagPole,VecBH>(*this)(magnetostaticField);
}

bool MagPole::prnOn( ostream& str )const
{
   CpmRoot::writeTitle("MagPole",str);
   cpmp(q);
   cpmp(r);
   cpmp(x);
   return true;
}
```

```
bool MagPole::scanFrom( istream& str )
{
   cpms(q);
   cpms(r);
   cpms(x);
   return true;
}

 /// class MagDipole ////////////////////////////////////////////////

AxVec MagDipole::torque(const VecBH& H)const
{
   return j.cross(H.v());
}

bool MagDipole::prnOn( ostream& str )const
{
   CpmRoot::writeTitle("MagDipole",str);
   cpmp(j);
   cpmp(r);
   cpmp(x);
   return true;
}

bool MagDipole::scanFrom( istream& str )
{
   cpms(j);
   cpms(r);
   cpms(x);
   return true;
}

ForTor MagDipole::forceAndTorque( const F<Spc,VecBH>& fH, R r)const
// number of fH-evaluations minimized
{
   if(SysControl::noInteractionMagneticField) return ForTor();
   X2<R,Rdim> jd=j.polDec();
   R  jabs=jd.c1();
   Rdim n=jd.c2();
   Vec a=r*n;
   Spc x1=x+a;
   Vec H1=fH(x1).v();
   Spc x2=x-a;
   Vec H2=fH(x2).v();
   R Qmag=jabs*CpmRoot::inv(2*r);
      // see Weizel I, p. 321, Abb 103, p. 322 first formula
   Vec F;
   if (!SysControl::noForceMagneticField) F=Qmag*(H1-H2);
      // in some situations it helps if a magnetic field only orients a
```

```
      // magnetized particle without carrying it away
   Vec H0=0.5*(H1+H2); // in order to avoid a third evaluation at the
   // center
   AxVec T=j.cross(H0);
   return ForTor(F,T);
}

ForTor MagDipole::forceAndTorque( const MagDipole& d )const
// see UM Ariton 99-4-21/4-6 und Nachtrag
{
   if(SysControl::noInteractionMagnetToMagnet) return ForTor();
      // was missing till 99-11-4
   Vec xr=x-d.x;
   X2<R,Rdim> xrpd=xr.polDec();
   R  rr=xrpd.c1();
   if (rr>SysControl::intactRange) return ForTor();
   Rdim n=xrpd.c2();
   regularize(rr,r);
   R fac1=kmag*pow(rr,-3);
   R fac2=3*fac1/rr;
   R jn=j|n;
   R jd=j|d.j;
   R djn=d.j|n;
   Vec H=n*3*djn-d.j;
   H*=fac1;
   AxVec T=j.cross(H);
   Vec F=d.j*jn+j*djn+n*(jd-5*jn*djn);
   F*=fac2;
   return ForTor(F,T);
}

VecBH MagDipole::magnetostaticDipoleField(Spc const& x,
   const MagDipole& md)
{
   if(SysControl::noInteractionMagneticField) return VecBH();
   Vec xr=x-md.x;
   X2<R,Rdim> xrpd=xr.polDec();
   R  rr=xrpd.c1();
   regularize(rr,md.r);
   Rdim n=xrpd.c2();
   R fac=kmag*pow(rr,-3);
   R jn=md.j|n;
   Vec H=n*3.*jn-md.j;
   return VecBH(H*=fac,"H");
}

R MagDipole::magnetostaticDipoleFieldU(Spc const& x,
   const MagDipole& md)
{
   if(SysControl::noInteractionMagneticField) return 0.;
```

```
   Vec xr=x-md.x;
   X2<R,Rdim> xrpd=xr.polDec();
   R  rr=xrpd.c1();
   regularize(rr,md.r);
   Rdim n=xrpd.c2();
   R fac=kmag*pow(rr,-2);
   R jn=md.j|n;
   return fac*jn;
}

FieldBH MagDipole::field()const
{
   return F1<Spc,MagDipole,VecBH>(*this)(magnetostaticDipoleField);
}

ScalarField MagDipole::fieldU()const
{
   return F1<Spc,MagDipole,R>(*this)(magnetostaticDipoleFieldU);
}

//////////////////// FieldSpace ////////////////////////////////////

bool FieldSpace::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("FieldSpace",str);
   cpmp(E_);
   cpmp(BH_);
   cpmp(G_);
   return true;
}

bool FieldSpace::scanFrom(istream& str)
{
   cpms(E_);
   cpms(BH_);
   cpms(G_);
   return true;
}

////////  class VectorField ////////////////////////////////////////////
namespace{ // anonymous namespace

FieldSpace combineEBHG(Spc const& x,
   const FieldE& fE, const FieldBH& fBH, const FieldG& fG)
{
   VecE vE=fE(x);
   VecBH vBH=fBH(x);
   VecG vG=fG(x);
   return FieldSpace(vE,vBH,vG);
}
```

```
Spc funcPeriodize(Spc const& x0, Spc const& x1, Spc const& x2)
{
   Word loc("funcPeriodize(...)");
   Vec v12=x2-x1;
   Vec v=v12;
   R l=v12.normalize();
   cpmassert(l>0,loc);
   Vec v10=x0-x1;
   R a=(v12|v10);
   R n=floor(a/l);
   return x0-v*n;
}

} // namespace

VectorField::VectorField(const FieldE& fE,
   const FieldBH& fBH, const FieldG& fG):
Base(F3<Spc,FieldE,FieldBH,FieldG,FieldSpace>(fE,fBH,fG)(combineEBHG)){}

static VecE f1(Spc const& x, const VectorField& f){return f(x).c1();}
static VecBH f2(Spc const& x, const VectorField& f){return f(x).c2();}
static VecG f3(Spc const& x, const VectorField& f){return f(x).c3();}

void ScalarField::periodize(Spc const& x1, Spc const& x2)
{
   leftCombine(F2<Spc,Spc,Spc,Spc>(x1,x2)(funcPeriodize));
}

void FieldE::periodize(Spc const& x1, Spc const& x2)
{
   leftCombine(F2<Spc,Spc,Spc,Spc>(x1,x2)(funcPeriodize));
}

FieldE VectorField::c1()const
{
   return FieldE(F1<Spc,VectorField,VecE>(*this)(f1));
}

FieldBH VectorField::c2()const
{
   return FieldBH(F1<Spc,VectorField,VecBH>(*this)(f2));
}

FieldG VectorField::c3()const
{
   return FieldG(F1<Spc,VectorField,VecG>(*this)(f3));
}

namespace{ // anonymous namespace
```

```
Vec v1(Spc const& x, const VectorField& f){return f(x).c1();}
Vec v2(Spc const& x, const VectorField& f){return f(x).c2().v();}
Vec v3(Spc const& x, const VectorField& f){return f(x).c3();}

FieldSpace fSetU(Spc const& x, const ScalarField& sf,
   const VectorField& vf)
{
   FieldSpace res=vf(x);
   Vec v=grad(sf)(x);
   VecE ve(-v);
   res.c1()=ve;
   return res;
}

FieldSpace fSetE(Spc const& x, const FieldE& fE, const VectorField& vf)
{
   FieldSpace res=vf(x);
   res.c1()=fE(x);
   return res;
}

FieldSpace fSetBH(Spc const& x, FieldBH const& fBH, VectorField const& vf)
{
   FieldSpace res=vf(x);
   res.c2()=fBH(x);
   return res;
}

FieldSpace fSetG(Spc const& x, const FieldG& fG, const VectorField& vf)
{
   FieldSpace res=vf(x);
   res.c3()=fG(x);
   return res;
}

FieldSpace faddE(Spc const& x, const FieldE& f, const VectorField& vf)
{
   FieldSpace res=vf(x);
   res.c1()+=f(x);
   return res;
}

FieldSpace faddBH(Spc const& x, const FieldBH& f, const VectorField& vf)
{
   FieldSpace res=vf(x);
   res.c2()+=f(x);
   return res;
}
```

```
FieldSpace faddG(Spc const& x, const FieldG& f, const VectorField& vf)
{
   FieldSpace res=vf(x);
   res.c3()+=f(x);
   return res;
}

} // namespace

VecField VectorField::operator[](Z i)const
{
   cpmassert(i>0 && i<4,"VecField VectorField::operator[](Z i)const");
   if (i==1) return VecField(F1<Spc,VectorField,Vec>(*this)(v1));
   if (i==2) return VecField(F1<Spc,VectorField,Vec>(*this)(v2));
   else return VecField(F1<Spc,VectorField,Vec>(*this)(v3));
}

VectorField::VectorField(const V<Spc>& x, const V<FieldSpace>& y):
Base(new CpmAlgorithms::AveragingInterpolator<Spc,FieldSpace>(x,y)){}


void VectorField::setU(const ScalarField& sf)
{
   *this=F2<Spc,ScalarField,VectorField,FieldSpace>(sf,*this)(fSetU);
}

void VectorField::setE(const FieldE& fE)
{
   *this=F2<Spc,FieldE,VectorField,FieldSpace>(fE,*this)(fSetE);
}

void VectorField::setBH(const FieldBH& fBH)
{
   *this=F2<Spc,FieldBH,VectorField,FieldSpace>(fBH,*this)(fSetBH);
}

void VectorField::setG(const FieldG& fG)
{
   *this=F2<Spc,FieldG,VectorField,FieldSpace>(fG,*this)(fSetG);
}

void VectorField::add(const FieldE& f)
{
   *this=F2<Spc,FieldE,VectorField,FieldSpace>(f,*this)(faddE);
}

void VectorField::add(const FieldBH& f)
{
   *this=F2<Spc,FieldBH,VectorField,FieldSpace>(f,*this)(faddBH);
}
```

```
void VectorField::add(const FieldG& f)
{
   *this=F2<Spc,FieldG,VectorField,FieldSpace>(f,*this)(faddG);
}

Vec gradient(Spc const& x , const ScalarField& f,const R& h)
{
   return CPM_DIM::grad(x,f,h);
}

VecField grad(const ScalarField& sf, R h)
{
   return F2<Spc,ScalarField,R,Vec>(sf,h)(gradient);
}

static Vec optTransfer(const Vec& v)
// This function will be used in ScalarField::mark() and
// VectorField::mark() only.
// It turns out that fields the order of magnitude of which varies
// strongly over the display area get visible representaton only at a few
// points if represented linearily. So, in analogy to imaging transfer
// curves I allow for a transformation of the display values too. Notice
// that one has to separate into absolute value and direction if the
// direction of the gradient is to be represented correctly.
{
   R r=v.polDec().c1();
   if (r==0) return v;
   R fac=pow(r,SysControl::fieldGamma)/r;
   return v*fac;
}

void markVecField(const VecField& f, const Camera& sc,
      R& scaleFactor, Z gridPointsHorizontal, Z gridPointsVertical,
      const Color& color)
{
   Z mL=3;
   Word loc("markVecField(...)");
   cpmmessage(mL,loc&" started, scaleFactor="&cpmwrite(scaleFactor));
   cpmassert(gridPointsHorizontal>0 && gridPointsVertical>0,loc);
// getting from gr representative points in object space
   C zLL(sc.getxLeft(),sc.getyLower());
      // lower left point of rectangle associated with gr
   R dx=sc.getdX()/gridPointsHorizontal;
   R dy=sc.getdY()/gridPointsVertical;
   C cell(dx,dy);
   R cellSize=0.5*CpmRootX::inf(dx,dy);
      // independent of its direction, the arrow should never leave
      // its cell
   C pointLL=zLL+0.5*cell;
```

```
   Z nP=gridPointsHorizontal*gridPointsVertical;
   V<Spc> spc(nP);
   Z i,j,iCounter=1;
   R pLx=pointLL[1];
   R pLy=pointLL[2];
   R spx=pLx;
   R spy=pLy;
   for (i=1;i<=gridPointsHorizontal;i++){
      for (j=1;j<=gridPointsVertical;j++){
         spc[iCounter++]=sc.inverse(C(spx,spy),1.);
         spy+=dy;
      }
      spy=pLy;
      spx+=dx;
   }
// recording field strength's at the test points

   V<Vec> fields(nP);

   for (i=1;i<=nP;i++){
      Vec fVal=f(spc[i]);
      fields[i]=optTransfer(fVal);
   }
   // if the scaleFactor is not yet determined we have to make
   // field vectors and position vectors comparable (they typically
   // don't even have the same dimension)
   R tWait=-1;
   if (scaleFactor<=0){
      tWait=2;
      Vo<R> absVal(nP);
      for (i=1;i<=nP;i++) absVal[i]=fields[i].absVal();
      R scale=absVal.sup();
      if (scale<=0){
         cpmwarning("denied to draw a trivial vector field");
         return; // nothing worthwhile to be drawn
      }
      scaleFactor=cellSize/scale;
   }
   R rHead=0.04*cellSize;
   for (i=1;i<=nP;i++){
      Spc spcEnd=spc[i]+fields[i]*scaleFactor;
      sc.mark(spc[i],spcEnd,color,rHead);
         // argument color introduced 2001-11-26
   }
   cpmwait(tWait);
   cpmmessage(mL,loc&" done");
}

R VectorField::scaleFactorE=-1;
R VectorField::scaleFactorM=-1;
```

```
R VectorField::scaleFactorG=-1;

void VectorField::mark(const Camera& sc)const
{
   if (SysControl::markElectricField>=0){
      // was > 0 till 2003-06-16 thus disabling anaglyphic field
      // representation
      VecField fE=(*this)[1];
      Z colNumber=SysControl::markElectricField;
      Color c(INVISIBLE);
      if (colNumber>0) c=Color(colNumber);
      markVecField(fE,sc,scaleFactorE,SysControl::nxElectricField,
         SysControl::nyElectricField,c);
         // SysControl::markElectricField is the color, 0 for
         // anaglyphic representation, negativ for no representation
   }
   if (SysControl::markMagneticField>=0){
         // remark as in electric case
      VecField fM=(*this)[2];
      Color c(INVISIBLE);
      Z colNumber=SysControl::markMagneticField;
      if (colNumber>0) c=Color(colNumber);
      markVecField(fM,sc,scaleFactorM,SysControl::nxMagneticField,
         SysControl::nyMagneticField,c);
   }
   if (SysControl::markGravityField>=0){
      VecField fG=(*this)[3];
      Color c(INVISIBLE);
      Z colNumber=SysControl::markGravityField;
      if (colNumber>0) c=Color(colNumber);
      markVecField(fG,sc,scaleFactorG,SysControl::nxGravityField,
         SysControl::nyGravityField,c);
   }
}

//////////////////// ScalarField ////////////////////////////////
namespace{
   R fConAux(Spc const& x, FieldSpace const& fs,
      Spc const& p)
   {
      Z pts=SysControl::potentialToShow;
      Vec fc;
      if (pts==2) fc=fs.c2().v();
      else if (pts==3) fc=(Vec)fs.c3();
      else fc=(Vec)fs.c1();
      return ((p-x)|fc);
   }
} // namespace

ScalarField::ScalarField(FieldSpace const& fs, Spc const& p):
```

```
Base(F2<Spc,FieldSpace,Spc,R>(fs,p)(fConAux)){}

void ScalarField::mark(const Camera& ca)const
{
   Z mL=2;
   Word loc("ScalarField::mark(..)");
   CPM_MA
   Response rsp=ca.getResponse();
   bool adjust=!rsp.normalizeGamma();
   Z ny=SysControl::nyScalarFields;
   cpmassert(ny>=16,loc); // warn or stop
   if (ny<16) ny=16;
   Z nx=(Z)(ny*ca.getGraph().getAspRat());
   Iv ivx=ca.getGraph().getIvX();
   Iv ivy=ca.getGraph().getIvY();
   V<R> vx=ivx.centers(nx);
   V<R> vy=ivy.centers(ny);
   cpmassert(vx.dim()==nx,loc);
   cpmassert(vy.dim()==ny,loc);
   R_Matrix val(ny,nx); // indexed like a image matrix
   Z i,j;
   C xij;
   Z count=0;
   R nxInv=1./nx;
   for (i=1;i<=nx;i++){
      xij[1]=vx[i];
      for (j=1;j<=ny;j++){
         xij[2]=vy[j];
         Spc sij=ca.inverse(xij,1.);
         val[ny-j+1][i]=operator()(sij);
      }
      count++;
      if (cpmverbose>1) cpmprogress("mark-loop",count*nxInv);
   }
   Iv riv;
   if (adjust){
      Iv ivRange=rsp.getRangeX();
      X2<R_Matrix,Iv> xri=val.adjustRange(ivRange);
      val=xri.c1();
      riv=xri.c2();
   }
   Image<R> img(val.toVV());
   if (adjust){
      R vL=riv.inf();
      R vU=riv.sup();
      if (cpmverbose>0){
         R xT=0.25, yT=0.1;
         Color cT(Z(255),Z(0),Z(0));
         Word wT=cpm(vL)&" <=val<= "&cpm(vU);
         ca.getGraph().addText(wT,xT,yT,cT);
```

```
      }
   }
   Z intPolMet=1; // bilinear
   img.mark(ca.getGraph(),rsp,intPolMet);
   CPM_MZ
}

ScalarField::ScalarField(const V<Spc>& x, const V<R>& y):
Base(new CpmAlgorithms::AveragingInterpolator<Spc,R>(x,y)){}

///////////////// class SourceOfVectorField //////////////////////////

void SourceOfVectorField::add(const PointCharge& s)
{
   R qs=s.get_q();
   q+=qs;
   d+=(s.get_x()-x)*qs;
   updateQD();
}

void SourceOfVectorField::add(const ElcDipole& s)
{
   d+=s.get_d();
   updateD();
}

void SourceOfVectorField::add(const MagDipole& s)
{
   j+=s.get_j();
   updateJ();
}

void SourceOfVectorField::add(const PointMass& s)
{
   R ms=s.get_m();
   m+=ms;
   updateM();
}

void SourceOfVectorField::add(const SourceOfVectorField& s)
{
   R qs=s.get_q();
   q+=qs;
   d+=(s.d+(s.get_x()-x)*qs);
   j+=s.j;
   m+=s.m;
   update();
}

bool SourceOfVectorField::prnOn( ostream& str )const
```

```
{
   cpmwt("SourceOfVectorField");
   cpmp(q);
   cpmp(d);
   cpmp(j);
   cpmp(m);
   cpmp(r);
   cpmp(x);
   return true;
}

bool SourceOfVectorField::scanFrom( istream& str )
{
   cpms(q);
   cpms(d);
   cpms(j);
   cpms(m);
   cpms(r);
   cpms(x);
   update();
   return true;
}

V<SourceOfVectorField> SourceOfVectorField::redistribute(
   const V<SourceOfVectorField>& sl, const V<Spc>& pl)
   // sl: source list, pl: point list
{
   Z mL=2;
   static Word loc("SourceOfVectorField::redistribute(...)");
   CPM_MA
   Z sld=sl.dim(),pld=pl.dim();
   Z i,k; // j is attribute!
   Vo<R> distances(pld);
   V<SourceOfVectorField> res(pld);
   R_Vector vr(sld);
   for (i=1;i<=sld;i++) vr[i]=sl[i].get_r();
   R rMean=vr.mean();
   for (k=1;k<=pld;k++) res[k]=SourceOfVectorField(pl[k],rMean);
   for (i=1;i<=sld;i++){
      Spc xi=sl[i].get_x();
      for (k=1;k<=pld;k++) distances[k]=(pl[k]-xi).absSqr();
         // absSqr() faster than absVal()
      Z iMin=distances.indInf();
         // indInf very efficient, no ordering
      res[iMin].add(sl[i]);
   }
   CPM_MZ
   return res;
}
```

```
V<SourceOfVectorField> SourceOfVectorField::redistribute2(
   const V<SourceOfVectorField>& sl, const V<Spc>& pl,
   const V<Z>& whereToGo)
{
   Z mL=3;
   static Word loc("SourceOfVectorField::redistribute2(...)");
   CPM_MA
   Z sld=sl.dim(),pld=pl.dim();
   Z i,k; // j is attribute!
   V<SourceOfVectorField> res(pld);
   R rMean=SysControl::singRange; // OK since not of
      // any influence in the application in cpmpardynsys.h
   for (k=1;k<=pld;k++) res[k]=SourceOfVectorField(pl[k],rMean);
      // here no index mistake can happen
   for (i=1;i<=sld;i++){
      Z wi=whereToGo[i];
      cpmassert(wi>0,loc);
      cpmassert(wi<=pld,loc);
         // in case that something went wrong with 'whereToGo'
      res[wi].add(sl[i]);
   }
   CPM_MZ
   return res;
}

///////  class PointMass ////////////////////////////////////////////

bool PointMass::prnOn( ostream& str )const
{
   CpmRoot::writeTitle("PointMass",str);
   cpmp(m);
   cpmp(r);
   cpmp(x);
   return true;
}

bool PointMass::scanFrom( istream& str )
{
   cpms(m);
   cpms(r);
   cpms(x);
   return true;
}

Force PointMass::force( const PointMass& pc )const
// exact force to the following regularized potential
{
   if(SysControl::noInteractionMassToMass) return Vec();
   Vec xr=x-pc.x; // points from pc to *this
   R rr=xr.absVal();
```

```
   if (rr>SysControl::intactRange) return Vec();
   R Fabs=kgrav*m*pc.m*pow(r*r+rr*rr,R(-1.5));
   return Force(xr*Fabs);
}

R PointMass::pot( const PointMass& pc )const
{
   if(SysControl::noInteractionMassToMass) return 0;
   Vec xr=x-pc.x; // points from pc to *this
   R rr=xr.absVal();
   if (rr>SysControl::intactRange) return 0;
   return kgrav*m*pc.m*pow(r*r+rr*rr,R(-0.5));
}

VecG PointMass::gravitationalField(Spc const& x, const PointMass& pc)
{
   Vec xr=x-pc.x;
   R rr=xr.absVal();
   rr=(rr<pc.r ? pc.r : rr);
   return VecG(xr*(pc.m*kgrav*pow(rr,-3)));
}

R PointMass::gravitationalPotential(Spc const& x, const PointMass& pc)
{
   Vec xr=x-pc.x;
   R rr=xr.absVal();
   rr=(rr<pc.r ? pc.r : rr);
   return pc.m*kgrav/rr;
}

FieldG PointMass::field()const
{
   return F1<Spc,PointMass,VecG>(*this)(gravitationalField);
}

ScalarField PointMass::fieldU()const
{
   return F1<Spc,PointMass,R>(*this)(gravitationalPotential);
}
```

79 cpmforcesx.h

```
//? cpmforcesx.h
//? Status of work 2008-10-25.
//?
/*

   Description: Classes for sources of forces and related physical
   quantities.
   The decision, whether geometry is treated in 2 dimensions
   or three dimensions is made by macro substitutions for
   CPM_DIM and CPM_CAMERA. See cpmforces.h for the mechanism which
   defines these macros and which puts the quantities to be defined in
   the present file into namespaces CpmPhysics2 or CpmPhysics3 according
   to the status of macros CPM_DIMDEF2 and CPM_DIMDEF3.
   This is not a header file for its own but cpmforce.h is.
   One finds the same mechanism for a common definition of 2D and
   3D versions of physical and geometrical concepts throughout the
   C+- class system.
   In the 2D case, sources of fields never are asumed to have an
   infinite extension in the direction perpendicular to the
   representation plane. Instead, only all vectorial quantities
   (such as velocities or directions of dipoles) are assumed to
   be parallel to this representation plane. In particular, our
   point charges are point charges generating fields decreasing as
   r^-2 (not logarithmic !) also in 2D; they thus are no charged thin
   wires.
*/

using namespace CpmRoot;
using CPM_DIM::AxVec;
using CPM_DIM::Vec;
using CPM_DIM::Spc;
using CPM_DIM::Group;
using CPM_DIM::Plane;
using CPM_DIM::SubsetOfPlane;
using CPM_DIM::SubsetOfSpace;
using CPM_DIM::RefSys;
using CPM_DIM::Rdim;
using CPM_CAMERA::Camera;

using CpmProperties::SysControl;
using CpmGraphics::Graph;
using CpmImaging::Color;
using CpmImaging::Palette;
using CpmImaging::GREEN;
using CpmRootX::B;
using CpmRootX::R1;
using CpmLinAlg::R3;
```

```
using CpmFunctions::F;
using CpmFunctions::F1;
using CpmArrays::V;
using CpmArrays::X2;

//doc namespace CpmPhysicsx{

//////////////////////// class VecE ////////////////////////////////////
// electrical field strength E (V/m in SI) (electric force/charge)
// There is no need to introduce D-Fields as united with E-fields
// in a way analogous to the magnetic case

   class VecE: public Vec{ // electrical field strength E

   public:
      VecE():Vec(){}
      explicit VecE(const Vec& a):Vec(a){}
      explicit VecE(V<R> const& vr):Vec(vr){}
      explicit VecE(Rdim const& vr):Vec(vr){}
      Word nameOf()const{ return Word("VecE");}
   };

//////////////////////// class VecG ////////////////////////////////////
// gravitational field strength (acceleration = gravity force/mass)

   class VecG: public Vec{ // gravitational field strength
   public:
      VecG():Vec(){}
      explicit VecG(const Vec& a):Vec(a){}
      explicit VecG(V<R> const& vr):Vec(vr){}
      explicit VecG(Rdim const& vr):Vec(vr){}
      Word nameOf()const{ return Word("VecG");}
   };

//////////////////////// class VecBH ////////////////////////////////////
// Magnetic induction B (V.s.m^-2 in SI) and magnetic field strength H
// (A/m in SI) in a single class.
// This has a particular behavior in 2D since we need here a field in the
//
// system plane to study chain formation of magnetized particles
// and also a perpendicular field for studying the influence of the
// Lorentz force. Thus, also in 2 dimensions, the field has 3 components.
// In 2D thus the dimension of the quantities say that H as a 2-component
// object is typed Vec, wheras the perpendicular field is typed AxVec.
// Also in 3D it is useful to have H and B available at once. But in this
// case these two are proportional to each other.
// The situation that in 2D, both av_ and v_ are different from zero
// is not strictly physical. The present interpretation of it is that
// from av_ one computes the Lorentz force and from v_ one computes
// the force and torque on magnetic dipoles.
```

```
class VecBH{//magnetic induction B and magnetic field strength H combined
      AxVec av_; // B (relative to unit Tesla)
      Vec v_; // H (relative to unit A/m)
      VecBH(AxVec const& av, Vec const& v):av_(av),v_(v){}
         // private constructor
      void ini(R3 const& a, Word const& ind);
         // compact initialization tool. All components are
         // understood as referring all to the SI unit T
         // or all to the SI unit A/m. The first case is
         // understood for ind="B", the second one for
         // ind="H".
   public:
      typedef VecBH Type;
      CPM_IO
      VecBH():av_(),v_(){}
         // zero field
      VecBH(Vec const& v, Word const& ind);
         // the components of v will be interpreted as
         // H components if ind=="H" and as B components if
         // ind=="B".
      VecBH(V<R> const& vr, Word const& ind ){ ini(R3(vr),ind);}
         // the numbers vr[i] get interpreted as having the dimension of
         // H if ind=="H" and as B components if ind=="B".
      VecBH(R3 const& r3, Word const& ind ){ ini(r3,ind);}
         // the numbers r3[i] get interpreted as having the dimension of
         // H if ind=="H" and as B components if ind=="B".
      Word nameOf()const{ return Word("VecBH");}
      Vec v()const{ return v_;}
         // H-field in umits A/m ('Ampere windings per meter')
      AxVec av()const{ return av_;}
         // B-field in units T (Tesla)
      Z dim()const{ return 3;}
      VecBH neg()const{ return VecBH(-av_,-v_);}
      VecBH operator*(R r)const{ return VecBH(av_*r,v_*r);}
      VecBH& operator-=(VecBH const& x)
         { av_-=x.av_; v_-=x.v_;  return *this;}
      VecBH& operator+=(VecBH const& x)
         { av_+=x.av_; v_+=x.v_;  return *this;}
      VecBH& operator*=(R r){ av_*=r; v_*=r;  return *this;}
      VecBH& operator*=(Group const& g){ av_*=g; v_*=g;  return *this;}
      Vec cross(Vec const& w)const{ return av_.cross(w);}
         // Defined such that the Lorentz force on a charge q of
         // velocity w is cross(w)*(-q)
   };

//////////////////////// class Force ////////////////////////////////

   struct Force{  // force
      // Example of a compact definition of an auxiliary class.
```

```
      Vec f;
      typedef Force Type;
      typedef R ScalarType;
      CPM_LINEAR
      CPM_ORDER

      Force():f(){}

      Force(const Vec& force):f(force){}
      Word nameOf()const{ return Word("Force");}

      Vec get_f()const{ return f;}

      void set_f(const Vec& f_){ f=f_;}
      void addForce(const Force& f_){ f+=f_.f;}

      Force& operator *=(const AxVec& d)
      { f*=d;  return *this;}

      Force& operator *=(const Group& g)
      { AxVec d=g.rot(); f*=d; return *this;}

      bool isVal()const{ return f.isVal();}

   };

   inline Force& Force::operator +=(const Force& a)
      { f+=a.f; return *this;}
   inline Force& Force::operator -=(const Force& a)
      { f-=a.f;  return *this;}
   inline Force& Force::operator *=(const R& a)
      { f*=a; return *this;}
   inline Force& Force::neg()
      { f.neg();  return *this;}
   inline Z Force::dim()const
      {return f.dim();}

//////////////////////// class ForTor ////////////////////////////////

   class ForTor{ // force and torque combined
      // combining force and torque into a linear Space.
      // Example of a compact definition of an auxiliary class.
      // It turns out that this class is also needed for describing
      // the combination of velocity and angular velocity
      // and also for acceleration and angular acceleration.
      // Actually it is a representation of the Lie-Algebra of Group
      Vec f;
      AxVec t;
   public:
      typedef ForTor Type;
```

```
typedef R ScalarType;
CPM_LINEAR
CPM_ORDER

ForTor():f(),t(){}

ForTor(const Force& f_, const Vec& v):
   f(f_.f),t(v.cross(f_.f)){}
   // Let the force f_ act on point pAct, then let the torque
   // refer to point pRef, then ForTor(f_,pAct-pRef) is the
   // object of type ForTor which corresponds to the situation

ForTor(const Force& force, const AxVec& torque):f(force.f),
   t(torque){}

ForTor(const V<X2<Force,Spc> >& vx, const Spc& sp);
   // combines forces, that are given together with their point of
   // action into a resulting total force and a torque with respect
   // to point sp

Word nameOf()const{ return Word("ForTor");}

void add(const Force& f_, const Spc& pAct, const Spc& pRef)
   // adds to *this the contribution of force f_ acting on point
   // pAct under the assumption that the torque refers to the
   // reference point pRef. One could think that a better notion of
   // ForTor should include the reference point for the torque
   // among the data members. Will not be carried out, though.
{ f+=f_.f; t+=(pAct-pRef).cross(f_.f);}

Vec get_force()const{ return f;}
AxVec get_torque()const{ return t;}

const Vec& c1()const{ return f;}
const AxVec& c2()const{ return t;}
   // semantically neutral access to components

void set_force(const Vec& f_){ f=f_;}
void addForce(const Force& f_){ f+=f_.f;}
void addForce(const Vec& v){ f+=v;}
void addTorque(const AxVec t_){t+=t_;}
void set_torque(const AxVec& t_){ t=t_;}

ForTor& operator *=(const AxVec& d)
{ f*=d; t=t.adjointAction(d); return *this;}
   // notice that t*=d would be a composition of rotations
   // which can't be the right thing
      // 't=' was missing till 2005-02-14

ForTor& operator *=(const Group& g)
```

```
      { AxVec d=g.rot(); f*=d; t=t.adjointAction(d); return *this;}
            // 't=' was missing till 2005-02-14

      bool isVal()const{ return f.isVal()&&t.isVal();}

   };

   inline ForTor& ForTor::operator +=(const ForTor& a)
      { f+=a.f; t+=a.t; return *this;}
   inline ForTor& ForTor::operator -=(const ForTor& a)
      { f-=a.f; t-=a.t; return *this;}
   inline ForTor& ForTor::operator *=(const R& a)
      { f*=a; t*=a; return *this;}
   inline ForTor& ForTor::neg()
      { f.neg(); t.neg(); return *this;}
   inline Z ForTor::dim()const
      {return t.dim()+f.dim();}

//////////////////////// class FieldSpace ////////////////////////////////

   class FieldSpace{
//electric, magnetic, gravitational fieldstrengths into one linear space
      VecE E_;
      VecBH BH_;
      VecG G_;
   public:
      typedef FieldSpace Type;
      typedef R ScalarType;
      CPM_LINEAR
      CPM_IO
      FieldSpace():E_(),BH_(),G_(){}
      FieldSpace(VecE const& E, VecBH const& H, VecG const& G=VecG()):
         E_(E),BH_(H),G_(G){}

      Word nameOf()const{ return Word("FieldSpace");}

// my standard notation for heterotypic components. I'll always
// use this correspondence:
// 1: electric, 2: magnetic, 3: gravitational
      VecE& c1(){ return E_;}
      const VecE& c1()const{ return E_;}
      VecBH& c2(){ return BH_;}
      const VecBH& c2()const{ return BH_;}
      VecG& c3(){ return G_;}
      const VecG& c3()const{ return G_;}
      FieldSpace& operator *=(const Group& g)
         { E_*=g; BH_*=g; G_*=g; return *this;}
      FieldSpace& operator+=(VecE const& v){ E_+=v; return *this;}
      FieldSpace& operator+=(VecBH const& v){ BH_+=v; return *this;}
      FieldSpace& operator+=(VecG const& v){ G_+=v; return *this;}
```

```
   };

   inline FieldSpace& FieldSpace::operator +=(FieldSpace const& a)
   { E_+=a.E_; BH_+=a.BH_; G_+=a.G_; return *this;}

   inline FieldSpace& FieldSpace::operator -=(FieldSpace const& a)
   { E_-=a.E_; BH_-=a.BH_; G_-=a.G_; return *this;}

   inline FieldSpace& FieldSpace::operator *=(R const& a)
   {  E_*=a; BH_*=a; G_*=a;  return *this;}

   inline FieldSpace& FieldSpace::neg()
      { E_.neg(); BH_.neg(); G_.neg(); return *this;}

   inline Z FieldSpace::dim()const{return E_.dim()+BH_.dim()+G_.dim();}

//////////////////// class ScalarField ////////////////////////////////////

class ScalarField:public CpmFields::ScalarField<Spc,R,Group>{
      // scalar field with group action
      // since we need functions that cannot yet be defined in
      // CpmFields::ScalarField<Spc,R,Group>, a typedef is not
      // sufficient.
      typedef CpmFields::ScalarField<Spc,R,Group> Base;
   public:
      ScalarField(const Base& f):Base(f){}
      ScalarField(const F<Spc,R>& f):Base(f){}

      explicit ScalarField(const SubsetOfSpace& sos):Base(sos.getF()){}
         // new constructor which mainly helps the visalization of
         // SubsetOfSpace objects via the well-working function
         // void ScalarField::mark(const Camera& sc)const;

      ScalarField(const V<Spc>& x, const V<R>& y);
         // General interpolation method for functions defined on a
         // metric
         // space and taking values in a linear space

      explicit ScalarField(FieldSpace const& fs, Spc const& p=Spc());
         // Potential of a homogeneous VectorField of (constant) strength
         // fs. The component selector c (1: electric, 2: magnetic, 3:
         // gravitation) determines the field under consideration.

      ScalarField():Base(){}

      Word nameOf()const{ return Word("ScalarField");}

      friend Base down(const ScalarField& s);//{ return s;}
         // works !!!
```

```
      void periodize(const Spc& x1, const Spc& x2);

      void mark(const Camera& sc)const;
         // Displays the scalar field under the control of the
         // data element typed Response of sc. Implemented by
         // means of image class Image<R> which supports much
         // higher dynamic range than Img24. This function
         // does not simply mark but also displays automaticly.
         // In the way I use it presently it does not cooperate well
         // with the frame number indication in movie replay.
         // This is considered a minor aesthetical problem.

      static ScalarField sum(const V<ScalarField>& vf);
    //  { return Base::sum(vf(F<ScalarField,Base>(down)));}
         // a direct implementation as in Base and avoiding .f was not
         // faster (even a bit slower) in a test with summing over the
         // field of several hundred particles.
   };

   inline CpmFields::ScalarField<Spc,R,Group>
      down(const ScalarField& s){ return s;}
         // works !!!

   inline ScalarField ScalarField::sum(const V<ScalarField>& vf)
   {
      return
      CpmFields::ScalarField<Spc,R,Group>::
         sum(vf(F<ScalarField,Base>(down)));
   }

//////////////////// class FieldE ////////////////////////////////////////

   class FieldE: public CpmFields::Field<Spc,VecE,R,Group>{
      // E-field with group action
      // since we need functions that cannot yet be defined in
      // CpmFields::Field<Spc,VecE,R,Group>, a typedef is not
      // sufficient for the electric field. Could well be
      // that a similar need would arise for the magnetic
      // field too
      typedef CpmFields::Field<Spc,VecE,R,Group> Base;
   public:
      FieldE(const Base& f):Base(f){}
      FieldE():Base(){}
      Word nameOf()const{ return Word("FieldE");}
      void periodize(const Spc& x1, const Spc& x2);
   };

//////////////////// class FieldBH ////////////////////////////////////////

   typedef CpmFields::Field<Spc,VecBH,R,Group> FieldBH;
```

```
//////////////////// class FieldG ////////////////////////////////////

   typedef CpmFields::Field<Spc,VecG,R,Group> FieldG;

/////////////////// class VecField ////////////////////////////////////

   typedef CpmFields::Field<Spc,Vec,R,Group>  VecField;

   void markVecField(const VecField& f, const Camera& sc,
      R& scaleFactor,
      Z gridPointsHorizontal=10,
      Z gridPointsVertival=6,
      const Color& color=GREEN);
   // scaleFactor is a factor which determines how long the field vectors
   // will be represented on screen. A negative value will not be
   // utilized as input but as a signal to determine a suitable value for
   // this parameter. After function call the variable scaleFactor will
   // hold a proper value that may be used in subsequent calls.

   VecField grad(const ScalarField&, R h=SysControl::singRange);
      // gradient as a vector field

   Vec gradient(const Spc& x , const ScalarField& f,
      const R& h=SysControl::singRange);
      // gradient as a vector-valued function of position

//////////// class VectorField ////////////////////////////////////////

// Multicomponent fields can be implemented as arrays of mono-component
// fields or as fields with multi-component values.
// Here we use the second method which looks more natural to me. The
// first one would presumably be slightly easier to implement.

   class VectorField:public CpmFields::Field<Spc,FieldSpace,R,Group>{
   // FieldSpace-valued field with group action

      static R scaleFactorE; // controls the graphical representation
      static R scaleFactorM;
      static R scaleFactorG;

   public:
      typedef CpmFields::Field<Spc,FieldSpace,R,Group> Base;
      VectorField(){}
      VectorField(const Base& f):Base(f){}
      VectorField(const F<Spc,FieldSpace>& f):Base(f){}
      explicit VectorField(FieldE const& fe,
         FieldBH const& fh=FieldBH(), FieldG const& fg=FieldG());
         // creating a vector field by prescribing the field components
      VectorField(V<Spc> const& x, V<FieldSpace> const& y);
```

```
   // General interpolation method for functions defined on a
   // metric
   // space and taking values in a linear space

VectorField(const FieldSpace& y):Base(y){}
   // constructor for a constant field, with y as the field value
Word nameOf()const{ return Word("VectorField");}
FieldE c1()const;
FieldBH c2()const;
FieldG c3()const;
   // components
VecE E(const Spc& x)const{ return (operator()(x)).c1();}
VecBH BH(const Spc& x)const{ return (operator()(x)).c2();}
VecG G(const Spc& x)const{ return (operator()(x)).c3();}

friend Base downV(const VectorField& s);//{ return s;}

static void newSclE(){ scaleFactorE=-1;}
static void newSclM(){ scaleFactorM=-1;}
static void newSclG(){ scaleFactorG=-1;}

static void newScl()
{ scaleFactorE=-1; scaleFactorM=-1; scaleFactorG=-1;}
   // managing graphical representation newScl = new scale
VecField operator[](Z i)const;
   // returns the electric, magnetic, and gravitational field part
   // as a quantity of uniform type. Notice that the implementation
   // in cpmforces.h would be syntactically correct also for a
   // return value VecField&. Calling this modified function causes
   // severe runtime errors (would be a reference to a local
   // variable, which my present compiler does not even warn)
void setU(const ScalarField&);
   // the electrical part of the field is set such that the
   // argument of the function is the potential of it
void setE(const FieldE&);
   // the electrical part of the field is set such that it equals
   // the argument of the function

void setBH(const FieldBH&);
   // the magnetic part of the field is set such that it equals the
   // argument of the function
void setG(const FieldG&);
   // the gravitational part of the field is set such that it
   // equals the argument of the function

void add(const FieldE&);
void add(const FieldBH&);
void add(const FieldG&);
   // adding is mostly more convenient than setting; no
   // different function names needed
```

```
      void mark(const Camera& sc)const;
         // The details of the display mode are controlled by
         // SysControl::nxElectricField
         // SysControl::nyElectricField
         // SysControl::markElectricField
         // SysControl::nxMagneticField
         // SysControl::nyMagneticField
         // SysControl::markMagneticField
         // SysControl::nxGravityField
         // SysControl::nyGravityField
         // SysControl::markGravityField

      static VectorField sum(const V<VectorField>& vf);
      //{ return Base::sum(vf(F<VectorField,Base>(downV)));}
    };

    inline CpmFields::Field<Spc,FieldSpace,R,Group>
       downV(const VectorField& s){ return s;}

   inline VectorField VectorField::sum(const V<VectorField>& vf)
   {
      return CpmFields::Field<Spc,FieldSpace,R,Group>::
         sum(vf(F<VectorField,Base>(downV)));
   }

//////////////////////// class PointMass ////////////////////////////////////

   class PointMass{ // point masses
      // describes point masses and the forces they feel and they create

      R m;
         // mass (kg)

      R r;
         // a regularization radius
         // Notice that interacting point-masses have to have all
         // the same regularization radius, otherwise momentum
         // conservation does not hold. So it is wise to use
         // SysControl::singRange as this value

      Spc x;
         // position

   public:
      typedef PointMass Type;
      CPM_IO
      Word nameOf()const{ return Word("PointMass");}
      PointMass():m(0),r(SysControl::singRange){}
         // x gets initilized as zero vectors automatically
```

```
PointMass(R m_, const Spc& x_, R r_=SysControl::singRange):
   m(m_),x(x_),r(r_){}

R get_m()const{ return m;}

void set_m(R m_){ m=m_;}

R get_r()const{ return r;}

void set_r(R r_){ r=r_;}

Spc get_x()const{ return x;}

void set_x(const Spc& x_){ x=x_;}

PointMass& operator *=(R s){m*=s; return *this;}

PointMass& operator *=(const Group& a){x*=a; return *this;}
   // action of the basic symmetry group on a point mass.

PointMass& operator *=(const AxVec& a){a; return *this;}
   // action of rotations on a point mass. The quantity
   // is scalar and so the action is trivial. Included here for
   // uniformity of the interface with similar classes

PointMass& operator +=(const Vec& a){x+=a; return *this;}
   // action of shift on a point mass.

Force force( const VecG& G )const{ return Force(m*G);}
   // force on *this acted on by an gravitational field strength G

Force force( const F<Spc,VecG>& fG)const{ return Force(m*fG(x));}
   // returns the force on the point charge as a result of the
   // interaction with the argument field, which is a vector field
   // giving the spatial distribution of the gravitational field
   // strength (G)

Force force( const PointMass& pm )const;
   // force on *this caused by a point mass pm

R pot( const PointMass& pm )const;
   // potential energy of a pair due to gravity

R potEnr(const VecG& G)const{ return (Spc()-x|G)*m;}
   // potential energy of *this with respect to the homogeneous
   // gravitational field strength G

FieldG field()const;
   // returns the gravitational field generated by the source *this
```

```
      VecG eval(const Spc& x)const
      { return gravitationalField(x,*this);}

      R evalU(const Spc& x)const
      { return gravitationalPotential(x,*this);}

      ScalarField fieldU()const;
         // returns the gravitational potential as a field

      static VecG gravitationalField(const Spc& x, const PointMass& pc);
         // the main physical law as a plain function

      static R gravitationalPotential(const Spc& x, const PointMass& pc);
         // the main physical law as a plain function
   };

////////////////// class PointCharge ////////////////////////////////

   class PointCharge{ // point charges
      // describes point charges and the forces they feel and they
      // create.
      // Actually a certain type of radially symmetric continuous
      // charge, that only for vanishing regularization radius r
      // becomes a point charge.

      R q;
         // electric charge (A*s)

      R r;
         // a regularization radius. This can be interpreted as
         // the approximate radius of a radially symmetric
         // diffuse charge distribution depending on distance
         // d like (d*d+r*r)^(-5/2)
         // see my Mathematica file coulombmod2.nb

      Spc x;
         // position

   public:
      typedef PointCharge Type;
      CPM_IO
      Word nameOf()const{ return Word("PointCharge");}
      PointCharge():q(0),r(SysControl::singRange){}
         // x gets initilized as zero vectors automatically

      PointCharge(R q_, const Spc& x_, R r_=SysControl::singRange):
         q(q_),x(x_),r(r_){}

      R get_q()const{ return q;}
```

```
void set_q(R q_){ q=q_;}

R get_r()const{ return r;}

void set_r(R r_){ r=r_;}

Spc get_x()const{ return x;}

void set_x(const Spc& x_){ x=x_;}

void invertCharge(){ q=-q;}

PointCharge mirror(const Plane& pl)const;
   // returns a PointCharge which is the mirror charge created by
   // *this in the presence of a perfectly conducting plane

PointCharge mirror(const Plane& pl, R eps)const;
   // returns a PointCharge which is the mirror charge created by
   // *this in the presence of a dielectric plane of
   // relative dielectric constant eps. The halfspace
   // where *this is located is assumed to be
   // characterized by relative dielectric constant equal
   // to 1. eps>>1 approximates the behavior of the previous
   // function.

PointCharge& operator *=(R s){q*=s; return *this;}

PointCharge& operator *=(const Group& a){x*=a; return *this;}
   // action of the basic symmetry group on a electric point
   // charge.

PointCharge& operator *=(const AxVec& a){a; return *this;}
   // action of rotations on a electric point charge. The quantity
   // is scalar and so the action is trivial. Included here for
   // uniformity of the interface with similar classes

PointCharge& operator +=(const Vec& a){x+=a; return *this;}
   // action of shift on a electric point charge.

void add(const PointCharge& pc){ q+=pc.get_q();}

Force force(VecBH const& BH, Vec const& v)const;
   // Lorentz force force on *this acted on by an magnetic field
   // vector BH. The velocity of the PointCharge is given as v.

Force force( const VecE& E )const{ return Force(q*E);}
   // force on *this acted on by an electric field strength E

Force force( const F<Spc,VecE>& fE)const{ return Force(q*fE(x));}
```

```
         // returns the force on the point charge as a result of the
         // interaction with the argument field, which is a vector field
         // giving the spatial distribution of the electric field
         // strength (E)

      Force force( PointCharge const& pc )const;
         // force on *this caused by a point charge pc

      Force forceMag(Vec const& v, PointCharge const& pc,
         Vec const& vpc )const;
         // Lorentz force on *this , moving with velocity v,
         // caused by the magnetic field,
         // generated by point charge pc, moving with velocity vpc.
         // No retardation taken into account.

      R pot( const PointCharge& pc )const;
         // potential energy of a pair due to Coulomb interaction

      R potEnr(const VecE& E)const{ return (Spc()-x|E)*q;}
         // potential energy of *this with respect by the homogeneous
         // electric field of strength E

      Force force( const V<PointCharge>& vpc )const;
         // force on *this caused by the family vpc of point charges

// Interaction with dielectric half-space and with a dielectric layer
// which is made perfectly conducting on the back side (like a photo
// conductor film in electro-photography. Explanations:

// On image charges

   // A electrophotographic photoconductor film is modelled here as a
   // dielectric layer of relative dielectric constant eps and thickness
   // d behind which there is a perfectly conducting half-space (a layer
   // of any thickness has the same effect, if conductivity is perfect).
   // A point charge q placed near the surface of the film will feel a
   // force resulting from influenced electric dipoles in the dielectric
   // layer and re-distributed charges in the conductor. It can be shown
   // (UM/PNP 99-11-17/1-4) that this force is approximately (Thomas
   // Dera, 99-12-6, pointed out that this holds not exactly as I thought
   // before. The condition on continuity of normal component of D
   // is not satisfied exactly. See my Mathematica document pnp.nb )
   // to the force exerted by a charge q1=-q*(eps-1)/(eps+1) located at
   // the mirror image position resulting from reflection at the PC
   // surface and a second charge q2=-q*2/(eps+1) located at the mirror
   // image position resulting from reflection at the surface of the
   // conducting half-space. Notice q+q1+q2=0. The combined field of
   // charge q together with the two image charges is the actual electric
   // field generated by q under the environmental condition posed by the
   // presence of the PC film. This implies that, in the presence of a PC
```

```
   // film, each particle feels from a partner particle the direct
   // Coulomb force from the partner particle and the Coulomb forces from
   // the mirror particles of the partner particle. It also feels a force
   // from 'itself' i.e. from its own mirror particles. Since in the
   // present application, the PC is treated as elastically deformable,
   // we
   // have to consider the possibility that the point charge q is placed
   // below the undisturbed PC surface. In this case we define the first
   // image charge one charge regularization radius (see PointCharge::r)
   // ahead of q

// Consider a system V<PointCharge> pcs of n point charges near a PC film
// where the latter is described by Plane pl, R eps, R d.
// Then the force felt by charge i0 is given by f from the following
// algorithm
//
//      Force f=pcs[i0].force(pl,eps,d);
//      for (i=1;i<=n;i++){
//         if (i==i0) continue;
//         f+=pcs[i0].force(pcs[i],pl,eps,d);
//      }
// It thus can be considered consistently as a sum over all particles -
// including the particle feeling the force - we thus have non-trivial
// self-interaction. The PC film appears as a parameter in the particle-
// particle force law and does not appear as an interaction partner (as
// in the erroneous Ansatz: force to a particle = force to that particle
// from the PC + force to that particle from all the other particles.

      Force force(const Plane& pl, R eps )const;
         // force on *this caused by the interaction of *this and a
         // half-space determined by the Plane pl and filled by
         // dielectric
         // matter of relative constant of dielectricity eps.

      Force force(const Plane& pl, R eps, R d)const;
         // force on *this caused by a dielectric layer (thickness d) and
         // a perfectly conducting half-space

      FieldE field()const;
         // returns the electric field generated by the source *this

      FieldBH fieldBH(const Vec& v)const;
         // returns the magnetic field generated by the source *this
         // if this moves with velocity v. No retardation is taken into
         // account

      VecE eval(const Spc& x)const
         { return electrostaticField(x,*this);}

      R evalU(const Spc& x)const
```

```
         { return electrostaticPotential(x,*this);}

      ScalarField fieldU()const;
         // returns the electric potential as a field

      static VecE electrostaticField(const Spc& x,const PointCharge& pc);
         // the main physical law as a plain function
         // see my Mathematica file coulombmod2.nb for a
         // consistent physical interpretation of the regularization
         // as a non-singular charge distribution.

      static VecBH magneticField(const Spc& x, const PointCharge& pc,
         const Vec& v);
         // the main physical law as a plain function
         // Magnetic field as generated by a moving charge
         // no retardation taken into account.

      static VecE electrostaticField(const Spc& x,
         const V<PointCharge>& vpc);
         // the main physical law as a plain function

      static R electrostaticPotential(const Spc& x,
         const PointCharge& pc);
         // the main physical law as a plain function

      static R electrostaticPotential(const Spc& x,
         const V<PointCharge>& pc);
   };

///////////////// class PatchCharge ////////////////////////////////////

   class PatchCharge{ // charged planar rectangle (patch)
      // describes a charged planar rectangle (patch) one side of which
      // is of length w and the other of length depth.
      // Till ~ 03-02-xx the quantity depth was not present as a data
      // member since it had the 'fixed value infinity'.
      // Some of the commentaries may still refer to this situation.
      // Since 03-04-09 also formula for fieldU is generalized
      // (surprisingly simple) for this general case.

      RefSys rs;
         // this is a reference frame relative to which the charged area
         // is a subset of the 1-3-plane, given by the condition
         // -w/2 <= x[1] <=w/2, -depth/2 <= x[3] <= depth/2
         // (i.e. the charged patch lies in the 1-3-plane, symmetrical
         // arround the line x[1]=0.
         // Notice that in 2D we can evaluate fields and potentials
         // only in the 1-2-plane but the field generating physical
         // structure (the charged patch) nevertheless has an extension
         // in 3-direction. We prefer here to have the patchcharge in the
```

```
      // 1-3 plane so that always the 2-axis (which is the vertical
      // axis both in the 3D and the 2D version of the PaDyAn-model)
      // is perpendicular to the charged rectangle. Since the general
      // theory in my Mathematica files is developed for the rectangle
      // lying in the 1-2-plane there is a slight complication in the
      // implementation that uses coordinates (x,y,z) corresponding to
      // axes (1,3,-2) and does so for 2D and 3D simulaneously.

   R w;
      // width of the patch

   R depth;
      // extension in 3-direction

   R sigma;
      // electric charge areal density
      // total electrical charge thus is w*depth*sigma

   R r;
      // a regularization radius

public:
   static B makeFieldFromFieldU;
   typedef PatchCharge Type;
   CPM_IO
   Word nameOf()const{ return Word("PatchCharge");}
   PatchCharge():w(0),depth(0),sigma(0),r(SysControl::singRange){}
   PatchCharge(const RefSys& rs_, R w_, R depth_,
      R sigma_=0, R r_=SysControl::singRange):
      rs(rs_),w(w_),depth(depth_),sigma(sigma_),r(r_){}

   R get_r()const{ return r;}
   void set_r(R r_){ r=r_;}

   R get_w()const{ return w;}
   R set_w(R w_){w=w_;}

   void setDepth(R d){ depth=d;}
   R getDepth(){ return depth;}

   void invertCharge(){ sigma=-sigma;}
   R get_sigma()const{ return sigma;}
   void set_sigma(R sigma_){ sigma=sigma_;}

   RefSys get_rs()const{return rs;}
      // read only, manipulation of rs should be confined to calling
      // operators *= and +=

   Spc get_x()const{  return rs.o();}
      // origin of the reference system
```

```
V<Spc> range()const;
   // returns two points on the boundary of the patch
   // the connecting line of which is orthogonal to this
   // boundary

SubsetOfPlane getPatch()const;
   // returns the charged area as a object

PatchCharge& operator *=(const Group& g){rs*=g; return *this;}
   // action of the basic symmetry group on a charged patch.

PatchCharge& operator *=(const AxVec& av){rs*=av; return *this;}
   // transformation specialized to rotation

PatchCharge& operator +=(const Vec& v){rs+=v; return *this;}
   // transformation specialized to translation

V<PatchCharge> increase(const V<Vec>& vv)const;
   // returns a set of copies of *this, each shifted by
   // a vector which is a component of vv

FieldE field()const;
   // returns the electric field generated by the source *this

VecE eval(const Spc& x)const
   { return electrostaticField(x,*this);}

R evalU(const Spc& x)const
   { return electrostaticPotential(x,*this);}

ScalarField fieldU()const;
   // returns the electric potential as a field

static VecE electrostaticField(const Spc& x,
   const PatchCharge& pc );
   // the main physical law as a plain function
   // no longer needed since 03-09-13 since now
   // we use FieldE = -grad(fieldU)

static R electrostaticPotential(const Spc& x,
   const PatchCharge& pc);
   // the main physical law as a plain function, very
   // transparent implementation.

void mark(const Camera& sc, Color c,  bool shAx=false)const;
   // if shAx==true the axes of RefSys.rs are shown too,
   // also in color c. Axes 1,2,3 can be discerned by the number of
   // structural elements they are made of.
```

```
      static R test();
         // Tests the consistence of computing the electrical field
         // directly or as a gradient of the electrical potential.
         // returns a number that had to be zero if no
         // numerical inaccuracies would be present.
         // More informations get written on cpmcerr so that
         // the return value should actually not be needed.
         // The tedious process of finally making sure that
         // electric field and electrical potential are
         // related in the right way triggered the idea to have
         // such a class test, which could be a model for
         // similar tests of other classes with logically involved
         // functionality. The present intention is not to create a
         // universal investigation tool with many parameters
         // but a randomly chosen object tested at random points.
   };

///////////////////////// ElcDipole ////////////////////////////////////

   class ElcDipole{ // electric dipoles
      // Describes 'real' electric dipoles (i.e. the limit to zero
      // spacing of a finitely spaced pair of charges. This is the
      // charge distribution for which the function
      // electrostaticDipoleField(...) gives the correct field.

      Vec d;
         // electric dipole moment (A*s*m)

      R r;
         // a regularization radius

      Spc x;
         // position

   public:
      typedef ElcDipole Type;
      CPM_IO
      Word nameOf()const{ return Word("ElcDipole");}

      ElcDipole():r(SysControl::singRange){}
         // x and d get initilized as zero vectors automatically

      ElcDipole(const Vec& d_, const Spc& x_,
         R r_=SysControl::singRange):d(d_),r(r_),x(x_){}

      Vec get_d()const{ return d;}

      Spc get_x()const{ return x;}

      R get_r()const{ return r;}
```

```
void add(const ElcDipole& ed){ d+=ed.d;}
   // adds the electric moment of a dipole

void set_r(R r_){ r=r_;}

void set_x(const Spc& x_){ x=x_;}

void setDir(const Vec& v){ d.setDir(v);}

void setMoment(const Vec& v){ d=v;}

Vec getMoment()const{ return d;}

R absVal()const{ return d.absVal();}
   // absolute value of the electric moment

ElcDipole& operator *=(R s){d*=s; return *this;}

ElcDipole& operator *=(const Group& a){x*=a; d*=a; return *this;}
   // action of the basic symmetry group on a electric dipole.
   // Notice that x and d belong to different classes (Spc and Vec,
   // respectively) the group element a does different things to
   // them.

ElcDipole& operator +=(const Vec& a){x+=a; return *this;}
   // action of a shift on a electric dipole.

ElcDipole& operator *=(const AxVec& a){ d*=a; return *this;}
   // action of a rotation on a the electric dipole vector (place
   // held fixed)

AxVec torque(const VecE& E)const;
   // torque on *this caused by the electric field strength E

R potEnr(const VecE& E)const{ return (-d|E);}
   // potential energy of *this with respect by the homogeneous
   // electric field of strength E

ForTor forceAndTorque( const F<Spc,VecE>& fE,
      R r=SysControl::singRange)const;
   // Force and torque on *this caused by the electric field fE.
   // The force depends on the the gradient of fE. Internally the
   // field is evaluated at two points, the distance of which is
   // 2*r

ForTor forceAndTorque( const ElcDipole& md )const;
   // Force and torque on *this caused by a electric dipole md.

FieldE field()const;
```

```
         // returns the electric field (E) generated by the source *this

      ScalarField fieldU()const;
         // returns the electric potential as a field

      VecE eval(const Spc& x)const
         { return electrostaticDipoleField(x,*this);}

      R evalU(const Spc& x)const
         { return electrostaticDipoleFieldU(x,*this);}

      static VecE electrostaticDipoleField(const Spc& x,
         const ElcDipole& md);
         // the main physical law as a plain function

      static R electrostaticDipoleFieldU(const Spc& x,
         const ElcDipole& md);
         // the main physical law as a plain function
   };

////////////////// class MagPole ////////////////////////////////////

   class MagPole{ // magnetic point charges
      // describes magnetic point charges and the forces they feel and
      // they create

      R q;
         // magnetic charge (V*s)

      R r;
         // a regularization radius

      Spc x;
         // position

   public:
      typedef MagPole Type;
      CPM_IO
      Word nameOf()const{ return Word("MagPole");}
      MagPole():q(0),r(SysControl::singRange){}
         // x gets initilized as zero vectors automatically

      MagPole(R q_, const Spc& x_ ,R r_=SysControl::singRange):
         q(q_),r(r_),x(x_){}

      R get_q()const{ return q;}

      void set_q(R q_){ q=q_;}

      R get_r()const{ return r;}
```

```
   void set_r(R r_){ r=r_;}

   Spc get_x()const{ return x;}

   void set_x(const Spc& x_){ x=x_;}

   void invertCharge(){ q=-q;}

   MagPole& operator *=(R s){q*=s; return *this;}

   MagPole& operator *=(const Group& a){x*=a; return *this;}
      // action of the basic symmetry group on a magnetic pole.

   MagPole& operator *=(const AxVec& a){a; return *this;}
      // see PointCharge::operator*=(const AxVec& a)

   MagPole& operator +=(const Vec& a){x+=a; return *this;}
      // action of a shift group on a magnetic pole.

   Force force( VecBH const& a )const{ return Force(a.v()*q);}
      // force on *this acted on by an magnetic field strength H

   Force force( const F<Spc,VecBH>& f)const{ return force(f(x));}
      // returns the force on the magnetic point charge as a result of
      // the interaction with the argument field, which is a vector
      // field giving the spatial distribution of the magnetic field
      // strength (H)

   Force force( const MagPole& mp )const;
      // force on *this caused by a magnetic pole mp

   FieldBH field()const;
      // returns the magnetic field generated by the source *this

   VecBH eval(const Spc& x)const
      { return magnetostaticField(x,*this);}


   static VecBH magnetostaticField(const Spc& x, const MagPole& mp);
      // the main physical law as a plain function
};

class MagDipole{ // magnetic dipoles
   // Describes 'real' magnetic dipoles (i.e. the limit to zero
   // spacing of a finitely spaced pair of poles

   Vec j;
      // magnetic dipole moment (V*s*m)

   R r;
```

```
      // a regularization radius

   Spc x;
      // position
public:
   typedef MagDipole Type;
   CPM_IO
   Word nameOf()const{ return Word("MagDipole");}
   MagDipole():r(SysControl::singRange){}
      // x and j get initilized as zero vectors automatically

   MagDipole(const Vec& j_, const Spc& x_,
      R r_=SysControl::singRange):j(j_),r(r_),x(x_){}

   Vec get_j()const{ return j;}

   Spc get_x()const{ return x;}

   R get_r()const{ return r;}

   void set_r(R r_){ r=r_;}

   void set_x(const Spc& x_){ x=x_;}

   void setDir(const Vec& v){ j.setDir(v);}

   void setMoment(const Vec& v){ j=v;}

   Vec getMoment()const{ return j;}

   R absVal()const{ return j.absVal();}
      // absolute value of the magnetic moment

   void add(const MagDipole& md){ j+=md.j;}

   MagDipole& operator *=(R s){j*=s; return *this;}

   MagDipole& operator *=(const Group& a){x*=a; j*=a; return *this;}
      // action of the basic symmetry group on a magnetic dipole.
      // Notice that x and j belong to different classes (Spc and Vec,
      // respectively) the group element a does different things to
      // them.

   MagDipole& operator +=(const Vec& a){x+=a; return *this;}
      // action of a shift on a magnetic dipole.

   MagDipole& operator *=(const AxVec& a){ j*=a; return *this;}
      // action of a rotation on a the magnetic dipole vector (place
      // held fixed)
```

```
      AxVec torque(const VecBH& H)const;
         // torque on *this caused by the magnetic field strength H

      R potEnr(const VecBH& H)const{ return (H.v()|-j);}
         // potential energy of *this with respect to the homogeneous
         // magnetic field of strength H

      R pot(MagDipole const& md)const
         { return md.potEnr(eval(md.get_x()));}
         // potential energy of the pair *this and md
         // of magnetic dipoles

      ForTor forceAndTorque( const F<Spc,VecBH>& fH,
            R r=SysControl::singRange)const;
         // Force and torque on *this caused by the magnetic field fH.
         // The force depends on the the gradient of fH. Internally the
         // field is evaluated at two points, the distance of which is
         // 2*r

      ForTor forceAndTorque( const MagDipole& md )const;
         // Force and torque on *this caused by a magnetic dipole md.

      FieldBH field()const;
         // returns the magnetic field generated by the source *this

      ScalarField fieldU()const;
         // returns the magnetic potential as a field (this is related
         // to the potential energy of a magnetic pole just as
         // the electric potential is related to the potential
         // energy of a point charge.

      VecBH eval(const Spc& x)const
         { return magnetostaticDipoleField(x,*this);}

      static VecBH magnetostaticDipoleField(const Spc& x,
         const MagDipole& md);
         // the main physical law as a plain function

      static R magnetostaticDipoleFieldU(const Spc& x,
         const MagDipole& md);
         // the main physical law as a plain function
   };

/////// class SourceOfVectorField ////////////////////////////////////

   class SourceOfVectorField{
      // Sources of static electromagnetic and gravitational fields.
      // Describes the far field of a static electromagnetic and
      // gravitational field by giving its lowest order terms of a
      // multipole expansion
```

```
// idependent quantities

      R1 m;
         // mass (kg)

      R1 q;
         // electric charge (A*s), sum of qi


      Vec d;
         // electric dipole moment (A*s*m)
         // sum of qi*(xi-x)

      Vec j;
         // magnetic dipole moment (V*s*m)

      R1 r;
         // a regularization radius

      Spc x;
         // position

// dependent quantities

      PointCharge pc;
      ElcDipole ed;
      MagDipole md;
      PointMass pm;

      void update()
      {
         pc=PointCharge(q,x,r);
         ed=ElcDipole(d,x,r);
         md=MagDipole(j,x,r);
         pm=PointMass(m,x,r);
      }

      void updateQD()
      {
         pc=PointCharge(q,x,r);
         ed=ElcDipole(d,x,r);
      }

      void updateDJ()
      {
         ed=ElcDipole(d,x,r);
         md=MagDipole(j,x,r);
      }
```

```
      void updateD(){ ed=ElcDipole(d,x,r);}

      void updateJ(){ md=MagDipole(j,x,r);}

      void updateM(){ pm=PointMass(m,x,r);}

   public:
      typedef SourceOfVectorField Type;
      CPM_IO
      Word nameOf()const{ return Word("SourceOfVectorField");}

      SourceOfVectorField(){}

      SourceOfVectorField(const Spc& x0, R r0):x(x0),r(r0){update();}
         // d and j get initilized as zero vectors automatically

      explicit SourceOfVectorField(const PointCharge& pcI,
         const ElcDipole& edI=ElcDipole(),
         const MagDipole& mdI=MagDipole(),
         const PointMass& pmI=PointMass()):
         q(pcI.get_q()),d(edI.get_d()),j(mdI.get_j()),
         m(pmI.get_m()), r(pcI.get_r()),x(pcI.get_x()){update();}

      PointCharge getCharge()const{ return pc;}
         // evident
      ElcDipole getElcDipole()const{ return ed;}
         // evident
      MagDipole getMagDipole()const{ return md;}
         // evident
      PointMass getMass()const{ return pm;}
         // evident

// trivial access functions
      Spc get_x()const{ return x;}
      R get_r()const{ return r;}
      R get_q()const{ return q;}
      R get_m()const{ return m;}
      Vec get_d()const{ return d;}
      Vec get_j()const{ return j;}
      void set_r(R r_){ r=r_;update();}
      void set_x(const Spc& x_){ x=x_;update();}

// adding sources of static electromagnetic fields
      void add(const PointCharge&);
      void add(const ElcDipole&);
      void add(const MagDipole&);
      void add(const PointMass&);
      void add(const SourceOfVectorField&);

// transformations
```

```
      SourceOfVectorField& operator *=(R s)
         {q*=s; d*=s; j*=s; m*=s; update();return *this;}
         // multiplying all sources by a common factor s

      SourceOfVectorField& operator *=(const Group& a)
         {x*=a; d*=a; j*=a; update(); return *this;}
         // action of the basic symmetry group on *this

      SourceOfVectorField& operator +=(const Vec& a)
         {x+=a; update();return *this;}
         // action of a shift

      SourceOfVectorField& operator *=(const AxVec& a)
         { d*=a; j*=a; updateDJ();return *this;}
         // action of a rotation on a the dipole vectors (place x
         // held fixed)

// fields

      VectorField field()const
         // returns the field created by the source *this
      {
         return
         VectorField(pc.field()+ed.field(),md.field(),pm.field());
      }

      ScalarField fieldU()const{
         if (!SysControl::noInteractionMassToMass) return pm.fieldU();
         else  return pc.fieldU()+ed.fieldU();
      }

      FieldSpace eval(const Spc& x)const
      {
         return
         FieldSpace(VecE(pc.eval(x)+ed.eval(x)),md.eval(x),pm.eval(x));
      }

      R evalU(const Spc& x)const{
         if (!SysControl::noInteractionMassToMass) return pm.evalU(x);
         else return pc.evalU(x)+ed.evalU(x);
      }

      static V<SourceOfVectorField> redistribute(
         const V<SourceOfVectorField>& sl, const V<Spc>& pl);
      // relacing a typically large list sl of sources, by
      // typically much smaller such list (the result res of the
      // function) where the positions of the result sources are
      // given as res[i].get_x()==pl[i].
      // Going from sl to res is a course graining procedure.
```

```
   // The far field of both lists of sources is the same.
   // The source list sl can play a role similar to a multipole
   // expansion of the far field. Since we need the field also
   // quite close to the point cloud pl, a set of a few
   // multipoles up to dipole order is more efficient than
   // a single multipole up to higher order.
   // The method of defining the course grained field is as follows:
   // Each source sl[j] gets transformed into a a source located
   // at this point pl[k] of the list pl which is closest to
   // sl[j].get_x();

   static V<SourceOfVectorField> redistribute2(
      const V<SourceOfVectorField>& sl, const V<Spc>& pl,
      const V<Z>& whereToGo);
   // Faster version of the previous function, applicable if
   // additional information is abvailable (2004-05-09)
   // relacing a typically large list sl of sources, by
   // typically much smaller such list (the result res of the
   // function) where the positions of the result sources are
   // given as res[i].get_x()==pl[i].
   // Going from sl to res is a course graining procedure.
   // The far field of both lists of sources is the same.
   // The source list sl can play a role similar to a multipole
   // expansion of the far field. Since we need the field also
   // quite close to the point cloud pl, a set of a few
   // multipoles up to dipole order is more efficient than
   // a single multipole up to higher order.
   // The method of defining the course grained field is as follows:
   // Each source sl[j] gets transformed into a a source located
   // pl[whereToGo[j]]
};
```

80 cpmfr.h

```
//? cpmfr.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_FR_H_
#define CPM_FR_H_
/*
   Purpose: Define classes describing function-like objects with
      arithmetic operations

*/

#include <cpmfa.h>
#include <cpmvo.h>

namespace CpmFunctions{

   using namespace CpmStd;

   using CpmRoot::R;
   using CpmRoot::Word;

template <class X, class Y>

class Fr: public Fa<X,Y> { // version of Fa with rich interface

   static Z imax;
   static Z cpl;
      // means for implementing order and equality
      // by comparison of function values ( actually all
      // values would be needed for being mathematically exact)
      // Another way of looking at the matter is that imax and cpl define
      // an equivalence relation on Fr<X,Y> and order and equality are
      // defined exactly on the set of equivalunce classes. The larger
      // imax and cpl are, the smaller the equivalence classes

public:

   typedef Fr<X,Y> Type;

   Fr(void):Fa<X,Y>(){}
      // default constructor

   Fr(const F<X,Y>& g):Fa<X,Y>(g){}
      // upcast constructor

   Fr(const Fo<X,Y>& g):Fa<X,Y>(g){}
      // upcast constructor
```

```
   Fr(const Fa<X,Y>& g):Fa<X,Y>(g){}
      // upcast constructor, note Fa = Fa

   Fr(const Fr<X,Y>& g):Fa<X,Y>(g){}
      // copy constructor

   Fr(FncObj<X,Y>* fop):Fa<X,Y>(fop){}
      // constructor from pointers to FncObj

// Construction from function pointers

   Fr( Y (*f)(const X& )):Fa<X,Y>(f){}
   Fr( Y (*f)(X)):Fa<X,Y>(f){}

// Construtor for constant function

   Fr(const Y& y1):Fa<X,Y>(y1){}
      // takes always the value y1, irrespective of x

   Fr(CpmArrays::V<X> const& xl, CpmArrays::V<Y> const& yl);
      // Construction from a list of x-values and a list of corresponding
      // y-values. Makes a staircase-function since real interpolation
      // assumes R-related operations in Y which we don't want to assme
      // here.

   virtual F<X,Y>* clone()const{ return new Fr(*this);}

   CPM_IO
   CPM_TEST_X
   CPM_DESCRIPTORS
   CPM_CONJUGATION

   Y operator()(X const& x)const;
      // redefinition with better error handling

   R compare(Word top, Fr<X,Y> const& rhs)const;

};

template <class X, class Y>
Y Fr<X,Y>::operator()(const X& x)const
      // notice the * operator from class P<>
{
   using CpmArrays::P;
   try{
      return (P< FncObj<X,Y> >::operator()())(x);
   }
   catch(...){
      CpmRoot::Word wx=CpmRoot::Name<X>()(X());
```

```
      Y y=Y();
      CpmRoot::Word wy=CpmRoot::Name<Y>()(y);
      cpmerror("trouble in Y Fa<X,Y>::operator()(const X& x)const\
      for X type "&wx&" and Y type "&wy);
      return y; // to avoid warning
   }
}

// helper classes for implementation

namespace aux{
template <class X, class Y>
class CONJFUO : public FncObj<X,Y>{
   const Fr<X,Y> f;
public:
   CONJFUO(const Fr<X,Y>& f_):f(f_){}
   Y operator()(const X& x)const
   { return CpmRoot::con<Y>(f(x));}
};

template <class X, class Y>
class RANDFUO : public FncObj<X,Y>{
   const Z i;
   const Z tvs;
   const Z memory;
public:
   RANDFUO(const Z& i_, Z tvs_, Z mem_ ):i(i_),tvs(tvs_),
      memory(mem_){}
   Y operator()(X const& x)const
      // never invoke ranVal() since then the random functions
      // depend on history
   {
   // would make operator() a mutating operator
   // this would have rather opaque implications
      Z iAct=(i==0 ? memory : i);
      Y ys;
      ys=CpmRoot::testVal<Y>(ys,tvs);
      ys=CpmRoot::ranVal<Y>(ys,iAct);
      Z j=CpmRoot::hashVal<X>(x);
      if (j==0)
         return ys;
      else
         return CpmRoot::ranVal<Y>(ys,j);
         // notice that for j==0 ranVal behaves not as a function
         // (value depends on how often the routine was called)
   }
};

template <class X, class Y>
class IPOLFUO: public FncObj<X,Y>{
```

```
   // IPOL stands for interpolation. However, here we don't really
   // interpolate, we use staircase approximation instead.
   // This needs no assumptions concerning Y.
   const CpmArrays::V<X> xl;
   const CpmArrays::V<Y> yl;
public:
   Y operator()(const X& x)const;
   IPOLFUO( const CpmArrays::V<X>& xlIn, const CpmArrays::V<Y>& ylIn ):
      xl(xlIn),yl(ylIn){}
};

template <class X, class Y>
Y IPOLFUO<X,Y>::operator()(X const& x)const
{
   Z i,n=xl.dim();
   Word loc("IPOLFUO<X,Y>::operator()");
   cpmassert(yl.dim()>=n,loc);
   CpmArrays::Vo<R> dis(n);
   for (i=1;i<=n;i++){
      X xd=x-xl[i];
      dis[i]=CpmRoot::absVal<X>(xd);
   }
   Z j=dis.indInf();
   return yl[j];
}

} // aux

// end of helper classes

template <class X, class Y>
Z Fr<X,Y>::imax=8;

template <class X, class Y>
Z Fr<X,Y>::cpl=8;

template <class X, class Y>
Fr<X,Y>::Fr(const CpmArrays::V<X>& xl,
   const CpmArrays::V<Y>& yl)
   // a rather crude interpolation based on weighted means
   // of function values from the list
{
   *this=Fr(new aux::IPOLFUO<X,Y>(xl,yl));
}

template <class X, class Y>
Fr<X,Y> Fr<X,Y>::con(void)const
{
   return Fr<X,Y>(new aux::CONJFUO<X,Y>(*this));
}
```

```
template <class X, class Y>
Fr<X,Y> Fr<X,Y>::ranVal(Z i)const
{
   static Z mem=37;
   if (i==0) mem++;
   return Fr<X,Y>(new aux::RANDFUO<X,Y>(i,cpl,mem));
}

template <class X, class Y>
Word Fr<X,Y>::nameOf(void)const
{
   Word wi="Fr<";
   X x; Y y;
   Word wx=CpmRoot::Name<X>()(x);
   Word wy=CpmRoot::Name<Y>()(y);
   return wi&wx&","&wy&">";
}

template <class X, class Y>
Word Fr<X,Y>::toWord(void)const
{
   ostringstream ost;
   prnOn(ost);
   return Word(ost.str());
}

template <class X, class Y>
Fr<X,Y> Fr<X,Y>::testVal(Z i)const
{
   Y y0;
   Y yt=CpmRoot::testVal<Y>(y0,i);
   Fr<X,Y> res(yt);
   imax=i;
   cpl=i;
   return res;
}

template <class X, class Y>
bool Fr<X,Y>::prnOn(ostream& out)const
{
   out<<endl<<"// printing of "<<nameOf()<<endl;
   out<<endl<<"// Number of (x,y)-pairs: "<<endl;
   out<<imax<<endl;
   X x;
   Y y;
   X xs=CpmRoot::testVal<X>(x,cpl);
   for (Z i=1;i<=imax;i++){
      x=CpmRoot::ranVal<X>(xs,i);
      y=(*this)(x);
```

```
      bool b;
      out<<endl<<"// value pair number "<<i<<" of "<<imax;
      out<<endl<<"// x = "<<endl;
      b=CpmRoot::prnOn<X>(x,out);
      if (!b) return false;
      out<<endl<<"// y = "<<endl;
      b=CpmRoot::prnOn<Y>(y,out);
      if (!b) return false;
   }
   return true;
}

template <class X, class Y>
bool Fr<X,Y>::scanFrom(istream& in)
// reading value pairs and creating an interrpolated function
// from a file that may contain comments
{
   static const Z readLimit=512;
   Z nList;
   if (!CpmRoot::read(nList,in)) return false;
   cpmassert(nList>0,"Fr<X,Y>::scanFrom");
   cpmassert(nList<=readLimit,"Fr<X,Y>::scanFrom");
   CpmArrays::V<X> xl(nList);
   CpmArrays::V<Y> yl(nList);
   for (Z i=1;i<=nList;i++){
      if (!CpmRoot::scanFrom<X>(xl[i],in)) return false;
      if (!CpmRoot::scanFrom<Y>(yl[i],in)) return false;
   }
   *this=Fr<X,Y>(xl,yl);
   return true;
}

template <class X, class Y>
Z Fr<X,Y>::hashVal(void)const
// returns a key value to *this. This should be not
// expensive to calculate. Therfore the number of function
// evaluation is restricted by imax
{
   Z h=0;
   X x,xs;
   Y y;
   xs=CpmRoot::testVal<X>(x,cpl);
   for (Z i=1;i<=imax;i++){
      x=CpmRoot::ranVal<X>(xs,i);
      y=(*this)(x);
      h^=CpmRoot::hashVal<Y>(y);
         // bit operation since += may give overflow
   }
   return h;
}
```

```
template <class X, class Y>
R Fr<X,Y>::absSqr(void)const
// returns as an absolute value a mean absolute value of function values
// where the arguments are chosen as a random sequence.
{
try{
   X x,xs;
   Y y;
   xs=CpmRoot::testVal<X>(x,cpl);
   R res=0;
   for (Z i=1;i<=imax;i++){
      x=CpmRoot::ranVal<X>(xs,i);
      y=(*this)(x);
      R a=CpmRoot::absVal<Y>(y);
      res+=a*a;
   }
   cpmassert(imax>0,"Fr<X,Y>::absSqr");
   res/=imax;
   return sqrt(res);
}
catch(...){
   cpmerror("trouble in R Fr<X,Y>::absVal(void)const");
   return 0; // to avoid warning
}
}

template <class X, class Y>
R Fr<X,Y>::absVal(void)const
{
   return ::sqrt(absSqr());
}

template <class X, class Y>
R Fr<X,Y>::disVal(const Fr<X,Y>& x)const
{
   Fr<X,Y> diffVector=*this;
   diffVector-=x;
   R da=diffVector.absVal();
   R xa=x.absVal();
   R ya=absVal();
   return CpmRoot::distFunc(xa,ya,da);
}

} // namespace

#endif
```

81 cpmframe.h

```
//? cpmframe.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_FRAME_H_
#define CPM_FRAME_H_
/*

   Purpose: Declaration of class CpmGraphics::Frame
Active
*/

#include <cpmimagingtools.h>
#include <cpmimg24.h>
#include <cpmfontgeneration.h>

namespace CpmGraphics{

   using CpmRoot::Word;
   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::posVal;
   using CpmLinAlg::Z2;
   using CpmLinAlg::R2;
   using CpmLinAlg::R4;
   using CpmArrays::V;
   using CpmArrays::X4;
   using CpmImaging::Rect;
   using CpmImaging::Color;
   using CpmImaging::WRITECOLOR;
   using CpmImaging::WHITE;
   using CpmImaging::LIGHTBLUE;

   typedef CpmArrays::Vl<R> Rn;
   typedef CpmArrays::Vl<R2> Rn2;

/*
The geometry of the virtual screen:
We assume a rectangular screen and associate with it an orthogonal
system of coordinates which we refer to as screen coordinates.
We adopt a horizontal x-axis and a vertical y-axis.
The origin is assumed to be the left lower corner and the scale on the
axes is chosen such that the right upper screen corner has coordinates
x=1 and y=1.

A rectangular subset of the screen (a frame) is characterized
by the coordinates (xL,yL) and (xU,yU) of the left-lower corner and
of the right-upper corner of the frame in this system of coordinates.
```

```
The screen is subdivided into pixels, which are addressable by ordered
pairs of integers, say (i,j). Unfortunately, the indexing scheme adopted
for (nearly ?) all conventional graphics functions is different from any
scheme to be used in mathematics: neither (i,j) is a discretized version
of (x,y) nor (i,j) is the matrix index of a screen element, if the screen
is considered a large rectangular matrix. Instead we have

      i_Graphics = Min (xPixel_-1, (int)(xPixel_*x))
      j_Graphics = Min(yPixel_-1,(int)(yPixel_*(1-y))

(the Min prescription comes into action only to prevent pixel indexes
xPixel_ and yPixel_, which do not occur since enumeration starts with 0
and xPixel_ and yPixel_ are the horizonal and vertical pixel numbers
respectively.). (i_Graphics=0,j_Graphics=0) is the UPPER-LEFT corner of
the screen and i_Graphics varies in x-direction (along a pixel line)
and j_Graphics varies in y-direction (along a pixel column)

Corresponding matrix indexes are:

      i_Matrix=j_Graphics+1      1 <= i_Matrix <= yPixel_
      j_Matrix=i_Graphics+1      1 <= j_Matrix <= xPixel_

Some of the member functions of Frame will use matrix indexing.

With respect to the frame, we introduce a second system of coordinates
according to the same rules used for the x,y-system, but with the screen
replaced by the frame. Let us denote these coordinates xF and yF
and refer to the coordinate system as frame system. If the frame is
considered a matrix, the matrix indexes corresponding to a pixel are
denoted (iF,jF)

new nomenclature:
symbol for matrix indexes:                      ij
symbol for graphics indexes:                    ab
symbol for continuous Cartesian coordinates:    xy
prefixes S for screen(or, better, application window), F for frame,
G for graph so: SabToSxy means: from graphics indexes referring to
the screen to (x,y) coordinates of the screen system

*/

//////////////////////////// class FrmText ////////////////////////////////
class Frame;

typedef CpmArrays::X4<R2,Word,Color,R> TextCom;
      //: text component
      // holds write-position, text, write color, font height.
      // Position is in Frame specific x,y coordinates ranging
      // from 0 to 1. Origin is lower-left. Font height is a
```

```
      // tendative pixel height from which the actual pixel
      // height gets chosen by a selection table that takes
      // int account the available fonts.
      // write-position is the position of the upper-left
      // corner of the Word's writing box.
      // Thus x=0 and y=1 marks is the first writing position if
      // a text is to be written in the usual way from left to right and
      // from top to bottom. This selection allows for no margin.

class FrmText{ //Frame text
// independent
   V<TextCom> tcs_;
      // text components, see typedef above
public:
   FrmText(V<Word> const& vw, Frame const& fr, V<Color> const& c);
      // tries to place a system of text lines in the pixel
      // rectangle given by fr.
   FrmText(V<TextCom> const& tcs= V<TextCom>(1)); // 1 was 0
   FrmText& operator+=(TextCom const& tp);
   FrmText operator+(TextCom const& tp)const;
   const TextCom& operator[](Z i)const{ return tcs_[i];}
   Z dim()const{ return tcs_.dim();}
   bool noText()const
   { return tcs_.dim()==1 && tcs_[1].c2().dim()==0;}
};

//////////////////////////// class Frame ////////////////////////////////////

class Frame {
// rectangular subsets of the screen with pixel structure
   // class contains no pointers, thus no = operator is needed
   // Class R2 describes pairs of continuous coordinates
   // Class Z2 describes discrete pixel positions.
   // A frame is a rectangular subset of the screen which is made up
   // of integer pixels. Thus the x-coordinates of the corners are
   // multiples of Dx=1./xPixel_ and the y-coordinates are multiples of
   // Dy=1./yPixel_.

friend class Frames;
friend class FrmText;

// independent data:

protected: // should be available in class Graph

   static Vl<CpmFonts::Font> ft_;

   static Z fontSelector(R h);

   static Z xPixel_;
```

```
      // maximum number of columns of bitmaps

   static Z yPixel_;
      // maximum number of rows of bitmaps

   static xyxy scrRec;
      // virtual screen rectangle, described by the new type for
      // rectangular subsets of Z2. Internal data are all Sab.

   static bool xyPixelInitialized;

   static bool fontInitialized;

   static void iniStatics(void);
      // static member function which initializes the data members which
      // characterize virtual screen. This function has to be
      // called as a first action in each constructor body

   static Color defaultColorActive;
      // this color will be chosen for colorActive of a frame in the
      // absence of explicit specification

   static R xPixelInv, yPixelInv;
      // 1/xPixel and 1/yPixel for efficiency

   static R hFrac;
      // Determines the default text hight. See function
      // textHeight().

   static R yText;
      // proposal for Fy of the line on which a single line
      // of text will be written on default

   static Z imageCounter;
      // useful in mode CPM_NOGRAPHICS in which screen-graphics
      // will be replaced by ppm-images

// independent data

   Color colorActive;
      // If a frame is created by any constructor except the default
      // constructor it will be painted first by a color.
      // This color is just colorActive.

   R2 cL; // cL[1], cL[2] are the coordinates of the
      // lower left corner of the frame in the screen system of
      // coordinates. In comments, cL[1] and cL[2] are sometimes
      // referred to as xL, yL. Notice that frame corners
      // coincide with pixel corners (not pixel centers)
```

```
   R2 cU; // upper right corner. See cL

   mutable Img24 img;
      // memory for RGB data, added 2000-9-18
      // Whether needed or not, it is always present with a size
      // determined by the pixel count of *this.

     FrmText text_;
      // notice that this will automatically initialized
      // such that text_.dim()==1, text_[1].c2()==""

// dependent data:

   Z2 pL;
      // lower left pixel of the Frame in the screen
      // graphics pixel numbering scheme.
      // putPel(pL[1],pL[2],c) puts the color c to the
      // lower left corner of the Frame *this on screen
   Z2 pU;
      // upper right pixel of the Frame in the screen
      // graphics pixel numbering scheme.
      // putPel(pU[1],pU[1],c) puts the color c to the
      // upper right corner of the Frame *this on screen

      // Given the discrete values, the continuous cL, cU are given as
      // follows:
      // cL[1]=pL[1]*xPixelInv // should be 0 for pL[1]=0
      // cU[1]=(pU[1]+1)*xPixelInv // should be 1 for pU[1]=xPix-1
      // cL[2]=1-(pL[2]+1)*yPixelInv // should be 0 for pL[2]=yPixel-1
      // cU[2]=1-pU[2]*yPixelInv // should be 1 for pU[2]=0
         // OK for these settings

   Z nCol; // Number of pixel columns of the frame:
      // nCol=pU.x1-pL.x1+1

   Z nRow;  // Number of pixel rows of the frame:
      // nRow=pL.x2-pU.x2+1

   R nColInv;      // inverse of nCol

   R nRowInv;      // inverse of nRow

 // methods:

   void iniFrame(R x, R y, R dx, R dy, Color const& color);
      // tool for defining constructors without code repetition

   void updateRowCol(void);
   // updates nRow, nCol, nRowInv, nColInv if cL or/and cU have been
   // changed
```

```
   void updateFromDiscrete(void);
   // updates cL,cU,pU, nColInv, nRowInv, img if
   // nCol or/and nRow and pL have already their desired values.
   // img content will be cleared.

public:

   typedef Frame Type;

   CPM_IO_V  // virtual version of the I/O basic functions
      // The I/O functions
      // bool prnOn(ostream&)const
      // and
      // bool scanFrom(istream&)
      // need explanation for that class which is more strongly
      // related to the computer hardware than most of my classes.
      // The intention is this: Let a program running on one computer
      // (under Windows) a program which instantiates some Frame fr;
      // and writes the status of fr on a textfile
      // ofstream out("fr_status.txt");
      // fr.prnOn(out); // or out<<fr;
      // Then a program running on any other Windows computer using
      // an arbitrary graphics resolution should get in a program
      // saying
      // Frame fr2;
      // ifstream in2("fr_status.txt");
      // fr2.scanFrom(in2); // or in2>>fr2;
      // a graphics frame which in proportion to the whole screen has the
      // same extension in both computers. It is not the intention of
      // this I/O operations to save the content of video memory.

   void static setStaticSize(Z width, Z height);
      // the static members xPixel_, yPixel_, xPixelInv, yPixelInv,
      // scrRec get set to values determined by the arguments.
      // By the recent code revision class Frame became nearly
      // independent in functionality of the graphics hardware.
      // Only the display mode makes use of it.
      // For producing high resolution graphics,
      // (i.e with much more pixels than my screen has) it becomes
      // possible with this function to create larger frames.
      // For such large images to make sense one has to extend
      // lines to lines with thickness, since one pixel width
      // becomes too thin for some applications in say 4000x6000 pixel
      // images. The benefit of this is that aliasing does not matter
      // if pixels are so small compared to total size. An alternative
      // pathway to high quality graphics is dercribed in the
      // 2006-01-15 entry to the History of this file.

   static Z getScreenWidth(){return xPixel_;}
```

```
static Z getScreenHeight(){return yPixel_;}

static void setDefaultColorActive(Color const& c)
   {defaultColorActive=c;}
   // allows to influence the primary color of frames which were
   // created without a color setting
static void set_hFrac(R x){ hFrac=x;}
static R get_hFrac(R x){ return hFrac;}
static void set_yText(R x){ yText=x;}
static R get_yText(R x){ return yText;}
static xyxy getScrRec(){ return scrRec;}
   // getting the screen rectangle in 'modern' form

virtual Word nameOf()const{ return Word("Frame");}

xyxy getFrmRec()const{ return xyxy(xy(pL.x1,pU.x2),xy(pU.x1,pL.x2));}
   // getting the frame rectangle in 'modern' form. We don't use a
   // data member to hold this object since I have too many
   // initilization and adaption methods to be sure that this would be
   // initialized correctly in all these methods.
   // Notice that in the graphics indexing scheme it is natural to
   // describe a rectangle by left-upper corner and right lower
   // corner.
   // This is the way followed in the definition of IvZ2.
   // By contrast, pL and pU are left-lower and right upper. This
   // causes the deviation from the probably expected formula
   // return IvZ2(Z2(pL.x1,pL.x2),Z2(pU.x1,pU.x2));

static R2 SabToSxy(Z2 p)
{   return R2((p.x1+0.5)*xPixelInv,1.-(p.x2+0.5)*yPixelInv);}
   // for given graphics indexes p=(a,b) (thus Sab)
   // of a screen pixel, the center of the screen pixel is
   // returned, expressed as screen coordinates (x,y) (thus Sxy)

static Z2 SxyToSab(R2 c);
   // given coordinates of a point in the screen (x,y)-sytem
   // the graphics indexes are given of the screen pixel to which
   // the point belongs

R getAspRat()const{ return nCol*nRowInv;}
   // aspect ratio of the frame assuming square pixels.
   // Landscape format gives values larger than 1
   // Portrait < 1;
   // image width / image height

R getAspRatInv()const{ return nRow*nColInv;}
   // inverse of aspect ratio

static R screenAspRat(){ return xPixel_*yPixelInv;}
static R screenAspRatInv(){ return yPixel_*xPixelInv;}
```

```
void setAspRatHoldWidth(R asp);
   // change the image height till the aspect ratio is as given
   // by the argument or the frame reaches the border of the
   // screen. The lower left corner is held fixed.
   // Image content will be cleared.

void setAspRatHoldHeight(R asp);
   // change the image width till the aspect ratio is as given
   // by the argument or the frame reaches the border of the
   // screen. The lower left corner is held fixed.
   // Image content will be cleared.

void setAspRat(R asp, bool holdHeight=true)
{ if (holdHeight) setAspRatHoldHeight(asp);
   else setAspRatHoldWidth(asp);}
   //: set aspect ratio
   // for convenience

void setThcLin_(bool b){ img.setThickLines(b);}
   //: set thick lines
   // if this is set to true, all calls to function line(R,R,R,R,rgb)
   // will create thick lines.

bool getThcLin()const{ return img.getThickLines();}
   //: get thick lines
   // returns the status of line thicking


void cut(R asp, Z yPixel);
   // reduces (or enlarges) *this to a maximum subset with
   // aspect ratio asp and an height in pixels less or equal to
   // yPixel.
   // If there is room for a frame height of yPixel, then
   // the subframe is placed approximately at a central
   // position.

explicit Frame(const Rect& rec);
   // constructor from a rectangle in the whole screen

Frame(const Frame& fr, const Rect& rec);
   // constructor from a rectangle of the rectangle given by fr

Rect getRectangle(void)const;
   // Returns the rectangle from which the Frame 'could have been
   // created'.

Frame(R x, R y, R dx, R dy, Color const& color=defaultColorActive);
   // Creates a frame, the lower corner of which has approximately
   // lower left corner (x,y) and upper right corner (x+dx,y+dy).
```

```
   // The precise corners are made to agree with the pixel pattern.
   // The argument color determines the data member colorActive
   // in which the frame will be shown on screen.
   // If one want to create a frame that is not visible on screen just
   // by call of the constuctor (e.g. since the aspect ration is only
   // preliminary and has to be reset by calling function
   //       setAspRatHoldHeight(R))
   // one may proceed as follows:
   // R x,y,dx,dy = ....
   // Color cF=...
   // Frame f0;
   // f0.setxy(x,y,dx,dy);
   // f0.setAspRatHoldHeight(1);
   // f0.paint(cF);
   // f0.display(); // only now, a visible action happens

Frame(void);
   // creates the full frame as given by xPixel_ and yPixel_
   // with colorActive=defaultColorActive but no painting activity
   // invoked

explicit Frame(Color const& c);
   // same as Frame() but colorActive=c and painting invoked

explicit Frame(Z code);
   // Constructor for convenient placing of a frame
   // The screen is divided in quadrants denoted in matrix-style
   // 11 12
   // 21 22
   // Posible position codes are 11, 12, 21, 22, 1112 (for upper
   // half part) 2122 (lower half part), 1121 (left half part)
   // 1222 (right part)

explicit Frame(Img24 const& im, R x=0, R y=0, bool showIt=true);
   // creates a frame which has lower-left corner (x,y)
   // ( where 0<=x<=1, 0<=y<=1) and has the pixel content of im.
   // Notice that the input image may have been obtained via the
   // getImg()-method from another Frame (or Graph).
   // For showIt==false, no screen display action is caused by
   // calling this constructor.

Z2 discrete(R2& pLower, const R2& p, R2& pUpper);
   // consider a point, which in the screen system has
   // coordinates (p.x, p.y). 0<=p.x<=1, 0<=p.y<=1.
   // The screen is partitioned into pixels in a disjoint and
   // complete manner ( a normal pixel is considerd to own its left
   // and its bottom boundary; only the pixel of the upper row and
   // the rightmost column are considered to own all its boundary)
   // Then screen point belongs to just one pixel. The return value
   // is the double index of this pixel according to the graphics
```

```
   // pixel numbering scheme. pLower is the lower-left corner of this
   // pixel and pUpper is the upper-right corner of this pixel.

R2 getLowerLeftCorner(void)const{ return cL;}
   // Put
   // R2 c=getLowerLeftCorner;
   // Then (c.x,c.y) is the lower left frame corner in the screen
   // system of reference

R2 getUpperRightCorner(void)const{ return cU;}
   // see getLowerLeftCorner()

Z dim1()const{ return nCol;}
Z dim2()const{ return nRow;}
   // uniform over Viewport, Img24, Frame
Z dim_x()const{ return nCol;}
Z dim_y()const{ return nRow;}
Z w()const{ return nCol;}
Z h()const{ return nRow;}

Img24 getImg()const{ return img;}
   //: get image
   // This is an important method since many of the C+- image
   // creation methods primarily create a graphical image
   // particularly employing class Graph which is derived from
   // Frame.

Z getCol(void)const{ return nCol;}
   // get number of columns (nCol, name as in image classes)

Z getLin(void)const{ return nRow;}
   // get number of lines (nRow, name as in image classes)

Z2 get_pU(void)const;
   // get upper right pixel pU.

Z2 get_pL(void)const;
   // get lower left pixel pL.

Z a0()const{ return pL.x1;}
   // difference Sa-Fa

Z b0()const{ return pU.x2;}
    // difference Sb-Fb

R2 shf()const{ return R2(-pL.x1,-pU.x2);}
   //: shift
   // combines a0(), and b0() in a way that they may
   // directly be used to shift real coordinates in the way
   // one needs it to connect the graphical primitives of class
```

```
      // Frame to those of class Img24, see function fill(TriAng)

// getting x,y,dx,dy having the same meaning as the arguments in
// setxy(R,R,R,R)

   R getx(void)const{ return cL.x1;}
      // (x,y) is the lower left corner

   R gety(void)const{ return cL.x2;}

   R getdx(void)const{ return cU.x1-cL.x1;}
      // (x+dx,y+dy) is the upper right corner

   R getdy(void)const{ return cU.x2-cL.x2;}

   R4 getxy()const{ return R4(cL.x1,cL.x2,cU.x1-cL.x1,cU.x2-cL.x2);}

   void setxy(R x, R y, R dx, R dy);
      // Helps to consistently set cL, cU. (x,y) is the intended
      // left-lower corner is the screen system; dx and dy are
      // width and height of the frame respectively in units in
      // which hight and width of the screen both are 1.
      // The precise values actually taken by the frame may differ
      // slightly from these prescriptions since the final frame will
      // always fit to the screen pixel raster. Clearly, the frame
      // never will be set larger than allowed by xPixel_ and yPixel_.

   void set_xy_(R4 const& xy){ setxy(xy[1],xy[2],xy[3],xy[4]);}
      //: set xy
      // same as previous function with a single argument

   void scl_(R fac)
   { R4 x=getxy(); set_xy_(R4(x[1],x[2],x[3]*fac,x[4]*fac));}
      //: scale
      // Location of the lower left corner remains fixed and the
      // with and the height, both get multiplied by fac.

   void setpLpU(Z2 pLIn, Z2 pUIn);
      // Helps to consistently set cL, cU. Corners of the frame are
      // set in terms of screen pixel numbers.
      // Needed to define a precise partition of a frame into subframes
      // without any overlap or gaps (96-11-28)

   void setColorActive(Color const& c){ colorActive=c;}
   void setColor(Color const& c){ colorActive=c; img.setColor(c);}
      // for uniformity with related classes. Controls
      // the color to be used by function clear
   Color getColorActive(void)const{ return colorActive;}

   void paint(Color const& c);
```

```
   // paints *this with color. The painted screen will become visible
   // only after calling display().

void clear();
   // paints *this with colorActive and clears text
   // The cleared screen will become visible
   // only after calling display().

void fullClear();
   // paints with screen area and image area with black
   // always

void clr_(Color c=LIGHTBLUE){clearText();paint(c);display();}
   //: clear
   // C+- conformant name of mutating function clear

void clr(Color c=LIGHTBLUE){ clr_(c);}
   // clear
   // old name, use clr_ instead

Word dis(bool file=false, bool lineArtProcessing=false)const;
   //: display
   // Displays the Frame together with the text content given by data
   // member text_. If the first argument has value true also a copy
   // on file as a ppm-image will be made. The name of the image
   // file is frame_001.ppm, ... frame_999.ppm
   // where the name and the number are assigned automatically.
   // If the second argument has value true, a transformation of the
   // image is done before display and image filing which is useful
   // for frame contents which are B/W line art. The pixel structure
   // which in my graphics system deviates sometimes by +/- one pixel
   // from what could be considered nice and natural should become
   // more regular by this processing.
   // Dependent on the return value of Message::appRunId(), in addition
   // to the frame number, a program run ID will appended to the name.
   // The return value is the name of the file which was written, or
   // the void Word() if no image file was written.

Word display(bool file=false, bool lineArtProcessing=false)const
   { return dis(file,lineArtProcessing);}
   // old name

void imgTest()const{ img.display();}

Word write(const Word& file)const;
   // writing the pictorial content of *this int a image file
   // file&".ppm" in the current directory
   // returned is the file name with the extension actually appended

Word wrt(Word const& file)const
```

```
   { if (file.isVoid()) return file; else return write(file);}
      //: write
      // nearly as write but C+- compliant name. Does not write
      // if the file name is void

   Word write(void)const;
      // the image file is that of img.write()

// transformation of xF,yF coordinates into graphics pixel indices

   Z ix(R xF)const;
   Z jy(R yF)const;
// transformation between frame related continuous and discrete
// coordinates (xF,yF) and (iF,jF)

   Z2 getiFjF(const R2& xFyF)const;
      // given coordinates of a point in the frame system, the
      // matrix indexes of the pixel to which the point belongs
      // is given

   xy FxyToFab( R2 p)const
      // frame x,y coordinates to frame graphics indexes
      // which are directly related to the pixels of img
   {
      return xy(Iv(0,1).quant(p.x1,w(),true)-1,
              Iv(0,1).quant(1-p.x2,h(),true)-1);
   }

   R2 getxFyF(const Z2& iFjF)const
      // for given matrix indexes of a pixel within the frame,
      // the center of this pixel is returned, expressed as
      // coordinated in the frame system
{ return R2((iFjF.x2-0.5)*nColInv,1.-(iFjF.x1-0.5)*nRowInv);}

   void addColors(){ img.addCol();}
      // graphical primitives add their colors to background
      // (transparent colors, watercolor, Aquarellfarben)

   void setColors(){ img.setCol();}
      // graphical primitives replace background colors by their
      // own ones. // (body colors, Deckfarben)

   void stereo();
      // anaglyphic representation

   bool getAntiAliasLines()const
      { return img.getAntiAliasLines();}

   void setAntiAliasLines(bool aal=true, R gamma=-1)
      {
```

```
      img.setAntiAliasLines(aal);
      if (aal==true){ // only then influence gamma
         if (gamma<=0) img.setGamma(Viewport::getGamma());
         else img.setGamma(gamma);
      }
   }
   // this allows creation of better line graphics
   // and increases the value of class VecGraph
   // working of antiAliasLines strongly depends on gamma

R getGamma()const{ return img.getGamma();}
void setGamma(R g){ img.setGamma(g);}

FrmText getText()const{ return text_;}
void setText(FrmText const& text){ text_=text;}
void addText(FrmText const& text);
   // adds text components if their locations are not yet
   // set. Otherwise components of text_ get replaced
   // by components of text.
void setText(Word const& text, R x=0.05, R y=0,
   Color c=WRITECOLOR, R h=0);
   // Notice that the left lower corner of the Frame *this is
   // (x=0,y=0) and the right upper corner is (x=1,y=1).
   // Setting y=0, or using the default y which happenes to
   // be y=0 too, enforces an automatic setting for y which
   // should be OK for a single line diagram title.
   // For y>0 and x>=0 (x,y) is the position of the left - upper
   // corner of the writing box. Thus (x,y) = (0,1) writes in contact
   // to the upper rim of the frame, which is okay if there is no
   // space to waste.
   // h<=0 and y<=0 determines the values to be used from static
   // data hFrac and yText.

R2 plcText(Word const& text, R2 const& leftUppCor,
   Color c=WRITECOLOR, Z i=1);
   //: place text
   // Notice that the left lower corner of the Frame *this is
   // (x=0,y=0) and the right upper corner is (x=1,y=1).
   // leftUppCor is the left upper corner of the rectangular writing
   // box, and the return value gives the right lower corner of it.
   // These positions are derived from the actual font metric
   // of the font Frame::ft_[i]. Thus presently i \in {0,1,2,3,4,5}

void setText(V<Word> const& lines, V<Color> const& c);
   // setting a multi-line text that fits the frame

void setText(V<Word> const& lines){
   Z nL=lines.dim(); V<Color> c(nL,WRITECOLOR); setText(lines,c);
}
   // setting a multi-line text that fits the frame
```

```
   void addText(const Word& text, R x=0.05, R y=0,
       Color c=WRITECOLOR, R h=0);
      // adds text components if their locations are not yet
      // set. Otherwise components of text_ get replaced
      // by components of text.
   void clearText(){ text_=FrmText();}
      // clear
   bool noText()const{ return text_.noText();}
      // returns true if text_ has no content to display

   R textHeight()const;
      // returns a proposal for font height (in pixels) to be used in a
      // frame for text that is ment primarily as a carrier of
      // information. Tries to achieve that smaller frames use
      // the smaller fonts. Presently ~ inf(nCol,nRow) with
      // a factor of proportionaliy hFrac is a static member of
      // Frame and thus gets set in cpmframe.cpp, presently to
      // hFrac=0.0225

// 'modern' form of the functions with arguments of type Sab. If these
// functions would not influence the data element img, they could be
// declared static.
// Their visible action refers to the application window and not to the
// Frame. The signature of these functions is the same as that of
// corresponding functions of Viewport and very similar to that of
// corresponding functions of Img24. In an attempt to unify the
// services of these two classes I sacrificed the variability of line
// style and line width.

   // pixel commands

   void putPixel(Z a, Z b, Color const& c)const
      // puts color c in a meaningful way to screen pixel (a,b)
      // where a,b is understood in the graphics numbering scheme
      // of the screen, thus Sab
{
    img.putPel(a-a0(),b-b0(),c);
}
    void put(Z2 const& Fij, Color const& c)const
      // puts color c in a meaningful way to screen pixel (i,j)
      // where i,j is understood as matrix indexes of the frame (frame =
      // matrix of pixels). Thus i=1,j=1 corresponds to the upper-left
      // corner of the frame
   { img.putPel(Fij.x2-1,Fij.x1-1,c);}

   void putPel(R a, R b, rgb c)const
      // action img according
   {
      img.putPel(a-a0(),b-b0(),c);
```

```
   }

   void setPel(R a, R b, rgb c)const
      // ensures color setting mode
   {
      img.setCol();
      img.putPel(a-a0(),b-b0(),c);
   }

   rgb getPel(R a, R b)const
      // action on img
   {
      return img.getPel(a-a0(),b-b0());
   }

  // line commands

   void line(R a1, R b1, R a2, R b2, rgb c)const;
      // draws a line from (a1,b1) to (a2,b2) (points included),
      // properly clipped to the frame.
      // If thcLin_=true a thick line is drawn

 // area commands

   void fillRectangle(R a1, R b1, R a2, R b2, rgb c)const
      // 1 refers the the upper left corner, 2 to the lower right
      // corner. The color is a solid fill color
   {
      img.fillRectangle(a1-a0(),b1-b0(),a2-a0(),b2-b0(),c);
   }

   void fill(TriAng const& tri, rgb c)const
      //: fill
      // filling a triangle with color c. No restriction to tri.
      // Will be clipped to the imaging window.
   {
      img.fill(tri.shf(shf()),c);
   }

   void fillEllipse(R a1, R b1, R a2, R b2, rgb c)const
      // 1 refers the the upper left corner, 2 to the lower right
      // corner of a minimal rectangle that contains the ellipse.
      // notice that circles are special cases which are not very useful
      // since different scales on the axes make them ellipses anyway.
   {
      img.fillEllipse(a1-a0(),b1-b0(),a2-a0(),b2-b0(),c);
   }

// Addition 2001-4-26:
// pixel commands referring to frame indexing (as opposed to screen
```

```
// indexing. A pixel put loop over the frame is most conveniently written
// as (notice the start with 1)
//    for (Z x=1;x<=dim_x();x++){
//       for (Z y=1;y<=dim_y();y++){
//          Color c=... x...y...;
//          put_xy(x,y,c);
//       }
//    }
// see implementation of void Graph::show(const F<C,C>&,R)
// for actual usage (addition 2001-4-26)
//
   void put_xy(Z x, Z y, Color const& c)const
   {
      putPel(x+pL.x1-1,pL.x2-y+1,c);
   }

   rgb get_xy(Z x, Z y)const{ return getPel(x+pL.x1-1,pL.x2-y+1);}

// consider
// Img24 img=...;
// Frame fr;
// img.display(fr.context(),10,10);
// This will show the image on a pixel by pixel basis with an upper left
// corner placed at graphical position 10,10 of the screen (not the
// window of frame).

   xyxy restrict(xyxy const& rec)const;
      // returns the part of the screen rectangle rec, that belongs to
      // *this in a way that it is directly a subrectangle of the pixel
      // matrix of img.
      // We look at the result as generated in two steps:
      // First *this and rec are considered as rectangles within the
      // screen pixel matrix. We form the set section and obtain a new
      // screen rectangle (which may be void). Then we express the
      // coordinates of this rectangle as relative to the upper left
      // corner of *this and obtain what we want.
};

// tool for autoscaling

Z lad_4(R a, R b, Z  n, R2& x);
   //: ladder
   //  Changes the fourth argument, as the name indicates.
   // Assure a<b. The return-value m and xl=x[0], xu=x[1] are determined
   // such that
   // (1) The numbers xl+k*(xu-xl)/m, k=0,1,2,...m are simple numbers in
   // decimal scientific notation.
   // (2) xl<=a, b<=xu, both inequalities not too far from equality.
   // (3) m not too far from input-value n.
   // So, the function creates a suitable axis division for data ranging
```

```
   // from a to b.

}  // namespace

#endif
```

82 cpmframe.cpp

```
//? cpmframe.cpp
//? Status of work 2008-10-25.
//?
#include <cpmframe.h>
#include <cpms.h>
#include <cpmfontgeneration.h>
#include <cpmimage.h>
//#include <cpmapplication.h>

namespace CpmGraphics{

   using CpmImaging::Rect;
   using CpmImaging::Mask;
   using CpmImaging::Image;
   using CpmImaging::Histogram;
   using CpmImaging::Color;
   using CpmImaging::INVISIBLE;
   using CpmImaging::BLACK;
   using CpmImaging::STEREOBLACK;
   using CpmImaging::WHITE;
   using CpmImaging::WRITECOLOR;
   using CpmImaging::LIGHTGREEN;
   using CpmRootX::order;

   using namespace CpmRoot;
   using namespace CpmSystem;
   using namespace CpmArrays;
   using namespace CpmFunctions;

//////////////////////// class FrmText ////////////////////////////////////

FrmText::FrmText(V<TextCom> const& tcs):tcs_(0)
{
   Z n=tcs.dim();
   S<R2> ps;
   for (Z i=1;i<=n;++i){
      TextCom ti=tcs[i];
      R2 pi=ti.c1();
      bool bi=ps.addAct(pi);
      if (bi) tcs_&=ti;
   }
}

FrmText::FrmText(V<Word> const& vw, Frame const& fr, V<Color> const& c):
tcs_(vw.dim())
{
   using CpmFonts::Font;
```

```
   Z i,j,nL=vw.dim();
   Vl<Font> ft=Frame::ft_;
   Z nF=ft.dim();
   Vo<Z> ht(nF);
      // total minimum height (in pixels) needed for the text
      // in the various available fonts
   Vo<Z> wt(nF);
      // total minimum width (in pixels) needed for the text
      // in the various available fonts
   Z hFr=fr.h();
   Z wFr=fr.w();
   for (i=1;i<=nF;++i){
      Font fi=ft[i-1];
      ht[i]=fi.h()*nL;
      Vo<Z> wi(nL);
      for (j=1;j<=nL;++j){ wi[j]=fi.len(vw[j]);}
      wt[i]=wi.sup();
   }
// Font selection
   V<bool> fit(nF,false);
   for (i=1;i<=nF;++i){
      bool bi=ht[i]<hFr && wt[i]<wFr;
      if (bi) fit[i]=true;
      else break;
   }
   Z i0=1;
   if (!fit[1]) ;
   else if (fit[nF]) i0=nF;
   else{
      while (i0<nF && fit[i0+1]) i0++;
   }
   Font ft0=ft[i0-1];
   R hFont=ft0.h();
   Z h0=ht[i0];
      // maximum width of text for ft0
   Z w0=wt[i0];
      // pure height of text for ft0
   Z dh=hFr-h0;
   Z dw=wFr-w0;
   R x0=0, y0=1; // start position
      // if no space is available
   R hFontRel=hFont/hFr;
   R lineShiftRel=1.25*hFontRel;
   if (dw>0) x0=(0.25*dw)/wFr;
      // leave a left margin if there is space
   if (dh>0) y0=1-(0.25*dh)/hFr;
      // leave an upper margin if there is space
// place the lines from top to bottom
   for (i=1;i<=nL;++i){
      tcs_[i]=X4<R2,Word,Color,R>(R2(x0,y0),vw[i],c[i],hFont);
```

```
      y0-=lineShiftRel;
   }
}

FrmText& FrmText::operator+=(TextCom const& tp)
{
   R2 p=tp.c1();
   Z i=1,n=tcs_.dim();
   bool found=false;
   while (!found && i<=n){
      TextCom ti=tcs_[i];
      if (p==ti.c1()){
         tcs_[i]=tp;
         found=true;
      }
      i++;
   }
   if (!found) tcs_&=tp;
   return *this;
}

FrmText FrmText::operator+(TextCom const& tp)const
{
   FrmText res=*this;
   return res+=tp;
}

////////////// utility functions for class Frame ///////////////////////

namespace{ // anonymus

   inline Z discr(R x){ return (Z)x;}

   void reduce(R& x)
// Auxiliar function which brings the argument back into the the interval
//      [0,1]
{
   static const R tiny=1e-6; // 1./ maximum pixel number to be considered
   static const R upper=1.-tiny;
   if (x<=tiny) x=tiny;
   if (x>=upper) x=upper;
}

   inline void confine(R& x, R low, R upp)
       // inline version of previous function
   {
      if (x<low) x=low;
      if (x>upp) x=upp;
   }
```

```
   R powOf10(R x)
{
   static const R ln10=log(10.);
   return exp(x*ln10);
}

   R cut(R x)
// Returns the nearest number to x, which up to sign and power of 10
// is one of the L-1 first numbers in the array c[L] in the function
// body.
// say c[L]={1,2,5,10}
{
   const Z mL=3;
   cpmmessage(mL,"cut(x) called for x="&cpmwrite(x));
   cpmcheck(x); // the function gets value which often are obtained
      // by complex computations. I sometimes had #INF here.
   Z L=5;
   Vl<R> c(L,10.); // initialized as 10
   c[0]=1.;    // starting with 0 !
   c[1]=2.;
   c[2]=5.;
   c[3]=7.5;
      // The list must be in ascending  order and must contain
      // (L - 1) numbers r 0<r<10
   R y1,yu,yl,yf;
   if (x==0) return 0.;
   Z  s=(x<0 ? -1: 1);
   y1=cpmabs(x);
   R ylog=log10(y1);
   R ylogInt=floor(ylog);
   R yMantisse=ylog-ylogInt;
   R y=powOf10(yMantisse);
   if (!(1<=y)){
      cpmerror("Frame::cut(): !(1<=y)");
   }
   if (!(y<10)){
      cpmerror("Frame::cut(): !(y<=10)");
   }
   Z i=0;
   while (y>c[i]) i++;
   if (i<0 ||i>=L) cpmerror("unexpected failure in function R cut(R)");
   yu=c[i];// yu is the smallest c[k] which is larger or equal to y
   if (i==0){
      yf=yu;
   }
   else{
      if ((i-1)<0 ||(i-1)>=L)
      cpmerror("unexpected failure in function R cut(R)");
      yl=c[i-1];
      yf=(yu-y<y-yl ? yu : yl);
```

```
   }
   // now yf is the c[k] which is nearest to y
   R res=s*yf*powOf10(ylogInt); // sign and exponent are restored
   cpmmessage(mL,"cut(x) done with result="&cpmwrite(res));
   return res;
}

}//namespace

Z lad_4(R a, R b, Z  n, R2& x)
{
   static Word loc("CpmGraphics::lad_4(R,R,Z,R2)");
   Z mL=3;
   CPM_MA
   cpmcheck(a);
   cpmcheck(b);
   const R epsilon=1e-6;
      // introduced 2004-11-24, see place of usage
   if (a>b) CpmRootX::swap<R>(a,b);
   if (n<0) n=-n;
   if (n==0) n=1;

   if (a==b){   // 92-9-7 to avoid error message in cases where function
   // or list is constant
      if (a==0){
         b=1.; a=-1.;
      }
      else{
         R step=0.1*cpmabs(a);
         b+=step; a-=step;
      }
   }
   R nInv=1./n;
   R d=(b-a)*nInv;
   R d_=cut(d); // here is the Pudels Kern
   if (d_==0) cpmerror("Zero division in function ladder");
   R d_inv=1./d_;
   R k_b=ceil(b*d_inv-epsilon);
      // without the epsilon correction it depends on roundoff
      // errors whether b_ becomes too large (2004-11-24)
   R k_a=floor(a*d_inv+epsilon);
      // without the epsilon correction it depends on roundoff
      // errors whether a_ becomes too small
   R b_=k_b*d_;
   R a_=k_a*d_;
   cpmassert(b_>=b,loc);
   cpmassert(a_<=a,loc);
   Z  n_= (Z )(0.5+(b_-a_)*d_inv);
   x.x1=a_;
   x.x2=b_;
```

```
   CPM_MZ
   return (n_==0 ? 1 : n_);
}

//////////////////// class  Frame ////////////////////////////////////////
// notice: colorActive is not static so it does not appear here !!!

bool Frame::xyPixelInitialized=false;
bool Frame::fontInitialized=false;

Z Frame::xPixel_=0;
Z Frame::yPixel_=0;
R Frame::xPixelInv=0;
R Frame::yPixelInv=0;
R Frame::hFrac=0.0225;
   // see function textHeight()
R Frame::yText=0.15;
Color Frame::defaultColorActive=WHITE;

Vl<CpmFonts::Font> Frame::ft_;

xyxy Frame::scrRec;
   // screen rectangle, static member. Will be meaningfully
   // initialized in Frame::iniStatics(void).

Z Frame::imageCounter=0;

static const Z N_EXCESS=40;
static const R IMMENSE=1e99;

Z Frame::fontSelector( R h)
{
   if (h<=10.1) return 0;
   else if (h<=12.1) return 1;
   else if (h<=14.1) return 2;
   else if (h<=16.1) return 3;
   else if (h<=18.1) return 4;
   else return 5;
}

Z2  Frame::get_pU(void)const{ return pU;}

Z2  Frame::get_pL(void)const{ return pL;}

void Frame::updateRowCol(void)
{
   nRow=pL.x2-pU.x2+1; // The lower corner has the larger y-component
      // since y is counted from above !
   nCol=pU.x1-pL.x1+1;
   cpmassert(nCol>0 && nRow>0,"void Frame::updateRowCol(void)");
```

```
   nRowInv=1./nRow;
   nColInv=1./nCol;
   ColSty sty=img.getSty();
   img=Img24(nCol,nRow,colorActive,sty);
}

void Frame::stereo()
{
   img.stereo();
   colorActive=Color(STEREOBLACK);
}

void Frame::updateFromDiscrete()
{
   pU.x1=pL.x1+nCol-1;
   if (pU.x1>=xPixel_) pU.x1=xPixel_-1;
   if (pU.x1<0) pU.x1=0;

   pU.x2=pL.x2-nRow+1;
   if (pU.x2>=yPixel_) pU.x2=yPixel_-1;
   if (pU.x2<0) pU.x2=0;

   nRow=pL.x2-pU.x2+1; // The lower corner has the larger y-component
      // since y is counted from above !
   nCol=pU.x1-pL.x1+1;
   cpmassert(nCol>0 && nRow>0,"void Frame::updateFromDiscrete(void)");
   nRowInv=1./nRow;
   nColInv=1./nCol;

   cL.x1=pL.x1*xPixelInv;
   cU.x1=(pU.x1+1)*xPixelInv;
   cL.x2=1-(pL.x2+1)*yPixelInv;
   cU.x2=1-pU.x2*yPixelInv;
   img.resize(nCol,nRow);  // paints the image with default color
}

Z2  Frame::SxyToSab(R2 c)
{
     static const R low=1e-6;
   static const R upp=1-low;
   R x=c.x1, y=1.-c.x2;
   confine(x,low,upp); // inline, thus fast
   confine(y,low,upp);
   return Z2((Z)(x*xPixel_),(Z)(y*yPixel_)); // efficient
}

Z2  Frame::discrete(R2& pLower, const R2& p, R2& pUpper)
   {
      Word loc("Frame::discrete(R2,R2,R2)");
      R2 q=p;
```

```
      reduce(q.x1); reduce(q.x2);
      if (xPixel_<=0) xPixel_=1;
      if (yPixel_<=0) yPixel_=1;
      cpmassert(xPixel_!=0,loc);
      cpmassert(yPixel_!=0,loc);
      R Dx=1./xPixel_;
      R Dy=1./yPixel_;
      R nx=q.x1*xPixel_;
      R ny=q.x2*yPixel_;
      Z2 z;
      z.x1=discr(nx); // result never <= -1 negative or >=xPixel_
         // due to reduce
      z.x2=discr(ny);
      pLower.x1=z.x1*Dx;
      pLower.x2=z.x2*Dy;
      pUpper.x1=pLower.x1+Dx;
      pUpper.x2=pLower.x2+Dy;
      z.x2=yPixel_-z.x2-1; // since z.x2 counts from above
      return z;
   }

Z2  Frame::getiFjF(const R2& xFyF)const
   // given coordinates of a point in the frame system, the
   // matrix indexes of the pixel to which the point belongs
   // is given
{
   R x=xFyF.x1, y=1.-xFyF.x2;
   reduce(x); reduce(y);
    Z i,j;
   i=(Z)(y*nRow); j=(Z)(x*nCol);
   return Z2(i+1,j+1);
}

void  Frame::setxy(R x, R y, R dx, R dy)
// Helps to consistently set cL.x, cU.x, cL.x2, cU.x2
// The pL, pU, nRow, nCol, nRowInv, nColInv are set accordingly
{
   R x_=x+dx, y_=y+dy;
   order(x,x_);
   order(y,y_);
   R2 cUpper,cLower;
   R2 cLow(x,y);
   R2 cUpp(x_,y_);
   pL=discrete(cL,cLow,cUpper);  // setting pL and cL (cUpper not used)
   pU=discrete(cLower,cUpp,cU);  // setting pU and cU (cLower not used)
   updateRowCol();
   if (nCol==1 || nRow==1){
      Word mess=" Frame is degenerated to a line or a point";
      cpmwarning("Frame",mess);
   }
```

```
}

void  Frame::setpLpU(Z2 pLIn, Z2 pUIn)
   // Helps to consistently set cL, cU. Corners of the frame are
   // set in terms of screen pixel numbers.
   // Needed to define a precise partition of a frame into subframes
   // without any overlap or gaps (96-11-28)
{
   Z iL=pLIn.x1;
   Z jL=pLIn.x2;
   Z iU=pUIn.x1;
   Z jU=pUIn.x2;
   order(iL,iU);  // i horizontal distance from left screen column
   order(jU,jL); // notice j : vertical distance from upper screen line.
      // thus jL must be larger than jU
   if (iL<0) iL=0;
   if (iU>=xPixel_) iU=xPixel_-1;
   if (jU<0) jU=0;
   if (jL>=yPixel_) jL=yPixel_-1;
   pL.x1=iL; // (*)
   pL.x2=jL;
   pU.x1=iU;
   pU.x2=jU;
   cL=SabToSxy(pL); // was placed directly after (*) till 2002-02-08
      // and thus did cause mysterious malfunction of the Frames
      // constructors.
   cU=SabToSxy(pU);
   updateRowCol();
}

void  Frame::iniStatics(void)
// initializing graphics for the first frame
{
   if (!fontInitialized){
      Word fontfile=Message::getFontDir()&"/cpmfont1.txt";
      IFileStream in(fontfile);
      cpmmessage("loading C+- fonts started");
      fontInitialized=ft_.scanFrom(in());
      cpmmessage("loading C+- fonts done");
   }
   if (!xyPixelInitialized && xPixel_>0 && yPixel_>0){
      // since this is a static member of the class it can be
      // manipulated by member functions
      scrRec=xyxy(xy(0,0),xy(xPixel_,yPixel_));
      xPixelInv=1./xPixel_;
      yPixelInv=1./yPixel_;
      xyPixelInitialized=true;
   }
}
```

```
void Frame::setStaticSize(Z width, Z height)
{
   xPixel_=width;
   yPixel_=height;
   xyPixelInitialized=false;
   iniStatics();
}

void  Frame::iniFrame(R x, R y, R dx, R dy, const Color& color)
   // tool for defining constructors without code repetition
{
   Z mL=3;
   Word loc("Frame::iniFrame(R, R, R, R, Color)");
   CPM_MA
   iniStatics();
   setxy(x, y, dx, dy);
   colorActive=color;
   if (color.stereo()) img.stereo();
   CPM_MZ
}

Frame::Frame(R x, R y, R dx, R dy, const Color& color)
// generic constructor
{
   if (cpmverbose>=4)
    cpmmessage("Frame(R x, R y, R dx, R dy, const Color& color) called");
   iniFrame(x,y,dx,dy,color);
   img=Img24(nCol,nRow,color);
   if (color.stereo()) img.stereo();
   img.display(a0(),b0());
      // there should be nothing wrong with showing a new Frame
   if (cpmverbose>=4)
      cpmmessage("Frame(R x, R y, R dx, R dy, const Color& color) done");
}

Frame::Frame(const Img24& im, R x, R y, bool showIt)
{
   Word loc("Frame(Img24,R,R,bool)");
   Z mL=4;
   CPM_MA
   iniStatics();
   Z nx=im.getCol();
   Z ny=im.getLin();
   R dx=((R)nx)/xPixel_;
   R dy=((R)ny)/yPixel_;
   cpmassert(x>=0&&x<=1.,loc);
   cpmassert(y>=0&&y<=1.,loc);
   setxy(x,y,dx,dy);
   Z2 pL0=pL;
   Z2 pU0=pL0+Z2(nx-1,-ny+1);
```

```
   setpLpU(pL0,pU0);
   img=im;
   cpmassert(nRow==ny,loc);
   cpmassert(nCol==nx,loc);
   if (showIt) display();
   CPM_MZ
}

Frame::Frame(void)
// default constructor, no painting
{
   if (cpmverbose>=4) cpmmessage("Frame() called");
   Color c=INVISIBLE; // this avoids the painting action
   iniFrame(0,0,1,1,c); // so far, colorActive=INVISIBLE
   colorActive=defaultColorActive;
   img=Img24(nCol,nRow,colorActive);
   if (cpmverbose>=4) cpmmessage("Frame() done");
}

Frame::Frame(const Color& c)
{
   if (cpmverbose>=4) cpmmessage("Frame(const Color&) called");
   iniFrame(0,0,1,1,c);
   img=Img24(nCol,nRow,c);
   if (c.stereo()) img.stereo();
   if (cpmverbose>=4) cpmmessage("Frame(const Color&) done");
}

Frame::Frame(Z code)
   // Constructor for convenient placing of a frame
   // The screen is divided in quadrants denoted in matrix-style
   // 11 12
   // 21 22
   // Posible position codes are 11, 12, 21, 22, 1112 (for upper
   // half part) 2122 (lower half part), 1121 (left half part)
   // 1222 (right part)
{
   if (cpmverbose>=4) cpmmessage("Frame(Z code) called");
   iniStatics();
   switch (code){
      case 11:   setxy(0.0, 0.5, 0.5, 0.5); break;
      case 12:   setxy(0.5, 0.5, 0.5, 0.5); break;
      case 21:   setxy(0.0, 0.0, 0.5, 0.5); break;
      case 22:   setxy(0.5, 0.0, 0.5, 0.5); break;
      case 1112: setxy(0.0, 0.5, 1.0, 0.5); break;
      case 2122: setxy(0.0, 0.0, 1.0, 0.5); break;
      case 1121: setxy(0.0, 0.0, 0.5, 1.0); break;
      case 1222: setxy(0.5, 0.0, 0.5, 1.0); break;
      default:   setxy(0.0, 0.0, 1.0, 1.0); break;
   }
```

```
   colorActive=defaultColorActive;
   if (colorActive.visible()) paint(colorActive);
   img=Img24(nCol,nRow,colorActive);
   if (cpmverbose>=4) cpmmessage("Frame(Z code) done");
}

Rect  Frame::getRectangle(void)const
   // returns the rectangle from which the Frame 'could have been
   // created'.
{
   return Rect(cL.x1,cU.x1,cL.x2,cU.x2);
}

Frame::Frame(const Rect& rec)
   // constructor from a rectangle in the whole screen
{
   R x=rec.getxL();
   R y=rec.getyL();
   R dx=rec.getxU()-x;
   R dy=rec.getyU()-y;
   iniFrame(x,y,dx,dy,defaultColorActive);
   img=Img24(nCol,nRow,colorActive);
}

Frame::Frame(const  Frame& fr, const Rect& rec)
   // constructor from a rectangle of the rectangle given by fr
{
   Rect recfr=fr.getRectangle();
   Rect recfinal=recfr.sub(rec);
   R x=recfinal.getxL();
   R y=recfinal.getyL();
   R dx=recfinal.getxU()-x;
   R dy=recfinal.getyU()-y;
   iniFrame(x,y,dx,dy,defaultColorActive);
   img=Img24(nCol,nRow,colorActive);
}

Z  Frame::ix(R xF)const
// 0<= xF <= 1 within the frame. Return value is the pixel number on
// screen which corresponds to to the coordinate xF within the frame.
{
   if (xF<0) return pL.x1;
   else if (xF>1) return pU.x1;
   else{
      R a=cL.x1+(cU.x1-cL.x1)*xF;
      return discr(xPixel_*a);
   }
}

Z  Frame::jy(R yF)const
```

```
// 0<= yF <= 1 within the frame. Return value is the pixel number on
// screen which corresponds to to the coordinate yF within the frame.
// Notice that the meaning of pixel numbers is determined by the syntax
// of the putPel function. The corners of the screen are
// putPel(0,0) (upper-left)        putPel(xPixel_,0) (upper-right)
// putPel(0,yPixel_) (lower-left)   putPel(xPixel_,yPixel_) (lower-right)
{
   if (yF<=0) return pL.x2;
   else if (yF>=1) return pU.x2;
   else{
      R b=cL.x2+(cU.x2-cL.x2)*yF;
      return discr(yPixel_*(1-b));
   }
}

void  Frame::paint(const Color& c)
{
   if (c.visible()) img.paint(c);
}

void Frame::setAspRatHoldWidth(R asp)
{
   Word loc("setAspRatHoldWidth");
   cpmassert(asp>0,loc);
   nRow=(Z)(0.5+nCol/asp);
   updateFromDiscrete();
}

void Frame::setAspRatHoldHeight(R asp)
{
   Word loc("setAspRatHoldHeight");
   cpmassert(asp>0,loc);
   nCol=(Z)(0.5+nRow*asp);
   updateFromDiscrete();
}

void Frame::cut(R asp, Z dyPixel)
{
   Z mL=4;
   Word loc("Frame::cut(R,Z)");
   cpmmessage(mL,loc&" started");
   cpmassert(asp>0,loc);
   cpmassert(dyPixel>0,loc);
   R4 xydxdy=getxy();
   R x=xydxdy[1];
   R y=xydxdy[2];
   R dx=xydxdy[3];
   R dy=xydxdy[4];
   R dy0=(1.*dyPixel)/yPixel_;
      // made such that dy0*yPixel_=dyPixel
```

```
   R a=asp*screenAspRatInv();
   R dx0=a*dy0;
      // made such that dx0*xPixel_/(dy0*yPixel_)=asp
   R f1=dx/dx0;
   R f2=dy/dy0;
   R f3=1.;
   R f=cpminf(f1,f2,f3);
   R dx1=dx0*f;
   R dy1=dy0*f;
   R x1=x+0.5*(dx-dx1);
   R y1=y+0.5*(dy-dy1);
   setxy(x1,y1,dx1,dy1);
   img=Img24(nCol,nRow,colorActive);
   cpmmessage(mL,loc&" done");
}

void Frame::line(R a1, R b1, R a2, R b2, rgb c)const
{
   R x1=a1-a0(),y1=b1-b0(),x2=a2-a0(),y2=b2-b0();
   img.line(x1,y1,x2,y2,c);
}

namespace{
   void lineArtPrc(Img24& img)
   { // using Image<R> as workhorse, quite efficient
      const R limL=193;
      const R limU=255;
      // these constants are adjusted to the properties
      // of the specific binomial mask to be used.
      Image<R> im(img);
      Mask m=Mask::bin3();
      im=im.conv(m);
      R2 a(limL,limU);
      R2 b(0,255);
      bool linear=true;
      im=im.transformToneScale(R_Func::transit(a,b,linear));
      img=im.toImg24();
   }
}//namespace

Word Frame::dis(bool file,bool lineArtProcessing)const
{
   Z mL=3;
   Word loc("Frame::dis(bool,bool)");
   CPM_MA
   if (!noText()){
      Z nt=text_.dim();
      for (Z i=1;i<=nt;++i){
         TextCom ti=text_[i];
         R2 xi=ti.c1();
```

```
         xy pi=FxyToFab(xi);
         Word wi=ti.c2();
         ColRef ci=ti.c3();
         R hi=ti.c4();
         Z fi=fontSelector(hi);
         ft_[fi].wrt(wi,pi,img,ci);
      }
   }
   if (lineArtProcessing) lineArtPrc(img);
   img.display(a0(),b0());
   Word res;
   if (file){
      imageCounter++;
      Word fn="frame_"&cpm(imageCounter,3);
      if (Message::getAppRunId()){
         Word app("_");
         app&=Message::getRunId();
         fn&=app;
      }
      res=img.write(fn);
   }
   CPM_MZ
   return res;
}

Word Frame::write(const Word& file)const
{
   Word res;
   Z mL=1;
   Word loc("Frame::write(const Word& file)");
   cpmmessage(mL,loc&" "&file&" started");
   res=img.write(file);
   cpmmessage(mL,loc&" "&file&" done");
   return res;
}

Word Frame::write(void)const
{
   Word res;
   Z mL=1;
   Word loc("Frame::write(void)");
   CPM_MA
   res=img.write();
   CPM_MZ
   return res;
}

void  Frame::clear()
{
   Z mL=3;
```

```
   Word loc("Frame::clear()");
   CPM_MA
   clearText();
   paint(colorActive);
   CPM_MZ
}

void Frame::fullClear()
{
   Z mL=3;
   Word loc("Frame::fullClear()");
   CPM_MA
   img.setColor(Color(BLACK));
   img.clear();
   CPM_MZ
}

bool  Frame::prnOn( ostream& str) const
{
   cpmwat;
   cpmp(colorActive);
   cpmp(cL);
   cpmp(cU);
   return true;
}

bool  Frame::scanFrom (istream& str)
{
   cpms(colorActive);
   cpms(cL);
   cpms(cU);
   R x=getx();
   R y=gety();
   R dx=getdx();
   R dy=getdy();
   setxy(x,y,dx,dy);
   return true;
}

R Frame::textHeight()const
{
   const R sizeLimitUpper=24.;
   const R sizeLimitLower=10.;
   Z n=CpmRootX::inf<Z>(nRow,nCol);
   R res=cpmround(hFrac*n);
   if (res>sizeLimitUpper) res=sizeLimitUpper;
   if (res<sizeLimitLower) res=sizeLimitLower;
   return res;
}
```

```
void Frame::setText(Word const& text, R x, R y, Color c, R h)
{
   if (y<=0) y=yText;
   if (h<=0) h=textHeight();
   TextCom tc(R2(x,y),text,c,h);
   setText(V<TextCom>("",tc));
}

R2 Frame::plcText(Word const& text, R2 const& leftUppCor, Color c, Z ft)
{
   using CpmFonts::Font;
   R x=leftUppCor[1];
   R y=leftUppCor[2];
   if (x<0) x=0;
   if (x>=1) return leftUppCor;
   if (y>1) y=1;
   if (y<=0) return leftUppCor;
   if (ft<ft_.b()) ft=ft_.b();
   if (ft>ft_.e()) ft=ft_.e();
   Font font=ft_[ft];
   R wi=font.len(text);
   R hi=font.h();
   R wr=wi/w();
   R hr=hi/h();
   R xf=x+wr;
   R yf=y-hr;
   TextCom tc(R2(x,y),text,c,hi);
   setText(V<TextCom>("",tc));
   return R2(xf,yf);
}

void Frame::setText(V<Word> const& lines, V<Color> const& c)
{
   FrmText ft(lines,*this,c);
   setText(ft);
}

void Frame::addText(FrmText const& text)
{
   Z n=text.dim();
   for (Z i=1;i<=n;++i) text_+=text[i];
}

void Frame::addText(const Word& text, R x, R y, Color c, R h)
{
   if (y<=0) y=yText;
   if (h<=0) h=textHeight();
   TextCom tc(R2(x,y),text,c,h);
   addText(V<TextCom>("",tc));
}
```

```
} // namespace
```

83 cpmframes.h

```
//? cpmframes.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_FRAMES_H_
#define CPM_FRAMES_H_
/*

Purpose: making a class of matrix-like subdivided frames

*/

#include <cpmframe.h>

namespace CpmGraphics{

   class Frames: public CpmArrays::VV<Frame>{ // array of Frame objects

   typedef CpmArrays::VV<Frame> Base;

public:

   Frames():Base(){}

   Frames(Frame const& f, Z m, Z n, R fill=0.95);
      // The frame f will be subdivided into a (m,n)-matrix
      // of non-overlapping subframes. In order to avoid overlap
      // of the subframes factor fill may be set to values slightly
      // below 1

   Frames(Frame const& f, const CpmArrays::Vo<R>& horizDiv,
      const CpmArrays::Vo<R>& vertDiv, R fill);
      // The frame f will be subdivided into a matrix
      // of non-overlapping subframes where the subdivision is given by
      // the real lists horizDiv, vertDiv
      // 0<= horizDiv[i]<=1; 0<= vertDiv[i]<=1;
      // List need not be ordered since done internally

   Frames(Frame const& f, Z n, R fill=0.95);
      // system of n frames reasonably distributed over
      // the viewing rectangle of f

   Frame& operator()(Z i);
// defining and explaining code:
/*
   { Z2 mtl=linearToMatrix(i,m,n);
   return (*this)[mtl.x1][mtl.x2];}
*/
```

```
/******** a typical application of the previous two functions ********
   Frame fr;
   fr.paint(Color(LIGHTGREEN));
   Frames frs(fr,n);
   V<Graph> vgr(n);
   for (Z i=1;i<=n;i++){
      vgr[i]=Graph(frs(i));
      vgr[i].setWithOrigin(1);
      vgr[i].show(res[i]);
      vgr[i].display(true);
   }
*/
};

} // namespace

#endif
```

84 cpmframes.cpp

```
//? cpmframes.cpp
//? Status of work 2008-10-25.
//?
#include <cpmframes.h>
#include <cpmvectormore.h>

using namespace CpmArrays;
using namespace CpmGraphics;

namespace{
   I_Vector chain(Z start, Z end, Z n)
   // a integer-valued chain starting with start ending at end
   // having n members and being equidistant as good as
   // integer-valuedness allows
{
   Word loc("I_Vector chain(Z start, Z end, Z n)");
   I_Vector res(n);
   cpmassert(n>1 && end > start,loc);
   R step=( end-start)/(n-1.);
   res[1]=start;
   res[n]=end;
   for (Z i=2;i<n;i++) res[i]=(Z)(0.5+start+step*(i-1));
   return res;
}

   I_Vector chain2(Z start, Z end, const Vo<R>& div)
   // a integer-valued chain starting with start ending at end
   // and growing according to the sequence div which normally will
   // start at 0 and will end with 1. If not, start and end may not be
   // reached.
   // 0<=div[i]<=1 assumed. Order not needed since made internally.
{
   Word loc("I_Vector chain2(...)");
   Z d=div.dim();
   cpmassert(d>1,loc);
   cpmassert( end > start,loc);
   V<R> divO=div.increasingCopy();
   cpmassert(divO[1]>=0.,loc);
   cpmassert(divO[d]<=1.,loc);
   V<R> divD(d);
   R dx=R(end-start);
   Z i,val;
   for (i=1;i<=d;i++) divD[i]=start+dx*divO[i];
   I_Vector res(d);
   for (i=1;i<=d;i++){
      val=(Z)(divD[i]+0.5);
      if (val<start) val=start;
```

```
      if (val>end) val=end;
      res[i]=val;
   }
   return res;
}

}
/********** implementation of class Frames ****************************/


Frames::Frames(const Frame& f, Z m, Z n, R fill):VV<Frame>(m,n,Frame(f))
   // The frame f will be subdivided into a (m,n)-matrix
   // of non-overlapping subframes
{
   f.display(); // experiment
   Z2 fpL=f.pL;
   Z2 fpU=f.pU;
   R cFill=1.-fill;
   Z bx=(Z)(0.5+(cFill*f.nCol)/(2.*n));
   Z by=(Z)(0.5+(cFill*f.nRow)/(2.*m));
   I_Vector horiz=chain(fpL.x1,fpU.x1,n+1);
   I_Vector vert=chain(fpU.x2,fpL.x2,m+1);

   Z i,j;
   Z2 pl,pu;

   for (i=1;i<=m;i++){
      pu.x2=vert[i]+by;
      pl.x2=vert[i+1]-1-by;
      for (j=1;j<=n;j++){
         pl.x1=horiz[j]+bx;
         pu.x1=horiz[j+1]-1-bx;
         ((*this)[i][j]).setpLpU(pl,pu);
      }
   }
}

Frames::Frames
(const Frame& f, const Vo<R>& horizDiv, const Vo<R>& vertDiv, R fill):
VV<Frame>(vertDiv.dim()-1,horizDiv.dim()-1)
   // The frame f will be subdivided into a matrix
   // of non-overlapping subframes where the subdivision is given by the
   // real lists horizDiv, vertDiv
   // 0<= horizDiv[i]<=1; 0<= vertDiv[i]<=1;
   // List need not be ordered since done internally
{
   f.display(); // experiment
   Z2 fpL=f.pL;
   Z2 fpU=f.pU;
   Z m=vertDiv.dim()-1;
```

```
   Z n=horizDiv.dim()-1;
   R cFill=1.-fill;
   Z bx=(Z)(0.5+(cFill*f.nCol)/(2.*n));
   Z by=(Z)(0.5+(cFill*f.nRow)/(2.*m));
   I_Vector horiz=chain2(fpL.x1,fpU.x1,horizDiv);
   I_Vector vert=chain2(fpU.x2,fpL.x2,vertDiv);
   Z i,j;
   Z2 pl,pu;

   for (i=1;i<=m;i++){
      pu.x2=vert[i]+by;
      pl.x2=vert[i+1]-1-by;
      for (j=1;j<=n;j++){
         pl.x1=horiz[j]+bx;
         pu.x1=horiz[j+1]-1-bx;
         ((*this)[i][j]).setpLpU(pl,pu);
      }
   }
}

Frame& Frames::operator()(Z i)
{
   Z2 mtl=linearToMatrix(i,dim1(),dim2());
   return (*this)[mtl.x1][mtl.x2];
}

Frames::Frames(Frame const& f, Z n, R fill)
{
   Z2 vzn=CpmAlgorithms::genSqrRoot(n);
   *this=Frames(f,vzn.x1,vzn.x2,fill);
}
```

85 cpmgraph.h

```
//? cpmgraph.h
//? Status of work 2008-10-25.
//?
#ifndef CPMGRAPH_H_
#define CPMGRAPH_H_
/*

Purpose: Declaration of graphics class Graph

*/
#include <cpmframe.h>
#include <cpmrmatrix.h>
#include <cpmr2func.h>

// Class Graph is derived from class Frame. An instance of class Graph
// represents a rectangular part of a two-dimensional object space
// which is mapped in a simple linear axis to axis manner onto a
// frame on screen. Positions in object space are referenced by
// instances of class R2, and the corresponding pixel position
// (instance of class Z2) is given via the member function
//
//               Z2 toPixel(const R2& p)

namespace CpmGraphics{

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::C;
   using CpmRootX::B;
   using CpmLinAlg::Z2;
   using CpmLinAlg::Z3;
   using CpmLinAlg::R2;
   using CpmLinAlg::R3;

   using CpmArrays::R_Matrix;
   using CpmArrays::R_Vector;
   using CpmArrays::V;
   using CpmArrays::X2;
   using CpmArrays::X3;

   using CpmFunctions::F;
   using CpmFunctions::R_Func;
   using CpmFunctions::R2_Func;
   using CpmFunctions::RR;

   using CpmImaging::Color;
   using CpmImaging::BLACK;
```

```
   using CpmImaging::WHITE;
   using CpmImaging::RED;
   using CpmImaging::GREEN;
   using CpmImaging::BLUE;
   using CpmImaging::LIGHTBLUE;
   using CpmImaging::rainBow;
   using CpmImaging::Response;

   using CpmGeo::Iv;

   enum SitRec{LOW,UPP,RIG,LEFT,CEN};
      // denotes the sites of a rectangle (such as the screen
      // window) lower (site), upper, right-hand, left-hand
      // 2006-02-14 added CEN for center, which -although
      // not being a site of the rectangle- turned out to be a
      // useful parameter for methods dealing with rectangles.

//////////////////// struct Fig ////////////////////////////////////////

   struct Fig{ // relevant for making Latex figures
      // collects the
      Word tit; // title of the figure
      Word abs; // name of the quantity shown on x-axis
      V<Word> ord; // name of the quantities shown on y-axis
   };

//////////////////////////////// class Graph ////////////////////////////
class Graph: public Frame{// Frame as rectangular part of object space

// independent data:
   Color gridColor;
   Color gridTextColor;
   Color graphColor;

   Z withOrigin;
      // withOrigin!=0 controls autoscaling such that the origin
      // is always on the diagram
   Z newGrid;
      // only =0 or !=0 matters. If it is 0 (should be an exception) one
      // let the grid remain unchanged and draw curves on the frame with
      // a meaning of object space wich is no longer in agreement with
      // the grid and the grid text.
   B autoScale;
      // Influences show(..) such that CL CU are adjusted to the data to
      // be represented prior to drawing them.
      // 'autoscaling'
      // The change of CL and CU remains till new values are assigned
      // by new action.

   B sclMem;
```

```
      //: scale memory, used in function mark(V< V<R2> >,...)

   B writeOnShow;
      // if this is true, calling any of the show-functions
      // will create an image

   B noGridText;
      // If this is true, graphics supresses the textual
      // indication of x-range and y-range

   B noGrid;
      // If this is true, graphics supresses the grid

   V<Word> gridText;
      // since 2005-04-14, we have V<Word> instead of Word to allow
      // multi-line grid text

   R2 gridTextPosition;
      // text position is understood in the Frame coordinate system
      // (xF,yF) i.e. (0,0) is left lower corner, (1,1) is right upper
      // corner

   R1 gridTextHeight_;
      // height of grid text in appr. pixel

   R2 CL;   // lower left corner of the rectangular domain in
         // object space
   R2 CU;  // upper right corner of the rectangular domain in
         // object space

// The following four quantities are simple derivates from
// the former primary geometric data. They are chosen to be data
// members since they are needed in the implementation of
// member functions in many places. Thus they had to be calculated
// many times for a single graph.

   R dX; // x-range in object space dX=CU.x1-CL.x1

   R dY; // y-range in object space dY=CU.x2-CL.x2

   R dXInv; // inverse of dX

   R dYInv; // inverse of dY

// methods

   void iniGraph(R Xl, R Yl, R dX, R dY);

   bool updatedXdY(void);
   // updates dX, dY, dXInv, dYInv if CL or/and CU have been changed.
```

```
   // Returns false if dX or dY would be <=0

public:
   typedef Graph Type;

   CPM_IO
      // It is not the intention of this I/O operations
      // to save the content of video memory, and other content data

// constructors and assignement

   Graph(R xl, R yl, R dx, R dy, const Color& color=defaultColorActive,
   R Xl=0., R Yl=0., R dX=1., R dY=1.);

   Graph(void);

   explicit Graph(const Color& c);
      // is Graph() except of Frame::colorActive=c

   Graph(const Frame& f);

   Graph(Frame const& fr, R rimRel);
      // *this will be placed within fr so that a rim of the same
      // width in x- and y-direction will arise. In x-direction,
      // this width is a fraction rimRel of frame width.

   Word nameOf()const{ return Word("Graph");}

// setting functions

   void setWriteOnShow(bool wos){ writeOnShow=B(wos);}
   void setNoGridText(bool ngt){ noGridText=B(ngt);}
   void setNoGrid(bool ng){ noGrid=B(ng);}
   void setGridColor(const Color& c);
   void setGridTextColor(const Color& c);
   void setGraphColor(const Color& c);
   void setWithOrigin(Z);
      // setWithOrigin(1) controls autoscaling such that the origin
      // is always within the diagram
   void setNewGrid(Z);
   void setAutoScale(bool ac){ autoScale=B(ac);}
      // enable auto scaling
   void setSclMem_(bool sm){ sclMem=B(sm);}
      // enable auto scaling
   void setGridText(Word a);
   void setGridTextPosition(const R2& pos);
   void setGridTextPosition(R xText, R yText);
      // 0<=xText<=1, 0<=yText<=1, sets position of left-lower corner
      // of grid text window within the frame of the Graph
   R2 getGridTextPos()const{ return gridTextPosition;}
```

```
      //: get grid text position
   void setGridTextPos_(R2 const& pos){setGridTextPosition(pos);}
      //: set grid text position
   R getGridTextHeg()const{ return gridTextHeight_;}
      //: get grid text height
   void setGridTextHeg_(R h){ gridTextHeight_=h;}
      //: set grid text height

   bool  setXY(R X, R Y, R dX1, R dY1);
      // sets the corners in object space

// modern style making use of class Iv (interval)
   bool setX(Iv iv);
      // sets the corners in object space X direction according to the
      // interval iv
   bool setY(Iv iv);
      // sets the corners in object space Y direction according to the
      // interval iv. Also sets autoScale to 0.
   bool setXY(Iv ivX, Iv ivY);
      // sets the corners in object space according to the
      // intervals ivX, ivY. Also sets autoScale to 0.
// some getting functions
   Color getGridColor()const{ return gridColor;}
   Color getGridTextColor()const{ return gridTextColor;}
   Color getGraphColor()const{ return graphColor;}
   bool getWriteOnShow()const{ return writeOnShow;}
   Iv getIvX()const{ return Iv(CL.x1,CU.x1,true);}
   Iv getIvY()const{ return Iv(CL.x2,CU.x2,true);}
   V<R> centersX()const{ return getIvX().centers(nCol);}
   V<R> centersY()const{ return getIvY().centers(nRow);}

// transformation between coordinates (xF,yF) in frame system and
// (xG,yG) in object space of the graph

   R2 get_xFyF(const R2& xGyG)const;

   R2 get_xGyG(const R2& xFyF)const;

// my old graphics functions with new interface, i.e as member functions
// of class Graph

Z   to_i(R x, Z& excess)const; // to i
// returns the column-number (\in {0,...,nCol-1} ) to which a point p
// belongs, that is known to have a x-coordinate in the object space
// of Graph *this given by number x. Column numbers increase from left to
// right. After call excess will have the value 0 if x was within the
// object space range, +1 if x was too large, and -1 if it was too low.

Z   to_j(R y, Z& excess)const; // to j
// returns the row-number (\in {0,...,nRow-1} ) to which a point p
```

```
// belongs, that is known to have a y-coordinate in the object space
// of Graph *this given by number y. Row numbers increase from
// above to below. Thus the output of this and the previous function
// give directly the pixel location in img that corresponds to
// p(x,y). Thus a call img.putPel(i(p[1]),j(p[2]) marks a point p
// characterized by object space coordinates. After call excess will have
// the value 0 if x was within the object space range, +1 if x was too
// large, and -1 if it was  too low.

R to_i_(R x)const{ return (x-CL.x1)*dXInv*nCol;}
   // new fast version to to_i taking into account that now pixel access
   // is by R
R to_j_(R y)const{ return (CU.x2-y)*dYInv*nRow;}
   // new fast version of to_j

Z   iX(R x)const;
// x<=XL corresponds to left boundery of the Frame
// x>=XU corr. to right bound. of Frame

Z   jY(R y)const;
// y<=YL corresponds to the lower boundery of the Frame
// y>=YU corresponds to the upper boundery of the Frame

R   iX_(R x)const;
// x<=XL corresponds to left boundery of the Frame
// x>=XU corr. to right bound. of Frame

R   jY_(R y)const;
// y<=YL corresponds to the lower boundery of the Frame
// y>=YU corresponds to the upper boundery of the Frame

R2 natTrn(R2 const& p)
//: natural transformation
// See implementation of fillTriangle for the typical usage
{ return R2(iX_(p.x1),jY_(p.x2));}

Z   iX0(R x)const;
// x=XL corresponds to left boundery of the Frame
// x=XU corr. to right bound. of Frame
// for x<XL or x>XU we return -1

Z   jY0(R y)const;
// y=YL corresponds to the lower boundery of the Frame
// y=YU corresponds to the upper boundery of the Frame
// for y<YL or y>YU we return -1

Z2 toPixel(const R2& p)const;
// maps a point from object space into a pixel (discrete position on
// screen)
```

```
R2 toObjectSpace(const Z2& z )const;
// maps a discrete pixel position into object space. More precisely it is
// the center of the pixel which is taken.

// access fuctions to corner coordinates in object space

R getxLeft(void)const { return CL.x1;}
R getxRight(void)const{ return CU.x1;}
R getyLower(void)const{ return CL.x2;}
R getyUpper(void)const{ return CU.x2;}
R get_dX(void)const{ return dX;}
R get_dY(void)const{ return dY;}

R getX(void)const{ return CL.x1;}
R getY(void)const{ return CL.x2;}
R getdX(void)const{ return CU.x1-CL.x1;}
R getdY(void)const{ return CU.x2-CL.x2;}

Iv ranX(void)const{ return Iv(CL.x1,CU.x1);}
   // range X

Iv ranY(void)const{ return Iv(CL.x2,CU.x2);}
   // range Y

// C+- names for manipulating the autoScale parameter
void setAutoScl_(bool b){ autoScale=B(b);}
//: set auto scale

// underscore since this is a mutating function
bool getAutoScl()const{ return autoScale;}
//: get auto scale

// changing the size in object space of the window

bool scale(R magnification);
   // argument >1 means that the rectangle in object space gets enlarged
   // by a factor 'magnification' in linear dimensions. The center of the
   // window remains fixed in object space. The display size is not
   // affected.
   // If the argument is < 1, the rectangle in object space gets reduced.

bool scaleY(R mag);
   // just as function scale, but acts only on the Y-range

void shift(R sx, R sy);
   // shifts along X-axis and  Y-axis in object space. The display size
   // is not affected. sx and sy are to be understood as fractions of dX
   // and dY. Thus g.shift(0.5,0.5) will always make the old center the
   // new left-lower corner.
```

```
void shiftAbs(R sx, R sy);
   // shifts along X-axis and  Y-axis in object space. The display size
   // is not affected. sx and sy are to be understood as given relative
   // to units in object space.

void getObjectSpaceFrom(const Graph& gr );
   // takes the object space related data CL, CU, dX, dY from gr and
   // changes the corresponding data of *this as to become identical to
   // them.

// Setting the window according to the size of objects to be
// represented graphically. Overloading with differently
// typed arguments may be added later according to need

bool setXY(R_Vector const& x, R_Matrix const& y);
/*
   (x,y) represents a family of curves. x contains the x-values
   and y[i][j] are the y-values belonging to x=x[i].
   The values of CL, CU such that these curves
   fill the window a bit less than exactly
*/

bool setXY(const R_Vector& x, const R_Vector& y, Z method=0);
// method 0 as above, where y is treated as a 1-matrix
// method 1: the first point is taken as the origin and
// scale on x- and y-axis are the same. This was first
// needed 2005-03-31 for a representation of Brownian motion.

bool setXY(const R_Matrix& y);

bool setXY(V<C> const& pos, R secFactor=1.1, Z met=0);
   // For secFactor=1 the function adjusts the frame just as to frame
   // the set of points given pos. A larger secFactor allows for a
   // boundary arround the points, a smaller one yields a subframe.
   // The new method parameter met allows for met=1 to set for
   // both directions the same range. This is done in a minimal way
   // under this restriction. This does not change the functionality
   // controlled by secFactor.

bool setXY(V< V<R2> > const& pos, R secFactorX=1.01, R secFactorY=1.01);
   // as previous function but here a set of points is given
   // by a more complex object pos (typically a list of
   // discretized curves). A different security factor may be set
   // for the x-range and the y-range.

bool setXY(const V<R2>& pos, R secFactor=1.1);
   // same behavior as previous function

Z2 adjXY(Z divX=10, Z divY=10);
// Enlarges the window such that the boundaries correspond to values
```

```
// which in scientific number format give short and convenient
// numbers. Also a grid is drawn which subdivides the window
// conveniently into approximately divX subintervals in the x-axis
// and divY subintervals in the y-axis. The actual grid may deviate
// from these numbers since only convenient values for the grid line
// coordinates are accepted (for instance, the unit interval will
// be devided into 2 or 4 subintervals but never in 3).
// It returns nx and ny the division numbers of a grid.

void adjustScaleY(void);
// The origin CL and the X-interval [CL.x1,CU.x1] are left unchanged
// and CU.x2 is adjusted such that a circle in object space (actually,
// object plane) will be mapped into a circle (as opposed to an
// ellipse) on screen. The essential part of this operation is to
// set an appropriate scale in Y direction, which explains the name
// of the function.

void adjustScaleX(void);
// The origin CL and the Y-interval [CL.x2,CU.x2] are left unchanged
// and CU.x1 is adjusted such that a circle in object space (actually,
// object plane) will be mapped into a circle (as opposed to an
// ellipse) on screen. The essential part of this operation is to
// set an appropriate scale in X direction, which explains the name
// of the function.

Z  adjust(R2& x)const;
// If x is very near to one of the 4 infinite lines defining the window,
// x is shifted to sit exactly on this line. This is sometimes necessary
// for making decisions like in function boundary meaningful. The return
// value counts the number of adjustments done. The return value ranges
// from 0 (no adjustment) to 2. (XU>XL,YU>YL assumed.)

Z boundary(const R2& p)const;
// Consider the rectangle with corners (Xi,Yj), i=1,2, j=1,2.
// This function determines the position of the point P=(x,y) relative
// to this rectangle. Return value 0 says that P belongs to the
// interior. Return value 5 says exterior. If (x,y) belongs to that
// boundary the number of the boundary segment is returned. (The 4 parts
// are numbered counter-clockwise starting from (XL,YL), they are
// disjoint as sets, so that P can belong at most to one of them.

Z   same_boundary(R2 p, R2 q);
// gives 1 if the points p and q both belong to the boundary and their
// connecting line also belongs to the boundary. Zero is returned if any
// of the points doesn't belong to the boundary. And -1 is returned if
// the connection enters the interiour of the rectangle.

Z   next_corner(R2 b, R2& c);
// Consider the rectangle with corners (Xi,Yj), i=1,2, j=1,2.
// We look to the boundary as a oriented curve starting at (XL,YL)
```

```
// and running counter-clock-wise. Let b be a point of this curve
// (hence a point of the boundary of the rectangle). c is the first
// corner encountered when starting at b and following the orientation
// given above.
// The return value is the number of the corner in the same order.
// If b doesn't belong to the boundary, the return value is 0
// The next point of a corner has to be another corner.

Z   line_bound_inters(R a, R b, R c, Rn2& x);
// line_bound_inters stands for line-boundary-intersection.
// The line under consideration is given by the implicit equation
// ax+by+c=0.
// The return value is the number of intersection points of this line
// and the boundary of the rectangle considered in function boundary.
// (x[k][0],x[k][1]) is the kth intersection point. (Enumeration
// starting with 0.

Z   connect_on_boundary(R2 xi, R2 xf, Rn2& xc);
// Let k be the return value. Then the points xc[0],...xc[k],
// form a connection from xi to xf along the boundary of the rectangle
// considered in function boundary. The orientation is always
// counter-clock wise for having a simple logic, as opposed to having
// always the shortest way. xc[0]=xi, xc[k]=xf. The chain may have up
// to 6 points!

R  goto_boundary(R2 x, R2 e, R2& y);
// if x is any point and e any direction (not necessarily a unit vector).
// xa is the adjusted x (it is shifted to the boundary if it is very
// close to it, see function "adjust".
// Then y is the nearest point of the form xa+t*e, t>0, which belongs to
// the boundary if there is such a point. In this case t_min is returned.
// (Note that only t's larger than zero are considered, so we do not
// return zero if xa is on the boundary.)
// If no such point exists we return t=IMMENSE and put y=x. (This
// occurrs for instance if e is the zero vector). So the meaning is:
// goto boundary in e-direction if you can. If you can't, stay where you
// are and state that the distance is IMMENSE.

bool indFunc(const R2& p)const;
// returns true if p belongs to the object space rectangle associated
// with *this and false else.

void mark(R2 const& p, Color const& color)const;
   // Sets a marked pixel of a color given by the second
   // argument at screen point which corresponds to point p in object
   // space.

void mark(C const& p, Color const& color)const
{ mark(R2(p),color);}
```

```
void mark(const R2& p, Z size, const Color& color)const;
   // Sets a mark 'of size s' of a color given by the second
   // argument at screen point which corresponds to point p in object
   // space.
   // 'Size', presently, is the number of pixels which get colored.
   // The shape of the mark follows a simple and effective rule but
   // is not optimum with respect to symmetry to the the marked
   // position. All sizes larger than 17 are marked identically.

void mark(const C& p, Z size, const Color& color)const
{ mark(R2(p),size,color);}

void draw(R2 const& p1, R2 const& p2, Color const& color);
// This draws the restriction to the graphics window of the infinite
// straight line segment determined by the two points p1, p2.

void draw(C const& p1, C const& p2, Color const& color)
{ draw(R2(p1), R2(p2), color);}

void fillCircle(R2 const& center, R radius, const Color& color);
   // Draws a circle to the graphics window filled by color.
   // Clearly, the value of radius refers to *this' object
   // space.

void fillTriangle(R2 const& p1, R2 const& p2, R2 const& p3,
   Color const& color);
   // analogous to as fillCircle

Z  polygonClip( const Rn2& x, Rn2& y);
// x is an array of points (in a plane). The polygon determined
// by them is restricted ("clipped") to the window in such
// a way that it remains a polygon and any point where the original
// polygon leaves the window is connected on the boundary with the point
// of reentrance. The return value is the number of points of the clipped
// polygon. After function call, the points of the clipped polygon are
// held in the array y. The value of y before function call has no
// influence. The return value is the dimension of y (after function
// call, of course).

Z  bundleClip(const R_Vector& a, const R_Vector& b,
   const R_Vector& c, Rn2& y);
/*
Consider a set of n straight lines. The ith line is described by the
equation
   a[i]*x+b[i]*y+c[i]=0.
These lines are clipped to the window and the end points of the line
segments are connected on boundary such that a connected polygon arises.
The return value is the number of points of this polygon.
The points of the polygon are held in the array y[]. It is tried to
arrange the connection on boundary such that for a continuous bundle
```

```
(lines not too much changing from i to i+1) a minimum number of points
is created.
*/

void  show( const R_Vector& a,const R_Vector& b, const R_Vector& c,
      const Color& color);
/*
Here, n,a,b,c have the same meaning as in function bundle_clip.
The restriction of the lines to the window is drawn to screen without
creating a connected polygon (which would be suited for printing).
*/

void  showConnected( const R_Vector& a,
   const R_Vector& b, const R_Vector& c, const Color& color);
/* Let n,a,b,c be as in bundle_clip. The polygon created by this function
is represented drawn on screen. Having show_bundle_simple, this is
usefull only as a test for the function bundle_clip, the latter being
important for printing results.
*/

void  gridL(Z n_x=4, Z n_y=4);
// Draws a grid by dividing x-range in n_x equally sized parts, and
// y-range into n_y such parts. No text indicating the x,y ranges.

void clearAndGrid(Z nX, Z nY);
// Fills the screen window with Color colorActive
// draws a grid (only if newGrid!=0 what is always the
// case if not changed intentionally).
// Enlarges the object space rectangle such that the
// boundaries correspond to valueswhich in scientific
// number format give short and convenient numbers.
// A grid is drawn which subdivides the window conveniently
// into approximately divX subintervals in the x-axis
// and divY subintervals in the y-axis. The actual grid may
// deviate from these numbers since only convenient values
// for the grid line coordinates are accepted (for instance,
// the unit interval will be divided into 2 or 4 subintervals
// but never in 3). Stores the grid legend in the variable text_.


void draw(R_Vector const& x, R_Vector const& y, const Color& c);
// Draws the polygon (x[1],y[1]),...(x[n],y[n]), properly clipped,
// to the presently valid window (in color c)

bool show(const R_Vector& x, const R_Matrix& y,
           const CpmArrays::V<Color>& cl=rainBow,
           const Word& title=Word(), R xRelTitle=0.5,
           bool xLog=false, bool yLog=false);
/*
Graphical representation of a family of curves:
```

```
 The m lists of n object space points

      (x[1],y[1][1]),......,(x[n],y[1][n])
       ..................................
      (x[1],y[m][1]),....,(x[n],y[m][n])

define m polygons with n points  each.

Depending on autoScale==0 or autoScale!=0
the object-space corners CL, CU of *this will be adjusted or not adjusted
according to the minimum window in object space containing all these
points.

Then the polygons will be drawn to the screen-window determined
by *this according to the then valid object space window (thus the
polygons may leave the window, or even not being there at all, if
autoScale is zero, and will not leave the window otherwise).

The m curves are drawn in colors defined in the array cl
If there are more curves than entries in colorList the sequence of colors
starts gets repeated as often as needed. The colors colorActive and
gridColor, even if present in cl will not be used for curve drawing.

Whether the grid text is updated depends on the value of newGrid (!=0 for
proper updating - which is the normal case). The updated gridtext gets
written after the curves, so that the grid text may not be covered by
the curves, it nevertheless may be difficult to read.

A typical code segment for flexible display of many curve families on
screen is the following

   Frame f;
   N ni=2;
   N nj=3;
   Frames fs(f,ni,nj);
   V<Graph> g(ni*nj);
      // a list of graphs corresponding to a ni*nj matrix of
      // subframes of f
      // one also could use directly a ni*nj matrix of Graphs:
      // VV<Graph> g(ni,nj)
   N k=0;
   R2 textPos(0.1,0.1);
      // standard text position might conflict with the curves
      // which then might become hidden
   for (i=1;i<=ni;i++) for (j=1;j<=nj;j++) {
      k++;
      g[k]=fs[i][j];
      (g[k]).setAutoScale(0);
      (g[k]).setXY(4.3,0,1.6,2.4);
```

```
         // given a fixed coordinatesystem to make the curves in
         // different
         // Graphs comparable
      (g[k]).setNewGrid(1);
      (g[k]).setGridTextPosition(textPos);
      (g[k]).adjustXY(10,10);
         // making an appropriate grid
       g[k].show(.....)
   }

The last argument xRelTitle gives the relative x-coordinate of the
beginning of the title text.
*/

void show(const R_Matrix& a, const CpmArrays::V<Color>& cl=rainBow,
   const Word& title=Word(), R xRelTitle=0.5);
/*
Similar to the last function; the first line of matrix a is
interpreted as x values. The further lines are interpreted
as corresponding y-values of a family of curves
*/

bool showGL(const R_Vector& x, const R_Matrix& y, V<Color> const& cl,
            bool xLog=false, bool yLog=false);

bool showGL(R_Vector const& x, R_Vector const& y,
   Color const& c, Z bars);

bool  show(const R_Vector& x, const R_Vector& y,
   const Color& color=BLACK, Z bars=0,
   const Word& title=Word(), R xRelTitle=0.5);
// Similar to the last function. Only one polygon is shown and the
// color of the latter is given by 'color'.
// If bars is different from 0, a bar diagram is made instead of a curve
// Bar representation sometimes is usefull to strictly indicate the
// distinct values since we have no 'points and lines style'

bool makeLaTexGraph(R_Matrix const& a, Fig const& fg,
   ostream& str, Iv rx=Iv(0,140), Iv ry=Iv(0,100));
// Writes on a text stream the LaTex commands that let
// LaTex create a figure similiar to the representation of a.
// Additional input that makes sense for a printed figure and
// not for a screen graphic is provided by the argument fg.

void  show(const R_Vector& y, const Color& color=BLACK, Z bars=0,
      const Word& title=Word(), R xRelTitle=0.5);
// The vector of x-values is {1,2,... dim(y)} and the y-values are given
// by y. In all other respects like the previous function.

bool  show(F<R,R> const& f, Z n=100, Color const& cf=RED,
```

```
   V<R> const& xList=V<R>(0), Color const& cx=LIGHTBLUE);
// The n points (x,f(x)), with x ranging equidistantly from XL to XU,
// are connected to form a polygon, and the this polygon is shown in a
// autoscaled window and grid. Window data are written to screen.
// The polygon is drawn in a color described by cf=1,...15.
// For negative cf, see auto_show_list xList is a list of x-Values that
// will be marked on the x-Axis by vertical lines as far as they are in
// the x-Range determined by *this. The color of these lines is set as cx.
// Returns false if no graph could be created. This function is used in:
// R_Func f; Graph gr; gr.show(f).

template <class FRR>
void show(V< FRR > const& f, Z n=100,
   CpmArrays::V<Color> const& cl=rainBow, bool logarithmic=false)
// Generalizing previous function for an array of curves. For more
// flexibility, F<R,R> is replaced by a template parameter.
{
   using CpmGeo::Iv; // interval
   if (n<=1) n=2;
   Z d=f.dim();
   Iv iv(CL[1],CU[1]);
   R_Vector x(iv,n);
   R_Matrix y(d,n);
   for (Z i=1;i<=d;i++){
      for (Z j=1;j<=n;j++){
         y[i][j]=f[i](x[j]);
      }
   }
   show(x,y,cl,"",0.5,false,logarithmic);

}

void show(const R2_Func& fR, const R2_Func& fG, const R2_Func& fB );
// Three functions, defined on the object space of the Graph are
// represented as a distribution of R,G,B-signals. For this purpose,
// function values are converted to Au by the automatics chosen by the
// compiler.

void show(const R2_Func& fR, const R2_Func& fG, const R2_Func& fB,
   R XL, R XU, R YL, R YU, bool noDisplay=false);
// Three functions, defined on the object space of the Graph are
// represented as a distribution of R,G,B-signals. For this purpose,
// function values are converted to Au by the automatics chosen by the
// compiler. Only the part of the f's living on the rectangle XL<=X<=XU,
// YL<=Y<=YU are actually 'painted'. This allows to update small subareas
// of an image much more quickly than redrawing of the whole image would
// allow. Unfortunately the C++ syntax doesn't allow to put the natural
// default values XL=CL.x,... YU=CU.y.

void show(const R2_Func& fR, const R2_Func& fG, const R2_Func& fB,
```

```
   const Response& rR, const Response& rG, const Response& rB,
   Iv roiX, Iv roiY, bool noDisplay=false);
// Three functions, defined on the object space of the Graph are
// represented as a distribution of R,G,B-signals. For this purpose,
// function values are converted to Au by the action of the three
// instances of class Response.
// Only the part of the f's living on the rectangle XL<=X<=XU,
// YL<=Y<=YU are actually 'painted'. This allows to update small subareas
// of an image much more quickly than redrawing of the whole image would
// allow. Unfortunately the C++ syntax doesn't allow to put the natural
// default values XL=CL.x,... YU=CU.y.

void show(R2_Func const& f, Response const& rsp, Iv roiX=Iv(),
   Iv roiY=Iv());
// Functions f, defined on the object space of the Graph is
// represented as a distribution of R,G,B-signals. For this purpose,
// function values is converted to Color by the action of the instance
// rsp of class Response. Here the capability of class Response
// to create directly Colors by the new function member color()
// is exploited.
// Only the part of the f's living on the rectangle roiX x roiY are
// actually 'painted'. This allows to update small subareas
// of an image much more quickly than redrawing of the whole image would
// allow. If roiX/Y is void it gets replaced by getIvX()/Y().
// Most convenient form to represent functions on the plane.

void show(const F<RR, V<R> >& f, R XL, R XU, R YL, R YU,
   bool noDisplay=false);
// A vector-valued function, defined on the object space of the Graph are
// represented as a distribution of R,G,B-signals.
// For this purpose, function values are converted to Au by the automatics
// chosen by the compiler. Only the part of the f's living on the
// rectangle XL<=X<=XU, YL<=Y<=YU are actually 'painted'. This allows to
// update small subareas of an image much more quickly than redrawing of
// the whole image would allow. Unfortunately the C++ syntax doesn't allow
// to put the natural default values XL=CL.x,... YU=CU.y.

void show(const F<R2, R3 >& f, R XL, R XU, R YL, R YU,
   bool noDisplay=false);
// A vector-valued function, defined on the object space of the Graph are
// represented as a distribution of R,G,B-signals.
// For this purpose, function values are converted to Au by the automatics
// chosen by the compiler. Only the part of the f's living on the
// rectangle XL<=X<=XU, YL<=Y<=YU are actually 'painted'. This allows to
// update small subareas of an image much more quickly than redrawing of
// the whole image would allow. Unfortunately the C++ syntax doesn't allow
// to put the natural default values XL=CL.x,... YU=CU.y.

void show(const F<RR, V<R> >& f);
// A vector-valued function, defined on the object space of the Graph are
```

```
// represented as a distribution of R,G,B-signals. For this purpose,
// function values are converted to Au by the automatics chosen by the
// compiler.

void show(const F<C,C>& f, R gamma=0.01, bool zeroIsZero=false);
// exaluates f all over the object space associated with *this
// (considering this rectangle as a part of the complex plane and
// displays the function value as a color according to
// CpmImaging::Color(C). Notice that all negative values of gamma have
// the same effect (logarithmic response).
// If the last argument is true, zero brightness belongs to f-value
// C(0) and not to the minimum calue of |f(z)| over the image.
// There are situations where the range of |f(z)| is very small
// and essentially due to numerical noise. In these cases we
// make this noise visible if zeroIsZero=false. In these cases
// zeroIsZero=true gives a more natural representation.

void  show(Z n, R pl, R pu, R f1(R z1), R f2(R z2), const Color& c);
// The n points (f1(p),f2(p)), with p ranging equidistantly from pl to
// pu, are connected to form a polygon, and the this polygon is shown
// in a autoscaled window and grid. Window data are written to screen.
// The polygon is drawn in a color c

void addFrame(void);
// marks the boundary of the Graph by a line in GraphColor

void addFrm_(Color const& cf);
//: add frame
// marks the boundary of the Graph by a line in Color cf

void  mark( V<R> const& xList, V<Color> const& cList=V<Color>(1,RED));
// xList is a list of x-Values that will be marked on the x-Axis
// by vertical lines as far as they are in the x-Range determined by
// *this. The color of these lines is determined by cList. If
// cList is shorter than xList, it gets repeated periodically.
// The argument *thick allows to use thick lines which sometimes improves
// printability.

// most advanced function for representing a family of curves

bool mark(V< V<R2> > const& dat, V<Color> const& cl=rainBow, R secFac=-1);
// Graphical representation of a family of curves which don't need
// to have the same list of x-values. The representation of a curve as
// an instance of vector<R2> allows fast iteration and flexible
// generation by adding points. Now this type can very efficiently
// be converted to V<R2> so that there is no need for introducing
// std::vector in the interface of the C+- classes.
// This function (as all functions named 'mark') only writes to image
// memory.  To make the curves visible one has to call display().
// Adding text is best done by calling addText(...) with suitable
```

```
// arguments. See mark(Vl< Vl<C> > const&, ....) for the meaning of the
// argument secFac. The argument *thick allows to use thick lines
// which sometimes improves printability.

// most advanced function for representing a rectangular field
// of complex values.

void mark(Vl< Vl<C> > const& mat,Iv range=Iv(), Z ipolMethod=0,
   bool faithful=true, bool periodic=true, R secFac=-1,
   bool matrixstyle=false)const;
// Most efficient way of Graph to graphically represent a rectangular
// field of data.
// Interpolating representation of a matrix 'mat' of complex numbers.
// The indexing of the matrix is as it arrose in 2-dimensional
// quantum mechanical applications: mat[i][j] is such that the first
// index i grows from 0 to mat.dim() along the graphical x-axis
// (i.e. horizontally from left to right). And the second indey j
// grows from 0 to mat[0].dim() along the y-axis on screen
// (i.e. vertically from bottom to top, sic!). This correspondence
// between matrix indexes and graphical position holds for the
// default value matrixstyle=false. For matrixstyle=true, mat will be
// represented as a mat.dim() times mat[0].dim() matrix with the
// first matrix index growing vertically downwards and the second
// matrix index growing horizontally to the right. Especially
// mat[0][0] is shown at the upper left screen corner.
// The parameter 'range' has the effect that the complex number z gets
// represented by the color Color(z,range), see cpmimagingtools.h class
// Color. If range is the void interval (default), the actual values
// of mat will be used to define a suitable range for representation.
// Notice that Color::gamma has an influence on Color(z,range) and
// thus on the functionality of the present function.
// Parameter 'ipolMethod' has to be either 0 or 1. 0 for pixelation,
// 1 for bilinear interpolation.
// Interpolation at the boundaries is controlled by parameter
// 'periodic'. If it has value true, we use periodic continuation of the
// data in mat which is the correct method for my quantum mechanics
// application. If it has value false the boundary interpolation data
// are taken from a  continuation 'as constant'.
// This is OK for arbitraty content of mat.
// The interpolation methods here are appropriate only if mat has
// not (much) more pixels than *this. This is the common situation
// in modelling, where mat is hard to compute and therefore has typically
// not more than 100 x 100 pixels and should be displayed on screen
// on a larger window for convenient visual presentation.
// If the argument 'faithful' has the value 'true', the actually painted
// area has the aspect ratio as prescribed by mat, and fills the
// frame of *this maximally while respecting this boundary condition.
// For value false, the whole frame of *this will be used for the
// representation of the data; this means that the aspect ratio
// gets set from *this.
```

```
// Addition 2007-09-26: 'first run auto adjustment' controlled
// by the newly introduced last argument 'secFac'.
// For secFac>0 the range will set in the first call to
//       secFac*Max{ mat[i][j].absVal() | i,j}
// and then used in further calls.

//void mark(Vl< Vl<R3> > const& mat, Z ipolMethod=0,
//   bool faithful=true, bool periodic=true)const;
};

} // namespace

#endif
```

86 cpmgraph.cpp

```
//? cpmgraph.cpp
//? Status of work 2008-10-25.
//?
#include <cpmgraph.h>
#include <cpmimagingtools.h>
#include <cpmsystemdependencies.h>

#include <vector>
#include <iomanip>

using CpmGraphics::Viewport;

using CpmGraphics::Frame;
using CpmGraphics::Graph;
using CpmGraphics::Fig;
using CpmLinAlg::R2;
using CpmRootX::Z1;
using CpmLinAlg::Z2;
using CpmLinAlg::Z3;
using CpmGraphics::Rn2;
using std::vector;

using namespace CpmRoot;
using namespace CpmSystem;
using namespace CpmArrays;
using namespace CpmFunctions;
using namespace CpmImaging;


namespace{
// used as initialization in iniGraph(...) , data may be changed by
// member functions.
   const R GRID_XPOS=0.05;
   const R GRID_YPOS=0.92;
   const R GRID_HEIGHT=15;
// some badly treated method parameters
   const Z N_EXCESS=100;
   // 2001-3-21 index out of range for N_EXCESS=40 obtained
   // now no longer needed in function polygonClip
   const R IMMENSE=1e99;
}

//////////////////////////// class Graph ////////////////////////////
// basic new geometry functions

namespace{
   const R tinyRim=1e-6;
```

```
}

Z   Graph::to_i(R x, Z& excess)const
{
   excess=0;
   R xRel=(x-CL.x1)*dXInv;
   if (xRel<=0){
      if (xRel<-tinyRim) excess=-1;
      return 0;
   }
   else if (xRel>=1){
      if (xRel>1+tinyRim) excess=1;
      return nCol-1;
   }
   else return (Z)(xRel*nCol);
}

Z  Graph::to_j(R y, Z& excess)const
{
   excess=0;
   R yRel_=(CU.x2-y)*dYInv; // reversal of direction
      // as compared with to_x
   if (yRel_<=0){
      if (yRel_<-tinyRim) excess=+1; // sic, y too large!
      return 0;
   }
   else if (yRel_>=1){
      if (yRel_>1+tinyRim) excess=-1;
      return nRow-1;
   }
   else return (Z)(yRel_*nRow);
}

////////// end geometry

void Graph::iniGraph(R Xl, R Yl, R dX, R dY)
{
   setXY( Xl, Yl, dX, dY);
   gridColor=GRIDCOLOR;
   gridTextColor=WRITECOLOR;
   graphColor=RED;
   withOrigin=1;
   newGrid=1;
   autoScale=B(true);
   gridText=V<Word>(1);
   gridTextPosition.x1=GRID_XPOS;
   gridTextPosition.x2=GRID_YPOS;
   gridTextHeight_=GRID_HEIGHT;
}
```

```
Graph::Graph(R xl, R yl, R dx, R dy, const Color& color, R Xl, R Yl,
   R dX, R dY):Frame(xl,yl,dx,dy,color)
{
   iniGraph(Xl, Yl, dX, dY);
}

Graph::Graph(void):Frame()
{
   iniGraph(0,0,1,1);
}

Graph::Graph(const Color& c):Frame(c)
{
   iniGraph(0,0,1,1);
}

Graph::Graph(const Frame& f):
Frame(f)
{
   iniGraph(0,0,1,1);
}

Graph::Graph(Frame const& fr, R rR):
Frame(rR,rR*fr.getAspRat(),1-2*rR,1-2*rR*fr.getAspRat())
{
   iniGraph(0,0,1,1);
}
bool Graph::prnOn( ostream& str) const
// return preliminary
//notice that content is not copied and retrieved !!!!
{
   using CpmRoot::write;
   Z mL=2;
   Word loc("Graph::prnOn(ostream)");
   CPM_MA
   cpmwt("Graph");
   Frame::prnOn(str);
   cpmwt("CL");
   CL.prnOn(str);
   cpmwt("CU");
   CU.prnOn(str);
   CPM_MZ
   return true;
}

bool Graph::scanFrom (istream& in)
// return preliminary
{
   using CpmRoot::read;
   Z mL=2;
```

```
   Word loc("Graph::scanFrom (istream)");
   CPM_MA
   Frame::scanFrom(in);
   CL.scanFrom(in);
   CU.scanFrom(in);
   updatedXdY();
   CPM_MZ
   return true;
}

void Graph::setGridColor(const Color& c)
{
   gridColor=c;
}

void Graph::setGridTextColor(const Color& c)
{
   gridTextColor=c;
}

void Graph::setGraphColor(const Color& c)
{
   graphColor=c;
}

void Graph::setWithOrigin(Z c)
{
   withOrigin=c;
}

void Graph::setNewGrid(Z c)
{
   newGrid=c;
}

void Graph::setGridText(Word a)
{
   gridText[1]=a;
}

void Graph::setGridTextPosition(const R2& pos)
{
   gridTextPosition=pos;
}

void Graph::setGridTextPosition(R xText, R yText)
// pos is understood in the Frame coordinate system (xF,yF)
{
   gridTextPosition=R2(xText,yText);
}
```

```
Z  Graph::iX(R X)const
{
   Z exc;
   return to_i(X,exc)+a0();
}

R  Graph::iX_(R x)const
{
   return to_i_(x)+a0();
}

Z  Graph::jY(R Y)const
// Y<=CL.x2 corresponds to the lower boundery of the Frame
// Y>=CU.x2 corresponds to the upper boundery of the Frame
{
   Z exc;
   return to_j(Y,exc)+b0();
}

R  Graph::jY_(R y)const
{
   return to_j_(y)+b0();
}

Z   Graph::iX0(R x)const
// x=XL corresponds to left boundery of the Frame
// x=XU corr. to right bound. of Frame
// for x<XL or x>XU we return -1
{
   if (x<CL.x1 || x> CU.x1) return -1;
   else return iX(x);
}

Z   Graph::jY0(R y)const
// y=YL corresponds to the lower boundery of the Frame
// y=YU corresponds to the upper boundery of the Frame
// for y<YL or y>YU we return -1
{
   if (y<CL.x2 || y> CU.x2) return -1;
   else return jY(y);
}

Z2 Graph::toPixel(const R2& p)const
// maps a point from object space into a pixel (discrete position on
// screen
{
   Z2 res(iX(p.x1),jY(p.x2));
   return res;
}
```

```
R2 Graph::toObjectSpace(const Z2& z )const
{
   Word loc("Graph::toObjectSpace(Z2)");
   cpmassert(pU.x1!=pL.x1,loc);
   cpmassert(pU.x2!=pL.x2,loc);
   R dx=(CU.x1-CL.x1)/(pU.x1-pL.x1);
   R dy=(CU.x2-CL.x2)/(pL.x2-pU.x2);   // positive with pL-pU
   R2 p;
   p.x1=CL.x1+dx*(z.x1+0.5-pL.x1);
   p.x2=CU.x2-dy*(z.x2+0.5-pU.x2);
    return p;
}

R2 Graph::get_xFyF(const R2& xGyG)const
{
   R xG=xGyG.x1, yG=xGyG.x2;
   R xF=(xG-CL.x1)*dXInv, yF=(yG-CL.x2)*dYInv;
   return R2(xF,yF);
}

R2 Graph::get_xGyG(const R2& xFyF)const
{
   R xF=xFyF.x1, yF=xFyF.x2;
   R xG=CL.x1+xF*dX, yG=CL.x2+yF*dY;
   return R2(xG,yG);
}

bool Graph::updatedXdY(void)
{
   dX=CU.x1-CL.x1;
   dY=CU.x2-CL.x2;
   dXInv=cpminv(dX);
   dYInv=cpminv(dY);
   if (dX<=0){
      cpmwarning("Graph::updatedXdY(): dX<=0: dX="&cpmwrite(dX));
      return false;
   }
   else if (dY<=0){
      cpmwarning("Graph::updatedXdY(): dY<=0: dY="&cpmwrite(dY));
      return false;
   }
   else{
      return true;
   }
}

bool Graph::setXY(R X, R Y, R dX1, R dY1)
{
   R X_=X+dX1, Y_=Y+dY1;
```

```
   CpmRootX::order(X,X_);
   CpmRootX::order(Y,Y_);
   CL.x1=X; CU.x1=X_;
   CL.x2=Y; CU.x2=Y_;
   return updatedXdY();
}

bool Graph::setX(Iv iv)
{
   CL.x1=iv.inf(); CU.x1=iv.sup();
   return updatedXdY();
}

bool Graph::setY(Iv iv)
{
   CL.x2=iv.inf(); CU.x2=iv.sup();
   autoScale=B(false);
   return updatedXdY();
}

bool Graph::setXY(Iv ivx, Iv ivy)
{
   CL.x1=ivx.inf(); CU.x1=ivx.sup();
   CL.x2=ivy.inf(); CU.x2=ivy.sup();
   autoScale=B(false);
   return updatedXdY();
}

// Setting the window according to the size of objects to be
// represented graphically. Overloading with differently
// typed arguments may be added later according to need

namespace{

void splitVal(R& a, R& b, R tol) // changed 2007-10-03
{
// precondition: a<=b and tol>0;  postcondition: a<b
   if (a>b) cpmerror("splitVal(R&,R&,R), a > b not allowed");
   if (a==b){
      R aa=cpmabs(a);
      R eps= aa==0 ? tol : aa*tol;
      a-=eps;
      b+=eps;
   }
}

} // namespace

bool Graph::setXY(const R_Vector& x, const R_Matrix& y)
{
```

```
   const Z mL=4;
   Word loc("Graph::setXY(R_Vector,R_Matrix)");
   CPM_MA

   const R fracTol=1e-6; // the object space should be a bit larger since
      // a horicontal line (x[i],y[k][i]), i=1,2,... can not define
      // exacly the boundary of the object space. Otherwise numerical
      // noise would influence whether this line gets drawn in
      // showGL(R_Vector,R_Matrix,...).
   Z n=x.dim(),  m=y.dim1(), n1=y.dim2(), i;
   if (n!=n1) cpmerror("Graph::setXY(R_Vector,R_Matrix): n!=n1");
   cpmassert(n>0,loc);

   Iv ivx=x.infSup();
   R x_min=ivx.inf(), x_max=ivx.sup();
   splitVal(x_min,x_max,fracTol);
   R d_x=x_max-x_min;

   R_Vector yMin(m);
   R_Vector yMax(m);
   for (i=1;i<=m;i++){
      Iv ivi=y[i].infSup();
      yMin[i]=ivi.inf();
      yMax[i]=ivi.sup();
   }
   R y_min=yMin.inf();
   R y_max=yMax.sup();
   splitVal(y_min,y_max,fracTol);
   // ***
   if (withOrigin==1){ // This allows make the point y=0 appear in
                  // the graph
      if (y_min>0.){
          y_min=0. ;
      }
      else if (y_max<0.){
         y_max=0. ;
      }
   }
   R d_y=y_max-y_min; // moved from *** to here, 2007-10-03
   R eps_x=fracTol*d_x;
   CL.x1=x_min-eps_x;
   CU.x1=x_max+eps_x;
   R eps_y=fracTol*d_y;
   CL.x2=y_min-eps_y;
   CU.x2=y_max+eps_y;

   bool res=updatedXdY();
   CPM_MZ
   return res;
}
```

```
bool Graph::setXY(const R_Vector& x, const R_Vector& y, Z method)
{
   const R secFac=1.01;
      // a 'security factor' which lets the x,y data being displayed
      // without endig up on the rim of the Graph
   if (method==0){
      R_Matrix yM(1);
      yM[1]=y;
      return setXY(x,yM);
   }
   else{ // so far, maybe other methods may have to be added
      Z n=x.dim(),i;
      cpmassert(n==y.dim(),"Graph::setXY(R_Vector,R_Vector,Z)");
      R x1=x[1];
      R y1=y[1];

      R_Vector xc=x;
      R_Vector yc=y;
      for (i=1;i<=n;++i){
         xc[i]-=x1;
         yc[i]-=y1;
         xc[i]=cpmabs(xc[i]);
         yc[i]=cpmabs(yc[i]);
      }
      R xMax=xc.sup();
      R yMax=yc.sup();
      xMax*=secFac;
      yMax*=secFac;
      R asp=getAspRat();
      R xComp=xMax/asp;
      R yComp=yMax;

      Iv ivx,ivy;
      if (yComp>=xComp){
         ivy=Iv(y1-yMax,y1+yMax);
         ivx=Iv(x1-yMax*asp,x1+yMax*asp);
      }
      else{
         ivx=Iv(x1-xMax,x1+xMax);
         ivy=Iv(y1-xComp,y1+xComp);
      }
      return Graph::setXY(ivx,ivy);
   }
}

bool Graph::setXY(const R_Matrix& y)
{
   R_Vector x=y[1];
   Z m=y.dim1()-1;
```

```
   if (m==1){ // y has only two lines and represents one curve
      R_Vector yy;
      yy=y[2];
      return setXY(x,yy);
   }
   else{   // y has more than two lines and represents a family
      // of curves
      R_Matrix yy(m);
      for (Z i=1;i<=m;i++) yy[i]=y[i+1];
      return setXY(x,yy);
   }
}

bool Graph::setXY(const V<R2>& pos, R enlarge)
{
   Z n=pos.dim();
   Vo<R> posx(n);
   Vo<R> posy(n);
   for (Z i=1;i<=n;i++){
      posx[i]=pos[i].x1;
      posy[i]=pos[i].x2;
   }
   pair<Z,Z> limx=posx.indInfSup();
   pair<Z,Z> limy=posy.indInfSup();
   Iv ivx(posx[limx.first],posx[limx.second]);
   Iv ivy(posy[limy.first],posy[limy.second]);
   ivx.stretch(enlarge); // experiment
   ivy.stretch(enlarge);
   return Graph::setXY(ivx,ivy);
}

bool Graph::setXY(V< V<R2> > const& pos, R enlargeX, R enlargeY)
// implementation without ordering and with fast iteration
// till 2007-11-24 I had a (if, else if, else)-construct
// which failed to work in rare cases. Now everything is waterproof.
{
   R rHuge=R1::max;

   R xMin=rHuge;
   R yMin=rHuge;
   R xMax=-rHuge;
   R yMax=-rHuge;

   for (Z i=pos.b();i<=pos.e();i++){
     for (Z j=pos[i].b();j<=pos[i].e();++j){
         R2 xy=pos[i][j];
         R x=xy.x1;
         if (x<xMin) xMin=x;
         if (x>xMax) xMax=x;
         R y=xy.x2;
```

```
         if (y<yMin) yMin=y;
         if (y>yMax) yMax=y;
      }
   }
   Iv ivx(xMin,xMax);
   Iv ivy(yMin,yMax);
   ivx.stretch(enlargeX);
   ivy.stretch(enlargeY);
   return Graph::setXY(ivx,ivy);
}

bool Graph::setXY(const V<C>& pos, R enlarge, Z met)
{
   Z mL=3;
   Word loc("Graph::setXY(const V<C>& pos, R enlarge)");
   CPM_MA
   Z n=pos.dim();
   Vo<R> posx(n);
   Vo<R> posy(n);
   for (Z i=1;i<=n;i++){
      posx[i]=pos[i][1];
      posy[i]=pos[i][2];
   }
   pair<Z,Z> limx=posx.indInfSup();
   pair<Z,Z> limy=posy.indInfSup();
   Iv ivx(posx[limx.first],posx[limx.second]);
   Iv ivy(posy[limy.first],posy[limy.second]);
// avoiding zero sized window 2007-10-04
   const R tol=1e-6;
   R a=ivx[1], b=ivx[2];
   splitVal(a, b, tol);
   ivx=Iv(a,b);
   a=ivy[1]; b=ivy[2];
   splitVal(a, b, tol);
   ivy=Iv(a,b);
   bool res;
   if (met==1){
      Iv ivAll=ivx|ivy;
      ivAll.stretch(enlarge);
      res=Graph::setXY(ivAll,ivAll);
   }
   else{
      ivx.stretch(enlarge);
      ivy.stretch(enlarge);
      res=Graph::setXY(ivx,ivy);
   }
   CPM_MZ
   return res;
}
```

```
void Graph::adjustScaleX(void)
{
   R asp=screenAspRat(); // added 2005-08-23
   R scale=(CU.x2-CL.x2)/(cU.x2-cL.x2);
   R dX=scale*(cU.x1-cL.x1)*asp;
   CU.x1=CL.x1+dX;
   updatedXdY();
}

void Graph::adjustScaleY(void)
{
   R aspInv=screenAspRatInv();
   R scale=aspInv*(CU.x1-CL.x1)/(cU.x1-cL.x1);
   R dY=scale*(cU.x2-cL.x2);
   CU.x2=CL.x2+dY;
   updatedXdY();
}

bool Graph::scale(R magnification)
{
   R2 C0=0.5*(CU+CL);  // center
   R2 dw=CU-C0;
   dw*=magnification;
   CU=C0+dw;
   CL=C0-dw;
   return updatedXdY();
}

bool Graph::scaleY(R mag)
{
   Iv iy=getIvY();
   iy=iy.stretch(mag);
   return setY(iy);
}

void Graph::shift(R sx, R sy)
{
   R2 ds(sx*dX,sy*dY);
   CU+=ds;
   CL+=ds;
   updatedXdY();
}

void Graph::shiftAbs(R sx, R sy)
{
   R2 ds(sx,sy);
   CU+=ds;
   CL+=ds;
   updatedXdY();
}
```

```
void Graph::getObjectSpaceFrom(const Graph& gr)
{
   CL.x1=(gr.CL).x1;
   CL.x2=(gr.CL).x2;
   CU.x1=(gr.CU).x1;
   CU.x2=(gr.CU).x2;
   updatedXdY();
}

Z  Graph::adjust(R2& x)const
{
   R tinyx=1e-10*(cpmabs(CU.x1-CL.x1));  // 93-3-10 static R changed to R
   R tinyy=1e-10*(cpmabs(CU.x2-CL.x2));  // static leaves tinyx,tinyy the
   // same even if Xi,Yi where changed
   R x1=x.x1,x2=x.x2;
   Z  k=0;
   if (cpmabs(x1-CL.x1)<tinyx) { x.x1=CL.x1; k++;}
   if (cpmabs(x1-CU.x1)<tinyx) { x.x1=CU.x1; k++;}
   if (cpmabs(x2-CL.x2)<tinyy) { x.x2=CL.x2; k++;}
   if (cpmabs(x2-CU.x2)<tinyy) { x.x2=CU.x2; k++;}
   return k;
}

Z  Graph::boundary(const R2& p)const
{
   R2 u=p; adjust(u);    // otherwise not meaningfull
   R x_=u.x1,y_=u.x2;
   if (CL.x1<x_ && x_<CU.x1 && CL.x2<y_ && y_<CU.x2) return 0;
   if (y_==CL.x2 && CL.x1<=x_ && x_<CU.x1) return 1;
   if (x_==CU.x1 && CL.x2<=y_ && y_<CU.x2) return 2;
   if (y_==CU.x2 && CL.x1<x_ && x_<=CU.x1) return 3;
   if (x_==CL.x1 && CL.x2<y_ && y_<=CU.x2) return 4;
   return 5;
}

Z  Graph::same_boundary(R2 p, R2 q)
{
   adjust(p); adjust(q);      // no effect if the points are exactly
                        // on the boundary
   Z  j1=boundary(p);
   Z  j2=boundary(q);
   if (j1==0||j1==5||j2==0||j2==5) return 0;
   if (j1==j2) return 1; // fastest decision but not complete
   if (p.x1==CL.x1 && q.x1==CL.x1) return 1;
   if (p.x1==CU.x1 && q.x1==CU.x1) return 1;
   if (p.x2==CL.x2 && q.x2==CL.x2) return 1;
   if (p.x2==CU.x2 && q.x2==CU.x2) return 1; // this would be complete
   // also without the fast pre-decision
   return -1;
```

```
}

Z  Graph::next_corner(R2 b, R2& c)
{
   adjust(b); // no effect if b is precisely on boundary
   R x=b.x1,y=b.x2;
   if ((y==CL.x2) && CL.x1<=x && x<CU.x1)
      { c.x1=CU.x1; c.x2=CL.x2; return 2;}
   if ((x==CU.x1) && CL.x2<=y && y<CU.x2)
      { c.x1=CU.x1; c.x2=CU.x2; return 3;}
   if ((y==CU.x2) && CL.x1<x && x<=CU.x1)
      { c.x1=CL.x1; c.x2=CU.x2; return 4;}
   if ((x==CL.x1) && CL.x2<y && y<=CU.x2)
      { c.x1=CL.x1; c.x2=CL.x2; return 1;}
   c.x1=CU.x1; c.x2=CU.x2; return 0;
}

Z  Graph::line_bound_inters(R a, R b, R c, Rn2& y)
{
   Rn2 p(4), x(2);
   R2 q;
   if (a==0) {
      if (b==0) return 0;
      R y_=-c/b;
      if (CL.x2<=y_ && y_<=CU.x2){
         x[0].x1=CL.x1;
         x[0].x2=y_;
         x[1].x1=CU.x1;
         x[1].x2=y_;
         y=x;
         return 2;
      }
      else return 0;
   }
   if (b==0) {
      if (a==0) return 0;
      R x_=-c/a;
      if (CL.x1<=x_ && x_<=CU.x1) {
         x[0].x1=x_;
         x[0].x2=CL.x2;
         x[1].x1=x_;
         x[1].x2=CU.x2;
         y=x;
         return 2;
      }
      else return 0;
   }
   R b_=-1./b, a_=-1./a;
   p[0].x2=b_*(c+a*CL.x1); //y-value belonging to x=CL.x1
   p[0].x1=CL.x1;
```

```
   p[1].x2=b_*(c+a*CU.x1);  // y-value for CU.x1
   p[1].x1=CU.x1;
   p[2].x1=a_*(c+b*CL.x2);  //x-value for CL.x2
   p[2].x2=CL.x2;
   p[3].x1=a_*(c+b*CU.x2);  // x-value for CU.x2
   p[3].x2=CU.x2;

   Z  k=0;
   for (Z  i=0; i<4; i++) { q=p[i];
      adjust(q);
      Z  j=boundary(q);
      if (j>0 && j<5) {
         if (k<2){
            x[k]=q; k++;
         }
         else{
            return k;
         }
      }
   }
    y=x;
   return k;
}

Z  Graph::connect_on_boundary(R2 xi, R2 xf, Rn2& yc)
{
   R2 xa,p;
   Rn2 xc(6);
   adjust(xi); adjust(xf);
   Z  k1=boundary(xi);
   Z  k2=boundary(xf);
   if (k1==0 || k1==5)
      cpmwarning("point not on boundary in connect_on_b..");
   if (k2==0 || k2==5)
      cpmwarning("point not on boundary in connect_on_b..");
    xc[0]=xa=xi;
   Z  k=1;
iteration:
   Z  jj=same_boundary(xa,xf);
   if (jj==1) {
      xc[k]=xf;
      yc=xc;
      return k;
   }
   else { next_corner(xa,p);
         xc[k]=xa=p;
         k++;
         if (k>5) cpmerror("Graph","connect_on_boundary: k too large");
         goto iteration;
   }
```

```
}

R Graph::goto_boundary(R2 x, R2 e, R2& y)
{
   Rn d(4),d_pos(4);
   R2 xa;
   Z  i;
   xa=x;
   adjust(xa);
   if (e.x1==0. && e.x2==0.){
      y=x; return IMMENSE;
   }
   R r0=(e.x1==0.? IMMENSE : 1/e.x1);// one of those may be zero
   R r1=(e.x2==0.? IMMENSE : 1/e.x2);
   d[0]=(CL.x1-xa.x1)*r0;
   d[1]=(CU.x1-xa.x1)*r0;
   d[2]=(CL.x2-xa.x2)*r1;
   d[3]=(CU.x2-xa.x2)*r1;
   Z  k=0;
   // The points xa+d[i]*e belong to one of the defining infinite lines
   for (i=0;i<4;i++){
      if (d[i]>0){
         d_pos[k]=d[i] ;
         k++;
      }
   }
   //Now k is the number of positiv d's found.
   if (k==0) { y.x1=x.x1; y.x2=x.x2; return IMMENSE;}
   // this means that  in the direction of e no boundary point was found.
   // Maybe, in direction -e there may be one,
   // but this is of no interest.
   R d_=IMMENSE;
   for (i=0;i<k;i++){
      y=x+d_pos[i]*e;
      adjust(y);
      Z  z=boundary(y);
      if (z>0 && z<5 && (d_pos[i]<d_)){
         d_=d_pos[i];
      }
   }
   if (d_ == IMMENSE){
      y=x;
      return d_;
   }
   else {
      y=x+d_*e;
      adjust(y);
      return d_;
   }
}
```

```
bool Graph::indFunc(const R2& p)const
{
   if (p.x1>CU.x1) return false;
   if (p.x1<CL.x1) return false;
   if (p.x2>CU.x2) return false;
   if (p.x2<CL.x2) return false;
   return true;
}

void Graph::mark( const R2& p, const Color& color)const
{
   Z exc;
   Z i=to_i(p.x1,exc);
   if (exc) return;
   Z j=to_j(p.x2,exc);
   if (exc) return;
   img.putPel(i,j,color);
}

void Graph::mark(const R2& p, Z size, const Color& color)const
{
   Z exc;
   Z i=to_i(p.x1,exc);
   if (exc) return;
   Z j=to_j(p.x2,exc);
   if (exc) return;
   img.putPel(i,j,color);

   if (size==1) return;
   i++;
   img.putPel(i,j,color);

   if (size==2) return;
   j++;
   img.putPel(i,j,color);

   if (size==3) return;
   i--;
   img.putPel(i,j,color);

   if (size==4) return;
   i--;
   img.putPel(i,j,color);

   if (size==5) return;
   j--;
   img.putPel(i,j,color);

   if (size==6) return;
```

```
j--;
img.putPel(i,j,color);

if (size==7) return;
i++;
img.putPel(i,j,color);

if (size==8) return;
i++;
img.putPel(i,j,color);

if (size==9) return;
i++;
img.putPel(i,j,color);

if (size==10) return;
j++;
img.putPel(i,j,color);

if (size==11) return;
j++;
img.putPel(i,j,color);

if (size==12) return;
j++;
img.putPel(i,j,color);

if (size==13) return;
i--;
img.putPel(i,j,color);

if (size==14) return;
i--;
img.putPel(i,j,color);

if (size==15) return;
i--;
img.putPel(i,j,color);

if (size==16) return;
i--;
img.putPel(i,j,color);

 if (size==17) return;
j--;
img.putPel(i,j,color);

 if (size==18) return;
j--;
img.putPel(i,j,color);
```

```
    if (size==19) return;
   j--;
   img.putPel(i,j,color);

    if (size==20) return;
   j--;
   img.putPel(i,j,color);

    if (size==21) return;
   i++;
   img.putPel(i,j,color);

    if (size==22) return;
   i++;
   img.putPel(i,j,color);

    if (size==23) return;
   i++;
   img.putPel(i,j,color);

    if (size==24) return;
   i++;
   img.putPel(i,j,color);

    if (size==25) return;
   i++;
   img.putPel(i,j,color);

    if (size==26) return;
   j++;
   img.putPel(i,j,color);

    if (size==27) return;
   j++;
   img.putPel(i,j,color);

    if (size==28) return;
   j++;
   img.putPel(i,j,color);

    if (size==29) return;
   j++;
   img.putPel(i,j,color);

    return;
}

void Graph::draw( R2 const& p1,  R2 const& p2,
   Color const& c)
```

```
{
   R x1=iX_(p1.x1);
   R y1=jY_(p1.x2);
   R x2=iX_(p2.x1);
   R y2=jY_(p2.x2);
   line(x1,y1,x2,y2,c);
}

void Graph::fillCircle(const R2& center, R radius, const Color& c)
{
   R  x1=iX_(center.x1-radius);
   R  y1=jY_(center.x2-radius);
   R  x2=iX_(center.x1+radius);
   R  y2=jY_(center.x2+radius);
   fillEllipse(x1,y1,x2,y2,c);
}

void Graph::fillTriangle(R2 const& p1, R2 const& p2, R2 const& p3,
   Color const& color)
{
   R2 q1=natTrn(p1);
   R2 q2=natTrn(p2);
   R2 q3=natTrn(p3);
   fill(TriAng(q1,q2,q3),color);
}

Z Graph::bundleClip(const R_Vector& a,
   const R_Vector& b, const R_Vector& c, Rn2& yClip)
{
   Rn2 x(2), xc(6);
   R2 x_old(CL.x1,CL.x2),x_new;
   Z k=0, n=a.dim();
   Rn2 y(2*n+N_EXCESS);

   Z i,i1=0,i2=1,i1_,j,p;
   for (i=1;i<=n;i++){
      j=line_bound_inters(a[i],b[i],c[i],x);
      if (j==2){
         x_new=x[i1];
         j=connect_on_boundary(x_old,x_new,xc);
         for (p=0;p<=j;p++){  // note that xc gives p+1 points
            y[k]=xc[p];
            k++;
         }
         y[k]=x[i2];
         k++;
         x_old=x[i2];
         i1_=i1; i1=i2; i2=i1_; // exchanging i1 and i2
      }
   }
```

```
   y=y.resize(k);
   yClip=y;
   return k;
}

void Graph::show(const R_Vector& a, const R_Vector& b, const R_Vector& c,
   const Color& color)
{
   Rn2 x(2);
   Z i,j, n=a.dim();

   for (i=1;i<=n;i++){
      j=line_bound_inters(a[i],b[i],c[i],x);
      if (j==2){
         draw(x[0],x[1],color);
      }
   }
   display(writeOnShow);
}

void Graph::showConnected(const R_Vector& a,
   const R_Vector& b, const R_Vector& c, const Color& color)
{
   Z i,k;

   Rn2 y;
   k=bundleClip(a,b,c,y);
   for (i=0;i<k-1;i++){
      draw(y[i],y[i+1],color);
   }
   display(writeOnShow);
}

void Graph::gridL(Z n_x, Z n_y)
{
   if (cpmverbose>4) cpmmessage("gridL() started");
   if (n_x<1) n_x=1;
   if (n_y<1) n_y=1;

   R h_x=1./n_x;
   R h_y=1./n_y;
   Z i,k;

   for (i=1; i<n_x; i++){ // vertical lines, lines of constant x
      k=ix(i*h_x);
      line(k, pL.x2, k, pU.x2, gridColor);
   }

   for (i=1; i<n_y; i++){ // horizontal lines, lines of constant y
      k=jy(i*h_y);
```

```
      line(pL.x1, k, pU.x1, k, gridColor);
   }

// vertical border lines
   line(pL.x1, pL.x2, pL.x1, pU.x2, gridColor);
   line(pU.x1, pL.x2, pU.x1, pU.x2, gridColor);

// horizontal border lines
   line(pL.x1, pL.x2, pU.x1, pL.x2, gridColor);
   line(pL.x1, pU.x2, pU.x1, pU.x2, gridColor);
   // here it is evident that we have a clean grid with no
   // 1-pixel gaps possible if lines actually fill their starting
   // and end pixel
}

void Graph::draw(const R_Vector& x, const R_Vector& y, const Color& c)
{
   Z mL=3;
   Word loc="Graph::draw(R_Vector,R_Vector,Color)";
   CPM_MA
   Z n=cpminf<Z>(x.dim(),y.dim());
   if (n==0){
      cpmwarning(loc&": n==0, nothing to draw");
   }
   else if (n==1)
      mark(R2(x[1],y[1]),c);
   else{
      for (Z i=1;i<n;++i) draw(R2(x[i],y[i]),R2(x[i+1],y[i+1]),c);
   }
   CPM_MZ
}

Z2 Graph::adjXY(Z divX, Z divY)
{
   R2 a;
   Z nX=lad_4(CL.x1,CU.x1,divX,a);
   CL.x1=a.x1;
   CU.x1=a.x2;
   Z nY=lad_4(CL.x2,CU.x2,divY,a);
   CL.x2=a.x1;
   CU.x2=a.x2;
   updatedXdY();
    // nX and nY are assured not to be 0
   return Z2(nX,nY);
}

void Graph::clearAndGrid(Z nX, Z nY)
{
   const Z nAcc=4;
      // parameter to ostringstream::setprecision, default value
```

```
      // was not always sufficient in my applications (added 2007-09-12)
   if (!noGridText){
      ostringstream ost1,ost2;
      ost1<<setprecision(nAcc);
      ost2<<setprecision(nAcc);
      if (nX==0 || noGrid){
         ost1<<CL.x1<<" < x < "<<CU.x1;
      }
      else{
         R xStep=(CU.x1-CL.x1)/nX;
         ost1<<"x:"<<CL.x1<<"("<<xStep<<")"<<CU.x1;
      }
      if (nY==0 || noGrid){
         ost2<<CL.x2<<" < y < "<<CU.x2;
      }
      else{
         R yStep=(CU.x2-CL.x2)/nY;
         ost2<<"y:"<<CL.x2<<"("<<yStep<<")"<<CU.x2;
      }
      R2 p1=gridTextPosition;
      R hT=gridTextHeight_;
      R dh=hT*2/h(); // shift of two text lines
      R2 p2=p1; p2.x2-=dh;
      Word w1=Word(ost1.str());
      Word w2=Word(ost2.str());
      TextCom t1(p1,w1,gridTextColor,hT);
      TextCom t2(p2,w2,gridTextColor,hT);
      text_+=t1;
      text_+=t2;
   }
   if (newGrid && !noGrid){ // newGrid set to 1 in constructor
      // and normally never changed. Name misleading;
      // comes not only then in action if the previous
      // grid actually has to be changed
      paint(colorActive);
      gridL(nX,nY);
   }
}

bool Graph::show(const R_Vector& x, const R_Vector& y,
   const Color& c, Z bars, const Word& title, R xPosRel)
{
   Z mL=5;
   Word loc("Graph::show(R_Vector,R_Vector,Color,Z,Word,R)");
   CPM_MA
   if (title.dim()>0) addText(title, xPosRel);
   bool res=showGL(x,y,c,bars);
   CPM_MZ
   return res;
}
```

```
void  Graph::show(const R_Vector& y, const Color& c, Z bars,
    const Word& title, R xPosRel)
{
   Z d=y.dim();
   R_Vector x(d);
   for (Z i=1;i<=d;i++) x[i]=i;
   show(x,y,c,bars,title,xPosRel);
}

void Graph::addFrame(void)
{
   addFrm_(graphColor);
}

void Graph::addFrm_(Color const& cf)
{
   Z k=pL.x1;
   line(k,pL.x2,k,pU.x2,cf);
   k=pU.x1;
   line(k,pL.x2,k,pU.x2,cf);
   k=pL.x2;
   line(pL.x1,k,pU.x1,k,cf);
   k=pU.x2;
   line(pL.x1,k,pU.x1,k,cf);
}

bool Graph::show(const R_Vector& x, const R_Matrix& y,
   const CpmArrays::V<Color>& cl, const Word& title, R xPosRel,
   bool xLog, bool yLog)
{
   Z mL=5;
   Word loc("Graph::show(R_Vector,R_Matrix,V<Color>,Word,R,bool,bool)");
   CPM_MA
   if (title.dim()>0) addText(title, xPosRel);
      // notice that setText cancels all previously set text
      // thus also the grid text
   bool res=showGL(x,y,cl,xLog,yLog);
   CPM_MZ
   return res;
}

bool Graph::showGL(R_Vector const& xi, R_Matrix const& yi,
   CpmArrays::V<Color> const& cl, bool xLog, bool yLog)
{
   Z mL=3;
   Word loc("Graph::showGL(R_Vector,R_Matrix,V<Color>)");
   CPM_MA
   R_Vector x=xLog ? xi.log10() : xi;
   R_Matrix y=yLog ? yi.log10() : yi;
```

```
   if (x.dim()!=y.dim2())
      cpmerror (loc&": dimension mismatch");

   if (autoScale){
      bool b=setXY(x,y);
      if (b==false){
         cpmwarning(loc&": no graph could be created");
         return false;
      }
      Z2 steps=adjXY();
      clearAndGrid(steps[1],steps[2]);
   }
   else{
      Word xL=cpm(CL.x1);
      Word xU=cpm(CU.x1);
      Word yL=cpm(CL.x2,3);
      Word yU=cpm(CU.x2,3);
      clearAndGrid(10,10); // was 0,0
   }
   Vo<Color> clAct(cl);
   V<Color> exceptions(2);
   exceptions[1]=colorActive;
   exceptions[2]=gridColor;
   clAct=clAct.purge(exceptions);
      // clAct may be void after purge (2006-02-05)
   Z cAd=clAct.dim();
   if (cAd==0){
      cpmwarning(loc&
         ": color array was void after purging, replaced by RED"
      );
      clAct=V<Color>("",RED);
   }
   Z m=y.dim1();
   for (Z i=1;i<=m;i++){
       draw(x,y[i],clAct(i,CYCLIC));
   }
   display(writeOnShow);
   CPM_MZ
   return true;
}

bool Graph::showGL(R_Vector const& x, R_Vector const& y,
   Color const& c, Z bars)
{
   Z mL=5;
   Word loc("Graph::showGL(R_Vector,R_Vectot,Color,Z)");
   CPM_MA
   if (x.dim()!=y.dim())
      cpmerror (loc&": dimension mismatch");
```

```
   if (autoScale){
      bool b=setXY(x,y);
      if (b==false){
         cpmwarning(loc&": no graph could be created");
         return false;
      }
      Z2 steps=adjXY();
      clearAndGrid(steps[1],steps[2]);
   }
   else{
      Word xL=cpm(CL.x1);
      Word xU=cpm(CU.x1);
      Word yL=cpm(CL.x2,3);
      Word yU=cpm(CU.x2,3);
      clearAndGrid(0,0);
   }
   Color cFinal=( c==colorActive ? graphColor : c);
   if (bars==0){
       draw(x,y,cFinal);
   }
   else{
      Z n=x.dim();
      cpmassert(n==y.dim(),loc);
      bool thick=true;
      bool limited=true;
      for (Z i=1;i<=n;i++){
         R2 p1(x[i],0.);
         R2 p2(x[i],y[i]);
         draw(p1,p2,cFinal);
      }
   }
   display(writeOnShow);
   CPM_MZ
   return true;
}

void Graph::show(const R_Matrix& a, const CpmArrays::V<Color>& cl,
    const Word& title, R xPosRel)
{
   Z mL=5;
   Word loc("Graph::show(const R_Matrix& , ...)");
   CPM_MA
   Z i,ma=a.dim1(),na=a.dim2();
   if (ma<1){
      cpmwarning("could not show a matrix of 0 lines");
   }
   else if (na<1){
      cpmwarning("could not show a matrix of 0 columns");
   }
```

```
   else{
      R_Vector x=a[1];
      ma--;
      R_Matrix y(ma);
      for (i=1;i<=ma;++i) y[i]=a[i+1];
      show(x,y,cl,title,xPosRel);
   }
   CPM_MZ
}

namespace{

   void lin(R2 p1, R2 p2, ostream& str)
   {
      Z np=1;
      str<<endl<<"   "<<"\\drawline[0]("<<p1[1]<<","<<p1[2]<<
         ")("<<p2[1]<<","<<p2[2]<<")";
   }

   void text(R2 pos, Word const& txt, R ox, R oy, ostream& str)
   {
      R px=pos[1]+ox;
      R py=pos[2]+oy;
      str<<endl<<"   \\put("<<
         px<<","<<py<<"){\\makebox(0,0)[bl]{"<<txt<<"}}";
   }

   void sym(R2 p, Z s, ostream& str)
   {
      if (s==-1)
         str<<endl<<
         "   \\put("<<p[1]<<","<<p[2]<<
         "){\\makebox(0,0){\\tiny $\\blacksquare$}}";
      else if (s==-2)
         str<<endl<<
         "   \\put("<<p[1]<<","<<p[2]<<
         "){\\makebox(0,0){\\tiny $\\blacktriangle$}}";
      else
         str<<endl<<
         "   \\put("<<p[1]<<","<<p[2]<<
         "){\\makebox(0,0){\\tiny $\\bullet$}}";
   }
   // -1 and -2 experiment, 1 and 2 are original
}

bool Graph::makeLaTexGraph(R_Matrix const& a, Fig const& fg,
   ostream& str, Iv px, Iv py)
{
   Z mL=1;
   Word loc("Graph::makeLaTexGraph(R_Matrix ...)");
```

```
CPM_MA
R hLeg=7;
R width=px.absVal();
R height=py.absVal();
R raster=2.5*hLeg;

Z nX=cpmround(width/raster);
Z nY=cpmround((height-hLeg)/raster);
Word grid_text=gridText[1];
if (autoScale){
   cpmmessage(mL,"block autoScale started");
   setXY(a);
   R2 aa;
   Z divX=nX,divY=nY;
   nX=lad_4(CL.x1,CU.x1,divX,aa);
   CL.x1=aa.x1;
   CU.x1=aa.x2;
   nY=lad_4(CL.x2,CU.x2,divY,aa);
   CL.x2=aa.x1;
   CU.x2=aa.x2;
   updatedXdY();
   // nX and nY are assured not to be 0
   R xStep=(CU.x1-CL.x1)/nX;
   R yStep=(CU.x2-CL.x2)/nY;
   ostringstream ost;
   Z ad=a.dim();
   if (ad==2){
      ost<<"$x="<<fg.abs<<"\\;:\\;"<<CL.x1<<"("<<xStep<<")"<<CU.x1<<
      " \\quad \\quad  y="<<fg.ord[1]<<" (\\bullet)"
      <<"\\;:\\;"<<CL.x2<<"("<<yStep<<")"<<CU.x2<<"$";
      grid_text=Word(ost.str());
   }
   else if (ad==3){
      ost<<"$x="<<fg.abs<<"\\;:\\;"<<CL.x1<<"("<<xStep<<")"<<CU.x1<<
      " \\quad \\quad  y="<<fg.ord[1]<<
      " (\\blacksquare), \\:"<<fg.ord[2]<<" (\\blacktriangle) "
      <<"\\;:\\;"<<CL.x2<<"("<<yStep<<")"<<CU.x2<<"$";
      grid_text=Word(ost.str());
   }
   else ;

   cpmmessage(mL,"block autoScale done");
}
Iv rx=ranX();
Iv ry=ranY();
Iv pyU(py[1]+hLeg,py[2]); // U means used (for graphics)
R_Func fx(rx,px);
R_Func fy(ry,pyU);

R ox=px[1];
```

```
   R oy=py[1];
   R lx=px[2]-ox;
   R ly=py[2]-oy;

   V<R> vpx=px.div(nX+1);
   V<R> vpy=pyU.div(nY+1);
   Z dx=vpx.dim();
   Z dy=vpy.dim();
   Z i,j;

   str.setf(ios_base::fixed,ios_base::floatfield);
   str<<endl<<"% Output of function Graph::makeLaTexGraph(...)";
   str<<endl<<"\\setlength{\\unitlength}{1mm}";
   str<<endl<<"\\begin{figure}";
   str<<endl<<"\\begin{center}";
   str<<endl<<"\\begin{picture}("<<lx<<","<<ly<<")("<<ox<<","<<oy<<")";
   str.precision(6);
   str<<endl<<"\\linethickness{0.01mm}";
   // making the grid
   for (i=1;i<=dx;++i) lin(R2(vpx[i],vpy[1]),R2(vpx[i],vpy[dy]),str);
   for (j=1;j<=dy;++j) lin(R2(vpx[1],vpy[j]),R2(vpx[dx],vpy[j]),str);
   // writing the gridtext
   lin(R2(vpx[1],pyU[1]),R2(vpx[dx],pyU[1]),str);
   lin(R2(vpx[1],py[1]),R2(vpx[dx],py[1]),str);
   lin(R2(vpx[1],py[1]),R2(vpx[1],pyU[1]),str);
   lin(R2(vpx[dx],py[1]),R2(vpx[dx],pyU[1]),str);
   text(R2(2,1),grid_text,ox,oy,str);

   Z m=a.dim1();
   Z n=a.dim2();
   for (i=2;i<=m;++i){
      Z sy=i-1;
      for (j=1;j<=n;++j){
         R2 p1(fx(a[1][j]),fy(a[i][j]));
         sym(p1,sy,str);

      }
   }

   str<<endl<<"\\end{picture}";
   str<<endl<<"\\caption{"<<fg.tit<<"}";
   str<<endl<<"\\end{center}";
   str<<endl<<"\\end{figure}";
   CPM_MZ
   return true;
}

bool Graph::show(const F<R,R>& f, Z n, const Color& cf,
   const V<R>& xList, const Color& cx)
{
```

```
   using CpmGeo::Iv; // interval
   Z mL=3;
   static Word loc("show(F<R,R>,Z,Color,V<R>,Color)");
   CPM_MA
   if (n<=1) n=2;
   Z i;
   Iv ivx=getIvX();
   R_Vector x(ivx,n);
   R_Matrix y(2,n);
   for (i=1; i<=n; i++){
      y[1][i]=x[i];
      y[2][i]=f(x[i]);
   }
   V<Color> vcf(1,cf);
   show(y,vcf);
// now vertical lines for selected x-values
// for analysis of curve behavior
   Z nx=xList.dim();
   if (nx==0){
      CPM_MZ
      return false;
   }
   else{
      Iv ivy=getIvY();
      R y1=ivy[1];
      R y2=ivy[2];
      for (i=1;i<=nx;i++){
         R xi=xList[i];
         if (ivx.contains(xi)) draw(R2(xi,y1),R2(xi,y2),cx);
      }
   }
   CPM_MZ
   return true; // trivial return
}

void  Graph::mark( V<R> const& xList, V<Color> const& cList)
{
   Z nx=xList.dim();
   if (nx==0) return;
   Iv ivx=getIvX();
   Iv ivy=getIvY();
   R y1=ivy[1];
   R y2=ivy[2];
   for (Z i=xList.b();i<=xList.e();++i){
      R xi=xList.cui(i);
      if (ivx.contains(xi))
         draw(R2(xi,y1),R2(xi,y2),cList(i,CYCLIC));
   }
}
```

```
void Graph::show(Z n, R pl, R pu,
                R f1(R z1), R f2(R z2), const Color& c)
{
   Z i;
   R h;
   if (n<=1) n=2;
   R_Vector x(n);
   R_Vector y(n);
   h=(pu-pl)/(n-1);
    R p_=pl;
   for (i=1; i<=n; i++){
      x[i]=f1(p_); y[i]=f2(p_); p_+=h;
   }
   show(x,y,c);
}

// common code for implementing the next two functions
// an attempt to use a function taking a function object as argument
// failed since Graph would be a parameter of such a function object
// and had to be constant in this role. (2001-4-20)

#define CPM_SHR_COD1\
   R2 cLowerLeft(XL,YL);\
   R2 cUpperRight(XU,YU);\
   Z2 pU_(iX(XU),jY(YU));\
   Z2 pL_(iX(XL),jY(YL));\
   R dy_=YU-YL;\
   if (!(dy_>0.)) cpmerror("Graph::forAll(...): dy_<=0.");\
   R dx_=XU-XL;\
   if (!(dx_>0.)) cpmerror("Graph::forAll(...): dx_<=0.");\
   Z nRow_=pL_.x2-pU_.x2+1;\
   if (!(nRow_>0)) cpmerror("Graph::forAll(...): !(nRow_>0)");\
   Z nCol_=pU_.x1-pL_.x1+1;\
   if (!(nCol_>0)) cpmerror("Graph::forAll(...): !(nCol_>0)");\
   Z i,j;\
   R yStep=dy_/nRow_;\
   R xStep=dx_/nCol_;\
   R xStart=cLowerLeft.x1;\
   R yStart=cUpperRight.x2;\
   xStart+=0.5*xStep;\
   yStart-=0.5*yStep;\
   R xAct=xStart;\
   R yAct=yStart;\
   for (j=pU_.x2;j<=pL_.x2;j++){\
      for (i=pL_.x1;i<=pU_.x1;i++){


#define CPM_SHR_COD2\
          putPixel(i,j,Color(vr,vg,vb));\
```

```
            xAct+=xStep;\
         }\
         xAct=xStart;\
         yAct-=yStep;\
      }\
      if (!noDisplay){\
         display(writeOnShow);\
      }

void Graph::show(const F<RR,V<R> >& f, R XL, R XU, R YL, R YU,
   bool noDisplay )
{
   CPM_SHR_COD1
          RR xy(xAct,yAct);
          V<R> val=f(xy);
          R vr=val[1];
          R vg=val[2];
          R vb=val[3];
   CPM_SHR_COD2
}

void Graph::show(const F<R2, R3 >& f, R XL, R XU, R YL, R YU,
   bool noDisplay )
{
   Z mL=1;
   Word loc("Graph::show(F<R2,R3>,R,R,R,R,bool)");
   CPM_MA
   CPM_SHR_COD1
          R2 xy(xAct,yAct);
          R3 val=f(xy);
          R vr=val[1];
          R vg=val[2];
          R vb=val[3];
   CPM_SHR_COD2
   CPM_MZ
}

void Graph::show(const R2_Func& fR, const R2_Func& fG, const R2_Func& fB,
   R XL, R XU, R YL, R YU, bool noDisplay )
{
   CPM_SHR_COD1
          RR xy(xAct,yAct);
          R vr=fR(xy);
          R vg=fG(xy);
          R vb=fB(xy);
   CPM_SHR_COD2
}

void Graph::show(const R2_Func& fR, const R2_Func& fG, const R2_Func& fB,
   const Response& rR, const Response& rG, const Response& rB,
```

```
   Iv roiX, Iv roiY, bool noDisplay )
{
   Z mL=3;
   Word loc("Graph::show(R2_Func,..Response,..Iv,Iv)");
   cpmmessage(mL,loc&" started");
   R XL=roiX.inf();
   R XU=roiX.sup();
   R YL=roiY.inf();
   R YU=roiY.sup();
   CPM_SHR_COD1
          RR xy(xAct,yAct);
          R vr=rR(fR(xy));
          R vg=rG(fG(xy));
          R vb=rB(fB(xy));
   CPM_SHR_COD2
   cpmmessage(mL,loc&" done");
}

#undef CPM_SHR_COD2

void Graph::show(R2_Func const& f, Response const& rsp, Iv roix, Iv roiy)
{
   Z mL=3;
   Word loc("Graph::show(R2_Func,Response,Iv,Iv)");
   CPM_MA
   Iv roiX=(roix.isVoid() ? getIvX() : roix);
   Iv roiY=(roiy.isVoid() ? getIvY() : roiy);
   R XL=roiX.inf();
   R XU=roiX.sup();
   R YL=roiY.inf();
   R YU=roiY.sup();
   CPM_SHR_COD1
      RR xy(xAct,yAct);
      R val=f(xy);
      Color c=rsp.color(val);
      putPixel(i,j,c);
          xAct+=xStep;
      }
      xAct=xStart;
      yAct-=yStep;
   }
   display(writeOnShow);
   CPM_MZ
}

#undef CPM_SHR_COD1


void Graph::show(F<C,C> const& f, R gamma, bool zeroIsZero)
{
```

```
   using CpmImaging::Color;
   R rMax=-1;
   R rMin=CpmRootX::hugeNumber;

   V<R> cx=centersX();
   V<R> cy=centersY();
   Z nx=dim_x();
   Z ny=dim_y();
   VV<C> val(nx,ny);
   Z i,j;
   C fz;
   // notice that our indexes start with 1
   for (i=1;i<=nx;i++){
      for (j=1;j<=ny;j++){
         C z(cx[i],cy[j]);
         fz=f(z);
         val[i][j]=fz;
         R r=fz.absVal();
         if (r>rMax) rMax=r;
         if (r<rMin) rMin=r;
      }
   }
   if (zeroIsZero) rMin=0;
   Iv range(rMin,rMax,true);
   Color::setGamma(gamma);
   for (i=1;i<=nx;i++){
      for (j=1;j<=ny;j++){
         fz=val[i][j];
         Color c(fz,range); // depends on gamma
         put_xy(i,j,c); // simpel indexing due to
            // a newly defined function put_xy
      }
   }
   display(writeOnShow);
}

void Graph::show(const R2_Func& fR, const R2_Func& fG, const R2_Func& fB)
{
   show(fR,fG,fB,CL.x1,CU.x1,CL.x2,CU.x2);
}

void Graph::show(const F<RR,V<R> >& f)
{
   show(f,CL.x1,CU.x1,CL.x2,CU.x2);
}

void Graph::mark(Vl< Vl<C> >const& mat, Iv range, Z method,
   bool faithful, bool periodic, R secFac, bool matrixstyle)const
{
   Z mL=3;
```

```
   Word loc("Graph::mark(Vl<Vl<C>>,Iv,Z,bool,bool,R,bool)");
   CPM_MA
   static bool firstrun=true;
   static Iv rangeAct;
   Z mx=mat.dim();
   cpmassert(mx>0,loc);
   Z my=mat[0].dim();
   cpmassert(my>0,loc);
      // it is assumed that the first index of mat corresponds to a
      // x-coordinate and the second index to either the x-coordinate of
      // a second particle and or the y-coordinate of the first particle
      // (then there is no second one)
   Z i,j;
// Renewing range if needed
   if ((range.isVoid()&& secFac<=0)||(firstrun && secFac>0)){
      R mm=-1;
      for (i=0;i<mx;++i){ //
         for (j=0;j<my;++j){
            R mij=mat.cui(i).cui(j).absVal();
            if (mij>mm) mm=mij;
          }
      }
      if (secFac<=0){
         rangeAct=Iv(0,mm);
      }
      else{
         cpmassert(firstrun==true,loc);
         firstrun=false;
         rangeAct=Iv(0,secFac*mm);
      }
   }
// checking method
   cpmassert(method==0||method==1,loc);
// Main loops
   R rx=1., ry=1.;
   if (faithful){ // we assume square pixels on screen
      R alpha_= (matrixstyle ? (1.*my)/mx : (1.*mx)/my);
         // aspect ratio of image
      R alpha=getAspRat(); // aspect ratio of frame
      if (alpha_ > alpha){ // image has to be placed full width
         rx=1.;
         ry=alpha/alpha_; // <= 1
      }
      else{ // image has to be placed full hight
         rx=alpha_/alpha; // <= 1
         ry=1.;
      }
   }
   Z nx,ny;
   nx=cpmround(getCol()*rx);
```

```
ny=cpmround(getLin()*ry);
   // Number of Graph-pixels to be used for rhe
   // image; in x-direction and y-direction respectively.
   // Thes numbers determine the pixel loops
cpmassert(nx>0,loc);
cpmassert(ny>0,loc);
R dpx=mx*1./nx, dpy=my*1./ny;
R dpxh=dpx*0.5, dpyh=dpy*0.5;
Z i1,i2,j1,j2;
R px,py,qx,qy,dx,dy,w1x,w2x,w1y,w2y;
if (matrixstyle){
   for (i=0,px=dpxh;i<nx;i++,px+=dpx){
      i1=(Z)px;
      if (i1==mx) i1--;
      if (method!=0){
         qx=i1+0.5;
         dx=px-qx;
         w2x=cpmabs(dx); // <= 1
         w1x=1-w2x;
         i2=dx>=0 ? i1+1 : i1-1;
      }
      for (j=0,py=dpyh; j<=ny; j++,py+=dpy){
         j1=(Z)py;
         if (j1==my) j1--;
         if (method!=0){
            qy=j1+0.5;
            dy=py-qy;
            w2y=cpmabs(dy); // <=1
            w1y=1-w2y;
            j2=dy>=0 ? j1+1 : j1-1;
            C v11=mat.cui(i1).cui(j1); // i1,j1 always in range
            C v12,v21,v22;
            if (periodic){
               v12=mat.cui(i1).cyc(j2);
               v21=mat.cyc(i2).cui(j1);
               v22=mat.cyc(i2).cyc(j2);
            }
            else{
               v12=mat.cui(i1).con(j2);
               v21=mat.con(i2).cui(j1);
               v22=mat.con(i2).con(j2);
            }
            R w11=w1x*w1y, w12=w1x*w2y, w21=w2x*w1y, w22=w2x*w2y;
            C z=v11*w11+v12*w12+v21*w21+v22*w22;
            img.putPel(j,i,Color(z,rangeAct).toRGB());
         }
         else{
            C z=mat.cui(i1).cui(j1);
            img.putPel(j,i,Color(z,rangeAct).toRGB());
         }
```

```
         }
      }
   } // matrixstyle
   else{ // !matrixstyle
      for (i=0,px=dpxh;i<nx;i++,px+=dpx){
         i1=(Z)px;
         if (i1==mx) i1--;
         if (method!=0){
            qx=i1+0.5;
            dx=px-qx;
            w2x=cpmabs(dx); // <= 1
            w1x=1-w2x;
            i2=dx>=0 ? i1+1 : i1-1;
         }
         for (j=ny-1,py=dpyh; j>=0; j--,py+=dpy){
            j1=(Z)py;
            if (j1==my) j1--;
            if (method!=0){
               qy=j1+0.5;
               dy=py-qy;
               w2y=cpmabs(dy); // <=1
               w1y=1-w2y;
               j2=dy>=0 ? j1+1 : j1-1;
               C v11=mat.cui(i1).cui(j1); // i1,j1 always in range
               C v12,v21,v22;
               if (periodic){
                  v12=mat.cui(i1).cyc(j2);
                  v21=mat.cyc(i2).cui(j1);
                  v22=mat.cyc(i2).cyc(j2);
               }
               else{
                  v12=mat.cui(i1).con(j2);
                  v21=mat.con(i2).cui(j1);
                  v22=mat.con(i2).con(j2);
               }
               R w11=w1x*w1y, w12=w1x*w2y, w21=w2x*w1y, w22=w2x*w2y;
               C z=v11*w11+v12*w12+v21*w21+v22*w22;
               img.putPel(i,j,Color(z,rangeAct).toRGB());
            }
            else{
               C z=mat.cui(i1).cui(j1);
               img.putPel(i,j,Color(z,rangeAct).toRGB());
            }
         }
      }
   } // matrixstyle
   CPM_MZ
}

bool Graph::mark(V< V<R2> > const& dat, V<Color> const& cl, R secFac)
```

```
{
   Z mL=3;
   Word loc("Graph::mark(V<V<R2> >, V<Color>)");
   CPM_MA
   static bool firstrun=true;
   static Z2 steps;
   if (secFac<=0){ // old functionality, so old code can't
      // get broken
      if (autoScale){
         bool b=setXY(dat);
         if (b==false){
            cpmwarning(loc&": no graph could be created");
            return false;
         }
         Z2 steps=adjXY();
         clearAndGrid(steps[1],steps[2]);
      }
      else{
         Word xL=cpm(CL.x1);
         Word xU=cpm(CU.x1);
         Word yL=cpm(CL.x2,3);
         Word yU=cpm(CU.x2,3);
         clearAndGrid(10,10);
      }
   }
   else{ // clean separation of new functionality controlled
      // by the new (2007-11-20) argument secFac
      if (!sclMem){ // no longer controlled by firstrun.
         // static data native to a function block are not flexible
         // they could be manipulated (e.g. reset) only by call of the
         // function; whereas Graph data can be well set by Graph member
         // functions.
         sclMem=true;
         bool b=setXY(dat,1.,secFac);
         if (b==false){
            cpmwarning(loc&": no graph could be created");
            return false;
         }
         steps=adjXY();
      }
      clearAndGrid(steps[1],steps[2]);
   }
   Vo<Color> clAct(cl);
   V<Color> exceptions(2);
   exceptions[1]=colorActive;
   exceptions[2]=gridColor;
   clAct=clAct.purge(exceptions);
      // clAct may be void after purge (2006-02-05)
   Z cAd=clAct.dim();
   if (cAd==0){
```

```
      cpmwarning(loc&
         ": color array was void after purging, replaced by RED"
      );
      clAct=V<Color>("",RED);
   }
   vector<R2>::const_iterator j,jF;
   for (Z i=dat.b();i<=dat.e();i++){
      Color ci=clAct(i,CYCLIC);
      vector<R2> dati=dat[i].std();
      j=dati.begin();
      jF=dati.end();
      while (j!=jF){ // elegant iteration
          R2 p1=*j++; // now j is the next one
          if (j!=jF) draw(p1,*j,ci); // j valid if used
      }
   }
   CPM_MZ
   return true;
}

/**************
void Graph::mark(Vl< Vl<R3> >const& mat, Z method,
   bool faithful, bool periodic)const
{
   Z mL=3;
   Word loc("Graph::mark(Vl<Vl<R3>>,Z,bool,bool,bool)");
   CPM_MA
   Z i,j,m=mat.dim(); // old notation m=mx ,n=my
   cpmassert(m>0,loc);
   Z n=mat[0].dim();
   cpmassert(n>0,loc);
      // m, n as in matices
// checking method
   cpmassert(method==0||method==1,loc);
// Main loops
   R rm=1., rn=1.; // new
   if (faithful){ // we assume square pixels on screen
      R alpha_= (1.*n)/m ;
         // aspect ratio of image
      R alpha=getAspRat(); // aspect ratio of frame
      if (alpha_ > alpha){ // image has to be placed full width
         rm=1.;
         rn=alpha/alpha_; // <= 1
      }
      else{ // image has to be placed full hight
         rm=alpha_/alpha; // <= 1
         rn=1.;
      }
   }
   Z nm,nn;
```

```
nm=cpmround(getCol()*rm);
nn=cpmround(getLin()*rn);
   // Number of Graph-pixels to be used for rhe
   // image; in x-direction and y-direction respectively.
   // Thes numbers determine the pixel loops
cpmassert(nm>0,loc);
cpmassert(nn>0,loc);
R dpm=m*1./nm, dpn=n*1./nn;
R dpmh=dpm*0.5, dpnh=dpn*0.5;
Z i1,i2,j1,j2;
R px,py,qx,qy,dx,dy,w1x,w2x,w1y,w2y;
for (i=0,px=dpxh;i<nx;i++,px+=dpx){
      i1=(Z)px;
      if (i1==mx) i1--;
      if (method!=0){
         qx=i1+0.5;
         dx=px-qx;
         w2x=cpmabs(dx); // <= 1
         w1x=1-w2x;
         i2=dx>=0 ? i1+1 : i1-1;
      }
      for (j=0,py=dpyh; j<=ny; j++,py+=dpy){
         j1=(Z)py;
         if (j1==my) j1--;
         if (method!=0){
            qy=j1+0.5;
            dy=py-qy;
            w2y=cpmabs(dy); // <=1
            w1y=1-w2y;
            j2=dy>=0 ? j1+1 : j1-1;
            R3 v11=mat.cui(i1).cui(j1); // i1,j1 always in range
            R3 v12,v21,v22;
            if (periodic){
               v12=mat.cui(i1).cyc(j2);
               v21=mat.cyc(i2).cui(j1);
               v22=mat.cyc(i2).cyc(j2);
            }
            else{
               v12=mat.cui(i1).con(j2);
               v21=mat.con(i2).cui(j1);
               v22=mat.con(i2).con(j2);
            }
            R w11=w1x*w1y, w12=w1x*w2y, w21=w2x*w1y, w22=w2x*w2y;
            R3 z=v11*w11+v12*w12+v21*w21+v22*w22;
            img.putPel(j,i,Color(z).toRGB());
         }
         else{
            R3 z=mat.cui(i1).cui(j1);
            img.putPel(j,i,Color(z).toRGB());
         }
```

```
            }
        }
    CPM_MZ
}
***************/
```

87 cpmgreg.h

```
//? cpmgreg.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_GREG_H_
#define CPM_GREG_H_

/*

    Functions for handling Gregorian calendar dates

*/

#include <cpmangle.h>

namespace CpmTime{ // Quantities related to calendar, date, time, clock

using namespace CpmStd;
using CpmRoot::R;
using CpmRoot::Z;
using CpmRoot::Word;
using CpmGeo::Angle;

//////////////////////////// enum TimeScale ////////////////////////////////

enum TimeScale {TD, UT}; // presenly not used in this general-purpose
   // (non-astronomical) version of Greg in order not to need the more
   // astronomical files.

   // TD = Temps dynamique, UT = universal time

//////////////////////////////// struct TimeStyle //////////////////////////

struct TimeStyle{ // how to interprete time with respect to the globe
   TimeScale tsc;
   Angle lambdaOfTimeMeridian;
   TimeStyle(void):tsc(UT),lambdaOfTimeMeridian(Angle()){}
   TimeStyle(TimeScale s, const Angle& a):tsc(s),
      lambdaOfTimeMeridian(a){}
};

////////////////////////////// class Greg ////////////////////////////////

class Greg{  // time, date, and Gregorian Calendar
   // An instance of class Greg determines a point in time
   // coded according to the Gregorgian calendar (and according to the
   // Julian calendar for dates prior to the introduction of the
   // Gregorian calendar). No reference to a particular time scale (UT
```

```
   // or TD) or to a time zone (e.g. MEZ or MESZ ) is implied

private:

   static Z yearOffset;
      // The 'century' which is understood for 2 decimal year input
      // e.g. 2000 for dates in the 21th century
      // May be changed via setCentury(). Presently initialized as 2000.

   static const R mjdOffset;
      // JD=MJD+mjdOffset, mjdOffset=2400000.5

protected:

// primary (independent) information
   Z y;         // year
   Z m;         // month
   Z d;         // day
   Z h;         // hour
   Z min;       // minute
   R s;         // second

// secondary (dependent) information (function of the primary
// information)
   R mjd;         // analog to the MJD without reference to a concrete
                // time scale such as UT or TD
   void update(void);
      // updates dependent information (i.e. mjd)

   void makeDate(void);
      // adjusts the rest of data members to present value of mjd

public:

   typedef Greg Type;
   CPM_IO
   CPM_NAM_V(Greg)
   virtual Greg* clone()const{ return new Greg(*this);}

// parts of the day (the unit) which are often needed

   static const R hour;
   static const R minute;
   static const R second;

   static R mjd_(Z y, Z m, Z d);
      // From Montenbruck, Pfleger: Astronomie mit dem Personal Computer
      // p. 12. Translated from PASCAL to C

   static Z trunc(R x);
```

```
      // See O. Montenbruck, Grundlagen der Ephemeridenrechnung S. 49

   static void caldat(Z mjd, Z& yy, Z& mm, Z& dd);
      // From Montenbruck, Pfleger: Astronomie mit dem Personal Computer
      // p. 13
      // Translated from PASCAL to C

   void setCentury( Z century)
      // setCentury(20) sets the value to be added to a short
      // year, such as 95, to 1900, thus resulting in 1995.
      // Notice that this coincides with the conventional enumeration
      // of centuries.
      // Century should be positive, we are not interested in using
      // Gregorian Calendar in times much prior to its introduction !
      {
         yearOffset=(century-1)*100;
      }

   Greg(void);
      // Default constructor, creates 2000-1-1 0:0:0

   Greg(Z year, Z mm, Z dd);
      // mm has to be in the range [1,12]
      // dd is arbitrary (e.g. 94-12-31 = 95-1-0 = 95-2-(-31) )
      // if year<99 it will be understood as year+yearOffset.

   explicit Greg(R date){ setDate(date,false);}
     // date is assumed to be of the form year.mmdd
     // where year is the integer year number e.g. 2006 or
     // 1905 (no shortened numbers like 50 for 1950
     // or 6 for 2006 !) Only the first 4 decimal places get
     // used (no fractions of a day).

// getting calendar data, always clear from code

   Z getYear(void)const{ return y;}

   Z getMonth(void)const{ return m;}

   Z getDay(void)const{ return d;}

   Z getHour(void)const{ return h;}

   Z getMinute(void)const{ return min;}

   R getSecond(void)const{ return s;}

   R getMJD(void)const{ return mjd;}

   R getJD(void)const{ return mjd+mjdOffset;}
```

```
   Word getWeekDay(void)const;

   R yearsSince1900(void)const{ return 50.+(mjd-33281.)/365.2422;}
   R yearsSince2000(void)const{ return (mjd-51544.)/365.2422;}
   R years(void)const{ return 2000+yearsSince2000();}

   Word getDate(bool verbose=true)const;
      // "yyyy-mm-ddThh:mm:ssZ" ( ISO6801 time stamp )
      // a compact representation of a date in which the newline commands
      // of printOut() would cause trouble

   Word getCode(Z n=14)const;
      // returns a characterization of a point in time which
      // is compact enough to be used as an appendix to file names.
      // It consists of the n trailing characters of the string
      // yyyymmddhhmmss. So the default value for n means that
      // a condensed form of the date will be given. For
      // n=8 one gets a characterisation of the time within the
      // present day.

   Word getCodeWord(Z n, Z p=5);
      // returns a sequence of n digits which is determined by the
      // instance of call and may be used as random signature
      // method: multiply mjd by 10^p, form the integer part
      // and take the last n digits of the result. Thus, the larger
      // p, the faster the result changes with time. The frequency
      // of change is 10^p per day. Thus p=5 means 10^5 changes
      // in 86400 s, i.e. 1.157 changes per second. This is a
      // reasonable time resolution for differentiating between
      // susequent runs of a program.

// setting Greg according to numbered days input (Julian date and
//   modified Julian date

   void setMJD(R mjd){ this->mjd=mjd; makeDate();}
      // adjusts *this according to MJD input.
      // In constructor form, this function would enable automatic
      // conversion from Rd to Greg and finally would lead to
      // Greg + Greg being defined (since Greg + R is defined). This
      // is not desirable, only Greg - Greg is OK.

   void setMJD(R mjd, const TimeStyle& ts);
      // adjusts *this according to MJD input. The correspondence
      // between what the function does and the values
      // UT or TD of the component ts.tsc is logical if we
      // consider mjd as referring to TD. Then ts.tsc=UT means that
      // a deltaT subtraction has to be done to get a result in
      // UT or the civil time of a time zone specified by
      // ts.lambdaOfTimeMeridian
```

```
   void setJD(R jd){ mjd=jd-mjdOffset; makeDate();}
      // adjusts *this according to JD input.

   R toMJDTD(const TimeStyle& ts)const;
      // returns MJD (as TD) from the Greg instance *this where ts
      // tells how to interprete *this
      // (e.g. as civil time of a particular time zone)

   void now(void);
      // sets *this according to the internal clock of the computer
      // at the moment of function call. Notice that shifting this
      // point in time according to needs can easily be done
      // by the arithmetics of Greg, so that no shift argument is needed
      // for this function.
      // e.g.
      // TimeStyle ts=...;
      // Greg g;
      // g.now();
      // g+=10*Greg::minute;
      // R t=g.toMJDTD(ts);
      // Earth e;
      // e.update(t);
      // will represent the state of the earth 10 minutes after the
      // function call if ts properly reflects the characteristics of the
      // computer clock

friend R operator -(const Greg& d1, const Greg& d2);
      // time span in days, notice that a corresponding + operator
        // makes no sense

   Greg& operator +=(R t);
      // shifting a point in time by t days

   Greg& nextMonth_();
      // shifting a point in time to the same day, minute, second
      // but one month in calander later (shift may be 28 days,
      // 30 days or 31 days according to the calender month we are
      // in). Name ends in '_' since this is a mutating function.

   Greg& nextDay_(){ mjd+=1.; makeDate(); return *this;}
      // shifting a point in time to by one day
      // in). Name ends in '_' since this is a mutating function.

   Greg& operator -=(R t);
      // shifting a point in time by -t days

   Greg operator +(R t){ Greg res=*this; return res+=t;}

   Greg operator -(R t){ Greg res=*this; return res-=t;}
```

```
   void setDate(R date, bool yShort=true);
      // Sets the values of y, m, d from date=Year.MMDD .
      // If yShort=true and Year<99 the integer number
      // yearOffset will be added to y.

   void setWatch(R watch);
      // Sets the values of h, min, s from watch=HH.MMSSSS

   bool readFormatted(istream& inStream);
   bool writeFormatted(ostream& outStream)const;
};

} // namespace

#endif
```

88 cpmgreg.cpp

```
//? cpmgreg.cpp
//? Status of work 2008-10-25.
//?
#include <time.h>
#include <iomanip>

#include <cpmgreg.h>

using namespace::CpmRoot;
using CpmTime::Greg;
using CpmTime::TimeStyle;

/***************** basic calendar routines *************************/

R Greg::mjd_(Z y, Z m, Z d)
// From Montenbruck, Pfleger: Astronomie mit dem Personal Computer
// p. 12
// Translated from PASCAL to C
{
   R a;
   Z b;
   a=10000.*y+100.*m+d;
   if (m<=2){
      m+=12;
      y-=1;
   }
   if (a<=15821004.1){
      b=-2+(y+4716)/4-1179;
   }
   else{
      b=y/400-y/100+y/4;
   }
   a=365.0*y-679004.0;
   R res=a+b+floor(30.6001*(m+1))+d;
    return res;
}

Z Greg::trunc(R x)
// See O. Montenbruck, Grundlagen der Ephemeridenrechnung S. 49
{
   Z sig=(x>=0 ? 1 : -1);
    x=(sig>=0 ? x : -x);
   Z y=(Z)(0.001+floor(x));
    return (sig>=0 ? y : -y);
}

void Greg::caldat(Z mjd, Z& yy, Z& mm, Z& dd)
```

```
// From Montenbruck, Pfleger: Astronomie mit dem Personal Computer
// p. 13
// Translated from PASCAL to C
{
   Z b,d,f;
   R jd,jd0,c,e;
   jd=2400000.5+mjd;
   jd0=trunc(jd+0.5);
   if (jd0<2299161.0){
      b=0;
      c=jd0+1524.0;
   }
   else{
      b=trunc((jd0-1867216.25)/36524.25);
      c=jd0+(b-trunc(b/4))+1525.0;
   }
   d=trunc((c-122.1)/365.25);
   e=365.0*d+trunc(d/4);
   f=trunc((c-e)/30.6001);
   dd=(Z)(trunc(c-e+0.5)-trunc(30.6001*f));
   mm=(Z)(f-1-12*trunc(f/14));
   yy=(Z)(d-4715-trunc((7+mm)/10));
}


Z Greg::yearOffset=2000;

const R Greg::mjdOffset=2400000.5;

const R Greg::hour=1./24;

const R Greg::minute=Greg::hour/60;

const R Greg::second=Greg::minute/60;


void Greg::update(void)
   // adjusts mjd according to rest of data members
{
   mjd=mjd_(y,m,d)+h*hour+min*minute+s*second;
}

void Greg::makeDate(void)
   // adjusts rest of data members according to mjd
{
   Z mjdInt=(Z)mjd;
   R rest=mjd-mjdInt;
   caldat(mjdInt,y,m,d); // now y,m,d are valid
   R h1=rest*24;
   h=(Z)h1;
```

```
   R min1=(h1-h)*60;
   min=(Z)min1;
   s=(min1-min)*60;
}

Greg::Greg(void)
{
   y=2000;
   m=1;
   d=1;
   h=0;
   min=0;
   s=0.;
//   mjd=mjd_(y,m,d);
//   cpmmessage("mjd(2000.1.1)"&cpmwrite(mjd));
   mjd=51544;
}

Greg::Greg(Z yy, Z mm, Z dd):y(yy),m(mm),d(dd),h(0),min(0),s(0)
{
   if (m<1 || m>12) cpmerror("Greg","month must be 1,...,12");
   update();
   if (d<1 || d>28) makeDate();
}

Greg& Greg::nextMonth_()
{
   m++;
   if (m>12){
      y++;
      m=1;
   }
   update();
   if (d>28) makeDate(); // this makes a valid date even when called
   // e.g. for March 31, where April 31 would not be a correct
   // date.
   return *this;
}

bool Greg::prnOn(ostream& str)const
{
   cpmwat;
   cpmp(y);
   cpmp(m);
   cpmp(d);
   cpmp(h);
   cpmp(min);
   cpmp(s);
   return true;
}
```

```
bool Greg::scanFrom(istream& str)
{
   cpms(y);
   cpms(m);
   cpms(d);
   cpms(h);
   cpms(min);
   cpms(s);
   update();
   return true;
}

Word Greg::getDate(bool verbose)const
// improved to represent ISO time stamp by Thomas Dera
{
   using std::setw;
   using std::setfill;
   ostringstream ost;
   if (verbose) ost<<"yyyy-mm-ddThh:mm:ss ";
   ost<<setw(4)<<setfill('0')<<y<<"-"
      <<setw(2)<<setfill('0')<<m<<"-"
      <<setw(2)<<setfill('0')<<d<<"T"
      <<setw(2)<<setfill('0')<<h<<":"
      <<setw(2)<<setfill('0')<<min<<":"
      <<setw(2)<<setfill('0')<<s<<"Z";
   return Word(ost.str());
}

Word Greg::getCode(Z n)const
{
   using std::setw;
   using std::setfill;
   ostringstream ost;
   ost<<setw(4)<<setfill('0')<<y
      <<setw(2)<<setfill('0')<<m
      <<setw(2)<<setfill('0')<<d
      <<setw(2)<<setfill('0')<<h
      <<setw(2)<<setfill('0')<<min
      <<setw(2)<<setfill('0')<<s;
   return Word(ost.str()).tail(n);
}

bool Greg::readFormatted(istream& inStream)
{
   cout<<endl<<
   "Entering a point in time in terms of the Gregorian calendar";
   R d,h;
   cout<<"\n Enter the date in the format yy.mmdd or yyyy.mmdd:  ";
   inStream>>d;
```

```
   cout<<"\n Enter the time in the format hh.mmss:  "; inStream>>h;
   setDate(d);
   setWatch(h);
   update();
   return true;
}

bool Greg::writeFormatted(ostream& str)const
//  working, changed 2006-02-06
{
   Word yS=Word::write(y,4);
   Word mS=Word::write(m,2);
   Word dS=Word::write(d,2);
   Word GregS=yS&"-"&mS&"-"&dS&" ";
   Word hS=Word::write(h,2);
   Word minS=Word::write(min,2);
   Word sS=toWord(s,"%2.1f");
   Word watchS=hS&":"&minS&":"&sS&" Z";
   Word res=GregS&watchS;
   str<<res;
   return (str!=0);
}

R CpmTime::operator - (const Greg& g1, const Greg& g2)
{
   return g1.mjd-g2.mjd;
}

Greg& Greg::operator +=(R t)
{
   mjd+=t;
   makeDate();
   return *this;
}

Greg& Greg::operator -=(R t)
{
   mjd-=t;
   makeDate();
   return *this;
}

void Greg::setDate(R date, bool yShort)
// Year, month, date to be set are taken from a real number: for instance
// 43.0506 means the 6th May in 1943. If shortdate is 0, this would be
// the 6th May in 43 p.Chr. This is easier in input than any scheme of
// separating the parts from a string. If there are more decimals,
// these are interpreted as hhmmss.
// Notice that y is always the correct (unabbreviated) year - only in
// input the abbreviation is possible
```

```
{
   const R eps=0.0001;
   Z sig=(date >=0. ? 1 : -1);
   date*=sig; // now date is positive
   y=(Z)date;
   date-=y; // fractional part of the absolute value of date
   y*=sig; // now the year carries its sign
   if (yShort){
      if (y<=99) y+=yearOffset;
   }
   date*=100;
   m=(Z)(date+eps);
   date-=m;
   date*=100;
   d=(Z)(date+eps);
   h=0;
   min=0;
   s=0.;
   update(); // added 99-11-27
}

void Greg::setWatch(R watch)
// sets only hours, minutes and seconds. watch=14.03567 means a watch
// display of 14 h 03 min 56.7 s
{
   const R eps=0.0001;
   if (watch<0) cpmerror("Greg::setWatch(R): argument should be >=0");
   h=(Z)(watch+eps);
   watch-=h; // fractional part of the absolute value of watch
   watch*=100;
   min=(Z)(watch+eps);
   watch-=min;
   watch*=100;
   s=watch;
   update(); // added 99-11-27
}

Word Greg::getWeekDay(void)const
{
   Z weekDay=(Z)getMJD();
   weekDay=weekDay%7;
    Word res;

   switch (weekDay){
      case 0: res="Wendsday";break;
      case 1: res="Thursday";break;
      case 2: res="Friday";break;
      case 3: res="Saturday";break;
      case 4: res="Sunday";break;
      case 5: res="Monday";break;
```

```
      case 6: res="Tuesday";break;
   }
   return res;
}

void Greg::now( void)
{
   Z firstrun=1;
   if (firstrun){
   #if !defined(CPM_POSIX)
      _putenv("TZ=GMT0");
      _tzset();
   #else
      putenv("TZ=GMT0");
      tzset();
   #endif
      firstrun=0;
   }
   time_t t;
   time(&t);
   tm* all;
   all=gmtime(&t); // due to putenv() this gives the normal conversion
      // without a time zone shift
   s=all->tm_sec;
   min=all->tm_min;
   h=all->tm_hour;
   d=all->tm_mday;
   m=1+all->tm_mon;
   y=all->tm_year; // comes out as 99 for 1999, and 100 for 2000

   if (y>=90 /* && y<=99 */) y+=1900; // Windows NT gives 100 in 2000
      // this means that the rule to add 1900 is not changed
    update();
    if (cpmverbose>=4){
        cout<<endl<<"s="<<s;
        cout<<endl<<"min="<<min;
        cout<<endl<<"h="<<h;
        cout<<endl<<"d="<<d;
        cout<<endl<<"m="<<m;
        cout<<endl<<"y="<<y;
        cout<<endl<<"mjd="<<mjd;
    }
}

// The two TimeStyle functions, very similar logic

R Greg::toMJDTD(const TimeStyle& ts)const
      // returns MJD (as TD) from the Greg instance *this where ts
      // tells how to interprete *this
      // (e.g. as civil time of a particula´r time zone)
```

```
{
    R shift=ts.lambdaOfTimeMeridian.toDeg(0);
    shift/=360.; // now it is in days
    R res=mjd;
    res+=shift; // now res can be compared with MJD of UT or TD
        // i.e. there is no longer a time zone contribution
//    if (ts.tsc==UT) res=Earth::mjdToTD(res);
    return res;
}

void Greg::setMJD(R mjdI, const TimeStyle& ts)
      // adjusts *this according to MJD input. The correspondence
      // between what the function does and the values
      // UT or TD of the component ts.tsc is logical if we
      // consider mjd as referring to TD. Then ts.tsc=UT means that
      // a deltaT subtraction has to be done to get a result in
      // UT or the civil time of a time zone specified by
      // ts.lambdaOfTimeMeridian
{
    mjd=mjdI;
 //   if (ts.tsc==UT)  mjd=Earth::mjdToUT(mjd);
    R shift=ts.lambdaOfTimeMeridian.toDeg(0);
    shift/=360.; // now shift is in days
    mjd-=shift; // now referring to the local meridian
    makeDate();
}

Word Greg::getCodeWord(Z n, Z p)
      // returns a sequence of n digits which is determined by the
      // instance of call and may be used as random signature
{
   now();
   R f=::pow(R(10),p);
   Z cod=(Z)floor(mjd*f);
   Word res=cpmwrite(cod);
   return res.tail(n);
}
```

89 cpmgroupgraph.h

```
//? cpmgroupgraph.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_GROUPGRAPH_H_
#define CPM_GROUPGRAPH_H_
/*
    Description: directed graphs for organizing the
      communication between parallel processes.
*/
#include <cpmv.h>

namespace CpmGeo{

   using CpmRootX::Z1;
   using CpmArrays::V;

   template<class Group>
   class GroupGraph{ // directed graph with group elements attached
      // to edges. Has nothing to do with
      // CpmGraphics::Graph

      Z1 n;
         // number of vertices;

      V< V<Z1> >c;
         // c[i][j]=1 iff (i,j) is a (directed) edge of the graph.
         // thus c is the connection matrix

      V< V<Group> > g;
         // g[i][j] should have a usefull value if (i,j) is an edge
         // of the graph

      static Z cutNeg(Z n){ return n>0 ? n : 0;}
         // trivial tool for safely defining the constructor

   public:
      typedef GroupGraph Type;
      virtual GroupGraph* clone()const{ return new  GroupGraph(*this);}
      CPM_IO_V
         // IO should be virtual in order to allow
         // additional attributes in derived classes
// all functions test the validity of their arguments and
// stop via cpmerror for unvalid ones.

explicit GroupGraph(Z n=0);
         // a graph with n vertices and no edges
```

```
      void addEdge(Z i, Z j, const Group& gr=Group() )
      { c[i][j]=1; g[i][j]=gr;}
          // makes (i->j) an edge and associates gr
          // with this edge

      void addEdgeSym(Z i,Z j, const Group& gr=Group() )
      { c[i][j]=1; g[i][j]=gr; c[j][i]=1; g[j][i]=!gr;}
         // makes (i->j) and (j->i) edges
         // Group has to define inversion

      void remEdge(Z i, Z j){ c[i][j]=0; g[i][j]=Group();}
         // 'removes' an edge

      void remEdgeSym(Z i, Z j)
         { c[i][j]=0; g[i][j]=Group(); c[j][i]=0; g[j][i]=Group();}
         // self-explanatory

      Z dim()const{ return n;}

      bool close(Z i,Z j)const{ return c[i][j]==1;}
      Group group(Z i, Z j)const { return g[i][j];}
   };

template<class Group>
GroupGraph<Group>::GroupGraph(Z n_):
n(cutNeg(n_)),c(n,V<Z1>(n)), g(n,V<Group>(n)){}

template<class Group>
bool GroupGraph<Group>::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("GroupGraph",str);
   cpmp(n);
   cpmp(c);
   cpmp(g);
   return true;
}

template<class Group>
bool GroupGraph<Group>::scanFrom(istream& str)
{
   cpms(n);
   cpms(c);
   cpms(g);
   return true;
}

} // namespace

#endif
```

90 cpmh.h

```
//? cpmh.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_H_H_
#define CPM_H_H_
/*

   Description: class H of quaternions.
      H could be used for a more conventional treatment of rotations.
      In a final form, the treatment of two-dimensional and
      3-dimensional geometry should probably be based on the field C
      and H respectively. Finally also 4-dimensional Minkowski geometry
      had to be added.

   History: Started 02-1-23

*/

#include <cpmlinalg.h>

namespace CpmLinAlg{

   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmRoot::Word;
   using std::pair;

//////////////////////////////// class H ////////////////////////////////
// Quaternions ( H in honor of Hamilton). Arithmetics is implemented
// less schematically than in R4. It aims at utmost efficiency just as
// in was done for complex numbers C. However, no reference to complex
// arithmetics occurs here.

class H{  //  H  quaternions
   typedef H Type;
   R a0;
   R3 a;

public:
// constructors

   H():a0(0.),a(){}
   // default constructor

explicit H(R a0_):a0(a0_),a(){}
   // embedding R
```

```
explicit H(const R3& a_):a0(0.),a(a_){}
   // embedding R3

   H(R a0_, const R3& a_):a0(a0_),a(a_){}
   // general constructor

   H(R a0_, R a1, R a2, R a3):a0(a0_),a(a1,a2,a3){}
   // constructor for very efficient code (which avoids to
   // call R3-functions

explicit H(const R4& r):a0(r.x1),a(r.x2,r.x3,r.x4){}
   // conversion from R4

// access
   R& operator[](Z i);
      // no exceptions; always the best match of i in {1,2,3,4}
      // is actually used

   const R& operator[](Z i)const;
      // no exceptions; always the best match of i in {1,2,3,4}
      // is actually used

   R4 toR4()const
      { return R4(a0,a.x1,a.x2,a.x3);}
      // conversion to quadruples of R's

// arithmetics within H
// operator +
   H& operator +=(const H& h)
   { a0+=h.a0; a+=h.a; return *this;}

   H operator +(const H& h)const
   { return H(a0+h.a0, a+h.a);}

// operator -
   H& operator -=(const H& h)
   { a0-=h.a0; a-=h.a; return *this;}

   H operator -(const H& h)const
   { return H(a0-h.a0, a-h.a);}

   H operator -(void)const
   { return H(-a0,-a); }

// operator *
   H operator *(const H& h)const;
      // returns  H(a0*h.a0-(a|h.a), a*h.a0+h.a*a0+crsPrd(a,h.a)).
      // Hunting for efficiency to present implementation does only
      // one function call: constructor H(R,R,R,R).
```

```
   H& operator*=(const H& h)
   {  return *this=operator*(h);}

// operator /

   H inv(void)const;

   H operator !(void)const
      { return inv();}

   H operator /(const H& h)const
      { return operator*(h.inv());}

   H& operator /=(const H& h)
      { return operator*=(h.inv());}

//  conjugation
   H con(void)const
      { return H(a0,-a);}

   H operator~(void)const
      { return H(a0,-a);}

// arithmetics with R

   H& operator *=(R r)
   { a0*=r; a*=r; return *this;}

   H operator *(R r)const
   { return H(a0*r,a*r);}

// quaternions and rotations

   R3 operator *(const R3& x)const;
      // action of quaternions on R3 vectors. See e.g.
      // E. Peschl: Analytische Geometrie, BI Hochschultaschenbuecher
      // 15/15a, 1961 p. 121 Satz 69.
      // There h*x:=h*(0,x)*h.con() assuming h.norm()=1.
      // Replacing con() by inv(), as is done in the present code,
      // eliminates the need for this assumption.

  static H fromRot(R phi, const R3& axis);
      // generating a normalized quaternion (the norm of which is 1)
      // by rotation angle and rotation axis.
      // Notice that a constructor of the natural signature is already
      // present, so that the most usual implementation as a constructor
      // is not viable. The axis needs not to be a unit vector, but only
      // the corresponding unit vector determines the axis to be taken.
      // The 'angle' phi will be cut to [0,2*Pi) prior to usage
```

```
   pair<R,R3> rot()const;
      // returns angle and axis. In order to find these quantities
      // efficiently, *this needs not to be normalized. We just consider
      // the rotation which belongs to (*this)/absVal() without
      // actually
      // doing this normalization

// metric quantities

   R absVal(bool careful=true)const
      { return toR4().absVal(careful);}

   R norm()const
      // this 'norm' plays an important role in quaternions;
      // one has h1.norm()*h2.norm()=(h1*h2).norm()
      // Not a norm in the sense of normed linear spaces
      // could be written as a0*a0+(a|a)
      { return a0*a0+a.x1*a.x1+a.x2*a.x2+a.x3*a.x3;}

   R normalize();
      // turns *this into a normalized quaternion
      // i.e. absVal()==norm()==1.
      // and returns the absVal() of the original in order not to
      // loose information. Notice that normalizing has to be done
      // by deviding by absVal(); dividing by norm() would not
      // work.

   R disVal(const H& h)const
      { return toR4().disVal(h.toR4());}

// standard interface

   CPM_IO
   CPM_ORDER
   Z dim()const{ return 4;}
   Word nameOf(void) const{ return Word("H");}
};

} // namespace

#endif
```

91 cpmh.cpp

```
//? cpmh.cpp
//? Status of work 2008-10-25.
//?
/*
   Description: cpmh.cpp see cpmh.h
*/

#include <cpmh.h>

using namespace CpmRoot;
using namespace CpmLinAlg;

//////////////////////////// H ////////////////////////////////////////////
R normArc(R arc)
// converts arbitrarily large arcs (positive
// or negative) back to an arc in the range (-pi,pi]
{
   const R period=2*Pi;
   const R i_period=1./period;
   R rev=arc*i_period;
   rev-=floor(rev);
    if (rev>0.5) rev-=1.;
   return (rev*period);
}

R H::normalize()
{
   R n=absVal();
   if (n>0){
      R nInv=1./n;
      a0*=nInv;
      a*=nInv;
   }
   return n;
}

R& H::operator[](Z i)
{
   if (i>=3) return a.x3;
   else if (i==2) return a.x2;
   else if (i==1) return a.x1 ;
   else return a0;
}

const R& H::operator[](Z i)const
{
   if (i>=3) return a.x3;
```

```
   else if (i==2) return a.x2;
   else if (i==1) return a.x1 ;
   else return a0;
}

H H::fromRot(R phi, const R3& axis)
{
   R phiHalf=normArc(phi)*0.5;
   R b0=::cos(phiHalf);
   R3 b=axis;
   b.normalize();
   R s=::sin(phiHalf);
   b*=s;
   return H(b0,b);
}

bool H::prnOn(ostream& str)const
{
   if (!CpmRoot::writeTitle("H",str)) return false;
   cpmp(a0);
   cpmp(a.x1);
   cpmp(a.x2);
   cpmp(a.x3);
   return true;
}

bool H::scanFrom(istream& str)
{
   cpms(a0);
   cpms(a.x1);
   cpms(a.x2);
   cpms(a.x3);
   return true;
}

Z H::com(const H& h)const
{
   return toR4().com(h.toR4());
}

H H::operator *(const H& h)const
   // return  H(a0*h.a0-(a|h.a), a*h.a0+h.a*a0+crsPrd(a,h.a));
   // would be more transparent, but probably less efficient
{
   return
   H(a0*h.a0-a.x1*h.a.x1-a.x2*h.a.x2-a.x3*h.a.x3,
     a.x1*h.a0 + h.a.x1*a0 + a.x2*h.a.x3-a.x3*h.a.x2,
     a.x2*h.a0 + h.a.x2*a0 + a.x3*h.a.x1-a.x1*h.a.x3,
     a.x3*h.a0 + h.a.x3*a0 + a.x1*h.a.x2-a.x2*h.a.x1);
}
```

```
H H::inv(void)const
// same logic as inv(R) for error handling
{
   static const Z maxMes=100;
   static Z mes=1;
   R n=norm();
   if (n==0.){
      goto DIVBYZERO;
   }
   else{
      R nInv=1./n;
      return H(a0*nInv,a*(-nInv));
   }

DIVBYZERO:

   if (mes==maxMes){
      mes++; // needed !
      cpmmessage("H::inv(): argument is 0 ... messages discontinued");
   }
   if (mes<maxMes){
      mes++;
      cpmwarning("C::inv(): argument is 0;  H() returned");
   }
   return H();
}

pair<R,R3> H::rot()const
{
   R2 x(a0,a.absVal());
      // by this method we don't need the normalized form of
      // *this! This is rather elegant and efficient
   R2 rphiHalf=x.carToPol();
   R phiHalf=rphiHalf.x1;
   R3 an=a;
   an.normalize();
   return pair<R,R3>(2*phiHalf,an);
}

R3 H::operator *(const R3& x)const
{
   return (operator*( H(x)*inv() )).a;
}
```

92 cpmhistogr.h

```
//? cpmhistogr.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_HISTOGR_H_
#define CPM_HISTOGR_H_

/*
  cpmhistogr.h
  Description: This class includes functions to make multimage-histograms,
      show multimage-histograms
      and find min and max values of the histogramms

*/
#include <cpmrmatrix.h>
#include <cpmframe.h>
#include <cpmrfunc.h>

namespace CpmImaging{

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmLinAlg::R2;
   using CpmArrays::VV;
   using CpmFunctions::R_Func;
   using CpmGraphics::Frame;
   using CpmGeo::Iv;

class Histogram { // histograms: generation, visualization, analysis
   Z nPlanes;
      // number of planes of the histogram
   Iv range;
      // Iv(valMin,valMax); helps if available also in this form
   R valMin;
      // Minimum of Histogram (normaly 0)
   R valMax;
      // Maximum of Histogram (normaly 255)

   V<R> binValues;
      // has nBins components of equi-distant x-values

   VV<Z> values;
      // Matrix contains the values of the histogram
      // values[i][j] is the number of values belonging to
      // plane i and bin number j
   Z nBins;
      // number of sectors of the histogram
```

```
   R binWidth;
      // width of the sectors of the histogram

   Word topic;
      // a title

public:
   Histogram( Z nPlanesI=1, Z nBinsI=1, R valMinI=0, R valMaxI=255,
      const Word& theme=Word());
      //constructor for a multi-histogram
      // nPlanes: Number of distributions curves
      // nBins:   Number of sectors in each distribution
      // valMin:  value belonging to the left boundary of the lowest
      // sector
      // valMax:   value belonging to the right boundary of the highest
      // sector

   R2 minMax(R min=0.01,R max=0.01)const;
      // Let res be the return value.
      // res.x1 is the smallest x for which ...
      // 2005-08-29 not clear what the function really does.
      // Has to be revisited. Action !
      // for minimum: find the first bin of the histogram containing
      // the fraction min of all Pixels.
      // for maximum: find the last sector of the histogram including
      // a special rate of pixels (declared in max) max = 0.01   means
      // find the last sector include 1% of all Pixels
      // the function conforms this values from sector numbers to absolut
      // Pixelvalues (normaly between 0 and 255) and returns this values

   R2 minMax(Z plane, R min=0.01,R max=0.01)const;
      // for minimum: find the fist sector of the histogram including
      // a special rate of pixels (declared in min) min = 0.01   means
      // find the first sector include 1% of all Pixels
      // for maximum: find the last sector of the histogram including
      // a special rate of pixels (declared in max) max = 0.01   means
      // find the last sector include 1% of all Pixels
      // the function conforms this values from sector numbers to absolut
      // Pixelvalues (normaly between 0 and 255) and returns this values

   void incrementValue(Z h,Z SectNum);
      // increments a sector of a histogram
      // h:   Plane of the Histogram (RGB)
      // SectNum:   Number of Sector whitch should be incremented

   void show(Frame& g, bool cumulative=true)const;
      // shows the histogram *this in cumulative form.
      // As pointed out to me by Thomas Dera, the cumulative form
      // is much more suitable for screen display ('more quiet curves')
      // than a direct display of bins.
```

```
      // g: Screen area in which the histogram is shown

   void put(Z h, R value, Z Method=0);
      // find the sector of a histogram to which value belongs, and
      // increase the count of this sector by one.
      // h:         Plane (RGB)
      // value:   signal value (normaly between 0 and 255)
      // you can select with the parameter Method how to handle out of
      // range values.
      // if Method=0, for all values out of borders the function put the
      // value to the borders
      // if Method=1 all values out of borders the function return 0
      // in this case the function put will forget this values

   R_Func getBrightValues(Z pl, R fracL=0.9, R fracU=0.001, R p=4)const;
      // a tone scale transformation function which makes the brightest
      // pixels white and all significantly darker pixels black.
      // Considers only plane pl for this procedure.
      // The pixel values that shall be identified as bright are by
      // definitiuon such that a fraction fracL of all pixels is
      // darker and a fraction fracU is brighter as the 'bright pixels'.
      // The tone scale transformation is such that the 'bright pixels'
      // get values near 255. The larger p, the harder the contrast of
      // the
      // scaled image.

   Iv firstPeak(Z pl=1, R frac=0.5, Z nL=1, Z nR=1);
      // returns the first bin (according to the normal numbering of
      // bins) that is a local maximum and has at least a hight
      // frac*total maximum.
      // Further it includes nL bins to the left and nR bins to the right
      // All these bins are united to a value range that is returned as
      // an intervalo in value space.
      // Considers only plane pl for this procedure.
};

} // namespace

#endif
```

93 cpmhistogr.cpp

```
//? cpmhistogr.cpp
//? Status of work 2008-10-25.
//?
#include <cpmhistogr.h>
#include <cpmgraph.h>
#include <cpmvectormore.h>

using CpmImaging::Histogram;
using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::Word;
using CpmLinAlg::R2;
using CpmArrays::R_Vector;
using CpmArrays::R_Matrix;
using CpmArrays::Vo;
using CpmFunctions::R_Func;
using CpmFunctions::F5;
using CpmGraphics::Frame;
using CpmGraphics::Graph;
using CpmGeo::Iv;

Histogram::Histogram(Z nPlanesI, Z nBinsI, R valMinI, R valMaxI,
   const Word& theme ):

   nPlanes(nPlanesI),
   range(valMinI,valMaxI),
   valMin(range[1]),
   valMax(range[2]),
   nBins(nBinsI),
   values(nPlanesI,nBinsI),
   topic(theme)
{
   cpmassert(nBins>0,"Histogram::Histogram(...)");
   binValues=range.centers(nBins);
   binWidth=range.absVal()/nBins;
}

R2 Histogram::minMax(R min,R max)const
{
   Z i,h;
   Z l=nPlanes;
   R_Vector all(l);
   R_Vector bMin(l);
   R_Vector bMax(l);
   R_Vector minSect(l);
   R_Vector maxSect(l);
   R large=1e33;
```

```
   R minS=large;
   R maxS=-large;
   R2 minmax;
   R sum=0;

   for(i=1;i<=nBins;i++) for(h=1;h<=l;h++) all[h]+=values[h][i];
   for(h=1;h<=l;h++) bMin[h]=all[h]*(1-min);
   for(h=1;h<=l;h++) bMax[h]=all[h]*(1-max);

   for(h=1;h<=l;h++)
   {
      i=nBins;
      while(sum<bMin[h])
      {
         sum+=values[h][i];
         i--;
         minSect[h]=i;
      }
   }
   sum=0;
   for(h=1;h<=l;h++)
   {
      i=1;
      while(sum<bMax[h])
      {
         sum+=values[h][i];
         i++;
         maxSect[h]=i;
      }
   }
   for(h=1;h<=1;h++)
   {
      if(minSect[h]<minS) minS=minSect[h];
      if(maxSect[h]>maxS) maxS=maxSect[h];
   }
   minmax.x1=valMin+minS*binWidth;
   minmax.x2=valMin+maxS*binWidth;

   return minmax;
}

R2 Histogram::minMax(Z pl,R min,R max)const
{
   Z i;
   R all=0;
   R bMin;
   R bMax;
   R minS;
   R maxS;
   R2 minmax;
```

```
   R sum=0;

   for(i=1;i<=nBins;i++) all+=values[pl][i];
   bMin=all*(1-min);
   bMax=all*(1-max);
   i=nBins;
   while(sum<bMin){
      sum+=values[pl][i];
      i--;
      minS=i;
   }
   sum=0;
   i=1;
   while(sum<bMax)      {
      sum+=values[pl][i];
      i++;
      maxS=i;
   }
   minmax.x1=valMin+minS*binWidth;
   minmax.x2=valMin+maxS*binWidth;

   return minmax;
}

void Histogram::incrementValue(Z h,Z SectNum)
{
     values[h][SectNum]++;
}

void Histogram::show(Frame& fr, bool cumulative)const
{
   using namespace CpmImaging;
   const R xPosRel=0.6;
   R_Vector xV(binValues);
   Z i,j;
   R_Matrix valR(nPlanes,nBins);
      // since suitable graphical representation is available
      // for R_Vector and R_Matrix
   for (i=1;i<=nPlanes;i++)
      for (j=1;j<=nBins;j++)
         valR[i][j]=values[i][j];
   if (cumulative)
      for (i=1;i<=nPlanes;i++)
         valR[i]=CpmArrays::accumulate(valR[i]);
   V<Color> vc(nPlanes);
   if (nPlanes==1)
      vc[1]=RED;
   else if (nPlanes==2){
      vc[1]=RED;
      vc[2]=LIGHTGREEN;
```

```
   }
   else if (nPlanes==3){
      vc[1]=RED;
      vc[2]=LIGHTGREEN;
      vc[3]=LIGHTBLUE;
   }
   else vc=rainBow; // has only 8 colors, no problem since colors are
   // used
      // cyclically if there are less than nPlanes
   Graph g(fr);
   g.setColorActive(Color(Z(0),Z(0),Z(0)));
      // background
   g.setGridTextColor(Color(Z(200),Z(200),Z(200)));
      // grid text should be much brighter than the background
   g.show(xV,valR,vc,topic,xPosRel);
}

void Histogram::put(Z h, R value, Z Method)
{
   R x=nBins*(value-valMin)/(valMax-valMin);
      // for value in the proper range, this varies from 0 to nBins
   if (value<valMin){
      if (Method==0) values[h][1]++;
   }
   else if (value>valMax){
      if (Method==0) values[h][nBins]++;
   }
   else{
      values[h][range.quant(value,nBins)]++;
   }
}

namespace{

R auxFunc(const R& x,
   const R& xL, const R& xU, const R& yL, const R& yU, const R& ex)
{
   if (x<=xL) return yL;
   else if (x>xU) return yU;
   else{
      R p=(x-xL)/(xU-xL);
      p=pow(p,ex);
      return yL+p*(yU-yL);
   }
}

}

R_Func Histogram::getBrightValues(Z pl, R fracL, R fracU, R p)const
{
```

```
   R yL=0;
   R yU=255;
   R2 mm=minMax(pl,fracL,fracU);
   R xL=mm[1];
   R xU=mm[2];
   return F5<R,R,R,R,R,R,R>(xL,xU,yL,yU,p)(auxFunc);

}

Iv Histogram::firstPeak(Z pl, R frac, Z nL, Z nR)
{
   static Word loc("Histogram::firstPeak(Z pl, R frac, Z nL, Z nR)");
   Vo<Z> val=values[pl];
   Z n=val.dim();
   cpmassert(n==nBins,loc);
   Z valMax=val.sup();
   Z valOk=cpmround(valMax*frac);
   Z i,i0=0;
   for (i=1;i<n;i++){
      if (val[i+1]<=val[i] && val[i]>=valOk){
         i0=i;
         break;
      }
   }
   R bhw=0.5*binWidth;
   Z iL=CpmRootX::sup(i0-nL,Z(1));
   Z iR=CpmRootX::inf(i0+nR,n);
   return Iv(binValues[iL]-bhw,binValues[iR]+bhw);
}
```

94 cpmhomunitvecxcpp.h

```
//? cpmhomunitvecxcpp.h
//? Status of work 2008-10-25.
//?
// common implementation for Vec::homUnitVec(Z,Z,Z,R,R)

namespace{
// tools for implementing function Vec::homUnitVec(Z, Z).
// Motion of particles on the surfase of a sphere
// This is a nice mechanical system, with a compact configuration space
// without boundaries.
// Taking into account the serving role of the concept, we do not
// built in the flexibility (e.g. different masses for different
// particles) that a treatment as a mechanical system would suggest.

class Pb{ // class for defining single partices
   Vec x_;
      // position relative to the center of the sphere
   AxVec omg_;
      // omega, angular velocity with respect
      // to the center of the sphere
public:
   static R eps_; // regularizing the Coulomb force
   CPM_NAM(Pb)
   Pb(Vec const& x=Vec(1,0,0)): x_(x), omg_(){}
     // argument needs to be a unit-vector (no adjustment for efficiency)
   Vec x()const{ return x_;}
   void rec_(AxVec const& torque, R dt){ omg_+=(torque*(dt));}
      //: reaction
      // to a 'Drehstoß'
   AxVec tor(Pb const& p)const;
     // torque felt by *this due to the presence of particle p
   void step_(R dt);
      // free motion for a time span dt
   void stop_(){ omg_=AxVec();}
      // stopping motion
};

R Pb::eps_=1e-2;

void Pb::step_(R dt)
{
   AxVec rot=omg_.exp(dt);
   x_*=rot;
}

AxVec Pb::tor(Pb const& p)const
{
```

```
   Vec to_p=p.x_-x_;
   R r=to_p.normalize_();
   R fac=-1./(r*r+eps_); // not singular
   Vec f=to_p*fac;
   return x_.cross(f);
 }

class PbSys{ // particle system
   V<Pb> vp_;
public:
   Vec operator[](Z i){ return vp_[i].x();}
   PbSys(V<Pb> const& vp):vp_(vp){}
   void stop_(){ for (Z i=vp_.b();i<=vp_.e();++i) vp_[i].stop_(); }
   void step_(R dt);
};

void PbSys::step_(R dt)
// direct midpoint integrator, no regularization
{
   Z i,j,n=vp_.dim();
   R tau=dt*0.5;
   for (i=1;i<=n;++i) vp_[i].step_(tau);
   for (i=1;i<=n;++i){
      AxVec tor_i;
      for (j=1;j<=n;++j){
         if (j!=i) tor_i+=vp_[i].tor(vp_[j]);
      }
      vp_[i].rec_(tor_i,dt);
   }
   for (i=1;i<=n;++i) vp_[i].step_(tau);
}

} // namespace

V<Vec> Vec::homUnitVec(Z n, Z nSteps, Z stopPeriod,
   R dtFac, R eps)
{
   R dt=dtFac/sqrt((R)n);
   Z i,ic=0;
   V<Pb> p(n);
   for (i=1;i<=n;++i){
      Vec xi=Vec::ranUnitVec();
      p[i]=Pb(xi);
   }
   Pb::eps_=eps;
   PbSys ps(p);
   i=0;
   while (ic<nSteps){
      i++;
      ps.step_(dt);
```

```
      if (i%stopPeriod==0){
         ic++;
         ps.stop_();
      }
   }
   V<Vec> res(n);
   for (i=1;i<=n;++i) res[i]=ps[i];
   return res;
}
```

95 cpmimage.h

```
//? cpmimage.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_IMAGE_H_
#define CPM_IMAGE_H_
/*
    Description: Defines template class ImageUM<T>
      Notice: Reading images from file is done in MultImageUM not here!
*/
#include <cpmgraph.h>
#include <cpmc.h>
#include <cpmhistogr.h>
#include <cpmfftmore.h>
#include <cpmfa.h>
#include <cpmva.h>
#include <cpmimagingtools.h>

namespace CpmImaging{

   using namespace CpmStd;

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::Rh;
   using CpmRoot::C;
   using CpmRoot::L;

   using CpmRootX::B;

   using CpmArrays::VV;
   using CpmArrays::X2;
   using CpmArrays::R_Vector;
   using CpmArrays::R_Matrix;

   using CpmFunctions::F;
   using CpmFunctions::Fa;

   using CpmLinAlg::R2;
   using CpmLinAlg::Z2;

   using CpmFourier::fft;

   using CpmGraphics::Frame;
   using CpmGraphics::Graph;
   using CpmGraphics::Img24;

   using CpmGeo::Iv;
```

```
template<class T>
class Image : public ImagingTools, public VV<T> {
// real-valued image, gray image with high dynamic range

   // The meaning of pixel indexes is in accordance with matrix indexing:
   // i.e. increasing the first index means moving downwards
   // increasing the second index means moving to the right.

   // In order for most of the following functions be meaningful it is
   // assumed that T is a type such that for t of type T statements
   //
   // R x=t;
   // R y=...; t=y;
   // make sense

   // From a data structure point of view T could also be a
   // multi-component type. Since earlier implementations of the basic
   // matrix template (now VV<T> did not allow this, I created the type
   // MultiImage which allows several image planes.

   R1 aspRat;    // aspect ratio = image width / image hight
      // (e.g. 1.5 for 35 mm film format) is n/m for square pixels
      // aspRat>1 for landscape and <1 for portrait
   typedef VV<T> Base;
   typedef Image Type;
public:

// constructors

   Image(void): VV<T>(1,1){aspRat=(R)Base::dim2()/Base::dim1();}

   Image(Img24 const&);
      // constructs a B/W image from an instance of the most
      // basic one of my image classes

   Image(Z mI, Z nI): VV<T>(mI,nI){aspRat=(R)Base::dim2()/Base::dim1();}
      // constructs a black image mI pixels high and nI pixels wide

   Image(Z mI, Z nI, T const& t): VV<T>(mI,nI,t)
      {aspRat=(R)Base::dim2()/Base::dim1();}

   Image(const VV<T>& y): VV<T>(y){aspRat=(R)Base::dim2()/Base::dim1();}

   Image(Z mI, Z nI, R xL, R yL, R dx, R dy,
      const F<C,C>& f, const F<C,R>& g):VV<T>(mI,nI)
      // see sampleComplexFunction(...) for details
      {   sampleComplexFunction(xL,yL,dx,dy,f,g);   }

   Image(Z mI, Z nI, R xL, R yL, R dx, R dy,
```

```
      const F<C,R>& f):VV<T>(mI,nI)
      // see sampleComplexFunction(...) for details
      {   sampleComplexFunction(xL,yL,dx,dy,f);   }

   explicit Image(VV<C> const& sp);
      // construction from a matrix of fourier components
      // in the format as they are returned by function
      // fourierTransform

// access functions

   Z getLin()const{return Base::dim1();}
      // number of lines = height of image
   Z getCol()const{return Base::dim2();}
      // number of columns = width of image
   Z h()const{ return Base::dim1();}
      // height of image
   Z w()const{ return Base::dim2();}
      //  width of image

   T getPix(Z i, Z j)const;
     // getting pixel values, if pixels outside the image are
     // assessed, the image appears 'continued as constant'

   T getLocalMean(Z, Z, const Mask& )const;
      // get a weighted mean of pixel value (i,j) and its neighbors
      // where the contributing neighbors and their weights are
      // determined by the Mask argument

   void putPix( Z i, Z j, const T& t);
     // setting pixel values, if pixels outside the image are
     // assessed nothing is done

   R getAspRat(void)const{ return aspRat;}
      // returns the aspect ratio
      // aspRat=(R)dim2()/dim1();

   void setAspRat(const R& r){ aspRat=r;}
      // sets the aspect ratio

 // others

   Image<T> interpolate(Z m, Z n, Z method)const;
      // returns a m*n Image created from *this by a interpolation
      // method selected by the parameter method.
      // method =1 means bilinear interpolation and method=0 means
      // pixelation. The latter is useful if the number of pixels is to
      // be compared to expectations.
      // Also intended for m << height and n << width.
```

```
Image<T> intPolAux(Z m, Z n, Z method)const;
   // auxiliar interolation method intended to be used
   // only for m > height/2, n > width/2

T operator()(R x, R y, Z method=1)const;
   // returns the value at a point with respect to a
   // normal coordinate system (origin lower-left, x-axis
   // to the right, y-axis vertically ascending. The upper-right
   // image corner has coordinates x=1, y=1. This system
   // will be referred to as base system. Pixelation, or
   // bilinear interpolation according to 'method' as in previous
   // function interpolate(). For points outside the basic unit-square
   // the image is replicated periodically. This assures that an
   // arbitrary transformation of the plane will always define a image
   // with image content in the basic unit square. This function is
   // implemented such that all calls to that function for one image
   // take approximately the same time. So no preparatory calculations
   // for the first call are allowed. This is essential for forming a
   // Cpm-function out of an image (see class R2_Func). Details for
   // interpolation: The unit square is divided into (m*n) rectangular
   // pixels of identical size. The image values (pixel values) are
   // considered to be valid function values (of a real function of
   // two real variables) at the center locations of the pixels.
   // Function values at intermediate positions are defined by
   // interpolation (dependent on 'method') from these values. Notice
   // that it would be more natural to consider the pixel value as a
   // integral of some unknown distribution over a pixel area and not
   // as a sampled function value. Probably such an understanding also
   // would lead to bilinear interpolation.

T evaluate( R x, R y)const;
   // returns the value at a point with respect to a
   // normal coordinate system (origin lower-left, x-axis
   // to the right, y-axis vertically ascending. The upper-right
   // image corner has coordinates x=1, y=1. This system
   // will be referred to as base system. Pixelation, no
   // interpolation.

R2 pixelCenter(Z i, Z j)const;
   // Let i,j be matrix indexes. Then the return value
   // is the pixel center in the base system (see previous
   // function for definition).

R2 minMax(R xL=0, R xU=1, R yL=0, R yU=1)const;
   // Let res be the return value. Then res.x is the minimum
   // and res.x2 the maximum of the pixel values of *this
   // This refers to the part of the image defined by xL,xU,yL,yU.
   // The pre-assigned values correspond to the whole image.

R_Vector statistics(R xL=0, R xU=1, R yL=0, R yU=1)const;
```

```
   // Let res be the return value. Then res[1] is the minimum
   // and res[2] is the mean value and res[3] is the the maximum
   // of the pixel values of *this.
   // This refers to the part of the image defined by
   // xL,xU,yL,yU. The pre-assigned values correspond to the whole
   // image.
   // Future extensions could also give standard deviation and median.

Histogram histogram(Z nBins=16,R BDown=0, R BUp=255)const;
   // makes a histogramm of an Image
   // nBins:   number of Sectors
   // BDown:   minimal value (normaly 0)
   // BUp:      maximal value (normaly 255)

void gamma(R val1, R val2, R valMin, R valMax, R gammaI);
   // Changes *this in a manner that is usually called
   // a tone-scale transformation and, more specifically,
   // a gamma conversion.
   // Notice that the tone scale control in function show
   // renders gamma transformations superfluous in all cases where
   // only the appearence on screen matters.
   // Let the pairs (Val1,Val2), and (ValMin,ValMax) result from
   // ordering the pairs (val1,val2), and (valMin, valMax)
   // (to be sure: Val1 < Val2), valMin <= valMax).
   // Uses a suitably shifted gamma curve f(.) to achieve
   //    f(Val1)=ValMin
   //    f(Val2)=ValMax.
   // Gamma has the usual meaning: gamma=1 for linear
   // curve, gamma <1 for enhancing shadow contrast, gamma > 1 for
   // enhancing contrast in full light.

void gamma(R valMin, R valMax, R gammaI=1.);
   // see gamma(R val1, R val2, R valMin, R valMax, R gammaI);
   // Uses a suitably shifted gamma curve for achieving a range of
   // gray-values from valMin to valMax. Gamma has the usual meaning:
   // gamma=1 for linear curve, gamma <1 for enhancing shadow
   // contrast,
   // gamma > 1 for enhancing contrast in full light.

void gamma(R gammaI);
   // see gamma(R val1, R val2, R valMin, R valMax, R gammaI);
   // Uses a gamma curve such that the range of gray-values remains
   // unchanged. Gamma has the usual meaning:
   // gamma=1 for linear curve, gamma <1 for enhancing shadow
   // contrast,
   // gamma > 1 for enhancing contrast in full light.

void sampleComplexFunction(const Graph& gr, C (*f)(const C&),
   R (*g)(const C&, const R_Vector&), const R_Vector& p );
   // not as flexible as the next function.
```

```
void sampleComplexFunction(R xL, R yL, R dx, R dy,
   const F<C,C>& f, const F<C,R>& g);
   // Changes *this into an image with the same number of pixels
   // and values determined by f and transformed by g.
   // We consider the area of the complex plane C determined by
   //
   //      xL <= Re(z) <= xU,   xU = xL + dx
   //      yL <= Im(z) <= yU,   yU = yL + dy
   //
   // This area is subdivided in m rows and n colums of rectangular
   // pixels. The pixel values of the image are g(f(zPixCen)) where
   // zPixCen is the center of the pixel under consideration.
   // The correspondence between position in C and in the
   // resulting Image is matrix oriented: Image(1,1) corresponds
   // to the upper left corner of the rectangle in C.

void sampleComplexFunction(R xL, R yL, R dx, R dy, const F<C,R>& f);
   // same as last function but f here corresponds to the composition
   // f&g there

void insertPattern
   (R xL, R yL, R dx, R dy, const F<C,T>& f, const F<C,B>& g);
   // as previous function, but pixels are set only if g(z)=TRUE
   // otherwise pixel value is not changed

void frame(const T& t);
   // a 1-pixel boundary line with value t is drawn in the picture

void mark(Frame& fr, Response const& rsp, Z method=0,
   Z faithful=1)const;
   // Writes the image *this on the Frame fr.
   // This does not display the image. fr.display() needs to
   // be called for showing the image on screen.
   // The object rsp transforms the pixel values
   // into values of type Color. The various member functions
   // of CpmImaging::Response give a wide range of options
   // for this transformation. E.g. rsp.setMethod(6) enables
   // pseudo-color representation of gray values.
   // The starting point so far is always the C++-provided
   // conversion from type T to type R so that Image<R> is the
   // most natural instantiation of the present template class.
   // The frame fr also determines the number of pixels to
   // be displayed. Since this may not coincide with the number of
   // pixels of the image, a suitable transformation has to be done.
   // This is controlled via parameters method and faithful.
   // The parameter 'method' controls pixel rendition:
   //    0 normal pixelation
   //    1 bilinear interpolation.
   // If the parameter 'faithful' is 1, the image will be placed in
```

```
      // such a manner to the upper left position that its aspect ratio
      // is correctly reproduced on the screen. Otherwise it is made
      // filling the whole frame.

   void mark(Frame& fr, R2 limits, Z method=0, Z faithful=1)const
      // Grayscale representation such lowest pixel value
      // (after C++-provided conversion to R)
      // corresponds to limits[1] and the highest color value to
      // limits[2].
      // Gamma is 1 since images created from outer sources (via Img24)
      // e.g. from electronic cameras, scanners, internet are normally
      // coded in a manner that does not assume a gamma transformation
      // before display. If the image is created as a visualization tool
      // for non-image data one should consider using the previous
      // function for display that allows also pseudo-color
      // representation
      // of real data
   {
      Response rsp(limits[1],limits[2]);
      mark(fr,rsp,method,faithful);
   }

   R2 mark(Frame& fr, Z method=0, Z faithful=1, R gamma=1.)const
      // As above, but limits is adjusted according to the value
      // range of the image. This value range is returned.
      {
         R2 limits=minMax();
         Response rsp(limits[1],limits[2]);
         rsp.setGamma(gamma);
         mark(fr,rsp,method,faithful);
         return limits;
      }

// trivial ordering not used but needed to allow Image as an template
// argument of V<> in its normal form
   Z operator < (const Image<T>& a){ return 0;}

   Image<T> cut(R xL, R xU, R yL, R yU)const;
      // reduces the size of the image by cutting the borders. All
      // arguments have to be in the interval [0,1].
      // The part of the image which is left after cut has normal
      // coordinates (x,y) with respect to the original which satisfy
      //   xL <= x <= xU
      //   yL <= y <= yU
      // Normal coordinate system: origin lower-left, x-axis
      // to the right, y-axis vertically ascending. The upper-right
      // image corner has coordinates x=1, y=1.

   Image<T> cut(Z2 cLL, Z2 cUR)const;
      // similar to the previous function. Here the lower left corner
```

```
   // and the upper right corner are given as pixel positions in
   // graphics format. Most recent version allows to use any pair of
   // diagonally opposed corners (without that, the function is very
   // prone to error)

Image<T> cut(Z i1, Z i2, Z j1, Z j2)const;
   // returning the subimage with pixels characterized by their
   // boundaries in matrix indexing scheme:
   // getPix(i,j) i1<=i<=i2, j1<=j<=j2

Image<T> clip(R clipL, R clipU)const;
   // x.clip(a,b) takes values between a*x.min() and b*x.max()

Image<T> transformToneScale(const F<T,T>& f)const;
   // returns a new image that takes pixel value f(val)
   // where the original image had pixel value val.

Image<T> conv(const R_Matrix& a, Z si=0, Z sj=0);
   // R_Matrix a is taken as an convolution kernel with a(ci+si,cj+sj)
   // considered the center of the kernel.
   // Here
   // ci := 1+dim1(a)/2
   // cj := 1+dim2(a)/2
   // which is the actual center if the dimensions of a both are odd
   // which is the normal case.
   // No normalization will be done on a, since this can be done more
   // flexibly in advance.

Image<T> conv(const Mask& a)const;
   // returns the image res for which
   // res[i][j]=getLocalMean(i,j,a) for all i,j
   // 'modern' form of convolution

Image<T>& operator *= (R r);
   // changing the image into one pixelwhise multilpied by r

Image<T>& operator += (R r);
   // changing the image into one where pixelwhise r is added

VV<C> fourierTransform()const;
   // returns the fourier-transformed image as it is used in
   // function fourierFilter

Image<T> fourierFilter(const Fa<R2,R>& resp)const;
   // multiplies the fourier transform by function rep which
   // takes meaningful arguments in [0,1]x[0,1], where 0 corresponds
   // to frequency 0 and 1 to Nyquist frequency along the
   // respective axis.
   // The negative frequencies (which are not independent) are treated
   // automatically in a consistent manner.
```

```
      // T is assumed to allow meaningful two-way conversion to R.
      // The resulting image has lines and columns equal to powers of
      // 2. The input image is not required to have this property.
      // For calculating its fourier transform we fill the missing pixels
      // in by 0.

   Image<T> fftFilter(const F<C,C>& resp)const;
      // similiar as above but using the more natural
      // rectangle [-0.5,0.5]x[-0.5,0.5] and operating on
      // the amplitude instead of the intensity. Assumes T=R.

   Image<T> cut(const Rect& rect)const;
      // creates a rectangular cut-ut of the image
      // See class Rect for details on the description of the
      // rectangular area.

   R norm1(void)const;
       // mean value of absolute pixel values

   VV<T> toVV()const{ return (Base)(*this);}

   Img24 toImg24()const;

   void horLine(Z x1,Z x2, Z y, T c);
      // fast way to draw a horizontal line

   static T biLin(R f11, R f12, R f21, R f22,
      R w11, R w12, R w21, R w22, Z method);
   // law for bi-linear interpolation (for method==1, else pixelation)

};

template <class T>
Image<T>::Image(const Img24& img)
{
   cpmmessage("Image(Img24) started");
   Z m=img.getLin();
   Z n=img.getCol();
   Image<T> res(m,n);
   Z i,j;
   R prfrac=1./m;
   for (i=1;i<=m;i++){
      cpmprogress(" putPix()-loop in constructor",i*prfrac);
      for (j=1;j<=n;j++){
         rgb pixel=img.getPel(j-1,i-1);
         res.putPix(i,j,pixel.gray());
      }
   }
   *this=res;
   cpmmessage("Image(Img24) done");
```

```
}

template<class T>
Image<T>::Image(VV<C> const& sp)
{
   Z mL=2;
   static Word loc("Image<>::Image(VV<C>)");
   CPM_MA
   Z m=sp.dim1(),n=sp.dim2(),i,j;
   cpmassert(m%2==0 && n%2==0,loc);
   VV<C> res=fft(sp,-1);
   *this=Image<T>(m,n);
   for (i=1;i<=m;++i) for (j=1;j<=n;++j){
      (*this)[i][j]=(res[i][j])[1];
   }
   CPM_MZ
}

template<class T>
T Image<T>::getPix(Z i, Z j)const
{
   Z m=Base::dim1(),n=Base::dim2();
   if (i<1) i=1;
   if (j<1) j=1;
   if (i>m) i=m;
   if (j>n) j=n;
   return (*this)[i][j];
}

template<class T>
void Image<T>::putPix( Z i, Z j, const T& t)
{
   if (i<1) return;
   if (j<1) return;
   Z m=Base::dim1(),n=Base::dim2();
   if (i>m) return;
   if (j>n) return;
   (*this)[i][j]=t;
}

template<class T>
T Image<T>::evaluate(R xI, R yI)const
   // returns the value at a point with respect to a
   // normal coordinate system (origin lower-left, x-axis
   // to the right, y-axis vertically ascending. The upper-right
   // image corner has coordinates x=1, y=1.
    // no interpolation involved
{
   Z i,j;
   const R tiny=1e-7;
```

```
   if (xI<=0) xI=tiny;
   if (xI>=1) xI=1-tiny;
   if (yI<=0) yI=tiny;
   if (yI>=1) yI=1-tiny;
   Z m=Base::dim1(), n=Base::dim2();
   i=(Z)(yI*m);  // i=0,...m-1
   j=(Z)(xI*n);  // j=0,...n-1
      // i,j discretized y,x
   j++; // j=1,...n
   i=m-i; // i=m,...1
      // now i,j matrix indexes
   return (*this)[i][j];
}

template<class T>
R2 Image<T>::pixelCenter(Z i, Z j)const
   // Let i,j be matrix indexes. Then the return value
   // is the pixel center in the base system
{
   Z m=Base::dim1(),n=Base::dim2(); // a silly compiler asked for
      // scope resolution
   R dx=1./n, dy=1./m;
   Z iy=m-i; //  iy= 0,...m-1
   Z jx=j-1; //  ix=0,...n-1
   R2 res(dx*(jx+0.5),dy*(iy+0.5));
   return res;
}

template<class T>
T Image<T>::biLin(R f11, R f12, R f21, R f22,
   R w11, R w12, R w21, R w22, Z method)
{
   if (method==1){
      R p11=w11*w21;
      R p12=w11*w22;
      R p21=w12*w21;
      R p22=w12*w22;
      return (T)(f11*p11 + f12*p12 + f21*p21 + f22*p22);
   }
   else{
      Z i= w11<=w12 ? 2 : 1;
      Z j= w21<=w22 ? 2 : 1;
      if (i==1 && j==1) return (T)f11;
      else if (i==1 && j==2) return (T)f12;
      else if (i==2 && j==1) return (T)f21;
      else return (T)f22;
   }
}

template<class T>
```

```
Image<T> Image<T>::intPolAux(Z mI, Z nI, Z method)const
{
   Z mL=3;
   cpmmessage(mL,"Image<T>::interpolate(...) started. m="&
      CpmRoot::toWord(mI)&" n="&CpmRoot::toWord(nI));
   Image<T> res(mI,nI);
   Z i,j;
   R f11,f12,f21,f22;
   Z m=h();
   Z n=w();
   if (m==mI && n==nI) return *this;
   Iv iv(0,1);
   V<R> ci=iv.centers(m);
   V<R> cj=iv.centers(n);
   V<R> cIi=iv.centers(mI);
   V<R> cIj=iv.centers(nI);
   for (i=1;i<=mI;i++){
      R xiI=cIi[i];
      X2<R2,Z2> qi=iv.intPolInfo(xiI,m);
      R2 wi=qi.c1();
      Z2 zi=qi.c2();
      R wi1=wi.x1;
      R wi2=wi.x2;
      Z i1=zi.x1;
      Z i2=zi.x2;
      for (j=1;j<=nI;j++){
         R xjI=cIj[j];
         X2<R2,Z2> qj=iv.intPolInfo(xjI,n);
         R2 wj=qj.c1();
         Z2 zj=qj.c2();
         R wj1=wj.x1;
         R wj2=wj.x2;
         Z j1=zj.x1;
         Z j2=zj.x2;
         f11=(R)(*this)[i1][j1];
         f12=(R)(*this)[i1][j2];
         f21=(R)(*this)[i2][j1];
         f22=(R)(*this)[i2][j2];
         res[i][j]=(T)biLin(f11,f12,f21,f22,wi1,wi2,wj1,wj2,method);
      }
   }
   return res;
}

template<class T>
Image<T> Image<T>::interpolate(Z mI, Z nI, Z method)const
{
   Z mL=3;
   cpmmessage(mL,"Image<T>::interpolate(...) started. m="&
      CpmRoot::toWord(mI)&" n="&CpmRoot::toWord(nI));
```

```
   Z m=getLin();
   Z n=getCol();
   if (mI>m/2 && nI>n/2){ // traditional case, interpolation to
      // similar or larger resolution
      return Image<T>::intPolAux(mI,nI,method);
   }
   else{ // new case: interpolating to lower resolution
         // based on averaging; works well for characters
         // in fonts
      R mIinv=1./mI, nIinv=1./nI;
      Z mf=cpmround(m*mIinv), nf=cpmround(n*nIinv);
      Z mA=mI*mf, nA=nI*nf;
      Image<T> resA=Image<T>::intPolAux(mA,nA,method);
         // image of similiar resolution than the original
         // but allowing schematic averaging
      Image<T> res(mI,nI);
      Z i,j,iR,jR,k,l;
      R sum,nmInv=1./(mf*nf);
      for (i=1;i<=mI;++i){
         iR=1+(i-1)*mf;
         for (j=1;j<=nI;++j){
            sum=0;
            jR=1+(j-1)*nf;
            for (k=0;k<mf;++k){
               for (l=0;l<nf;++l){
                  sum+=resA[iR+k][jR+l];
               }
            }
            res[i][j]=sum*nmInv;
         }
      }
      return res;
   }
}

namespace{
// tools for next function
inline void dec(Z& i){ if (i>1) i--;}
inline void inc(Z& i, Z m){ if (i<m) i++;}
   // for fast access, this checks correctness of indexes only where
   // leaving the range may happen, a inline version of getPix() would
   // check always.
}
// main part
template<class T>
T Image<T>::operator()(R x, R y, Z method)const
// should be as efficient as possible, thus let us accept the ugly code
// structure. In a typical application I observes 8% speed loss by
// replacing
// (*this)[i][j] by getPix(i,j). Since this code is at the heart of
```

```
// considering
// images as functions it might be worthwhile to be as efficient as
// possible.
{
   Z m=getLin();
   Z n=getCol();
   x-=floor(x);
   y-=floor(y);
      // now 0<=x<1, 0<=y<1, makes the periodicity with unit period
   x*=n;
   y*=m;
      // 0<=x<n, 0<=y<m,
   Z xd=floor(x);
      // varies from 0 to n-1
   Z yd=floor(y);
      // varies from 0 to m-1
   Z i=m-yd;   // varies from m to 1
   Z j=xd+1;   // varies from 1 to n
   if (method==0){
       return (*this)[i][j];
    //  return getPix(i,j);
   }
   else if (method==1){
      // unfortunately every case seems to need special consideration:
      // see my pages UM/CTL 98-3-6 in the Folder UM 98, CTL, KAGUM
        x-=xd;
      y-=yd;
      R p1,p2,f00,f01,f10,f11;
      Z iNext=i;
      Z jNext=j;
         // these will be changed by reference by dec and inc
         // this will happen only once
      if (x==0.5){
         if (y==0.5) return (*this)[i][j] ;

         if (y>0.5) {
            f00=(*this)[i][j];
            dec(iNext);
            f01=(*this)[iNext][j];
            p2=y-0.5;
         }
         else{ // y<0.5
            inc(iNext,m);
            f00=(*this)[iNext][j];;
            f01=(*this)[i][j];
            p2=y+0.5;
         }
         return f00*(1-p2)+f01*p2;
      }
```

```
if (x>0.5){
   if (y>0.5){  // OK
      dec(iNext);
      inc(jNext,n);
      f00=(*this)[i][j];
      f10=(*this)[i][jNext];
      f01=(*this)[iNext][j];
      f11=(*this)[iNext][jNext];
      p1=x-0.5;
      p2=y-0.5;
   }
   else if (y<0.5){   // OK
      inc(iNext,m);
      inc(jNext,n);
      f00=(*this)[iNext][j];
      f10=(*this)[iNext][jNext];
      f01=(*this)[i][j];
      f11=(*this)[i][jNext];
      p1=x-0.5;
      p2=y+0.5;
   }
   else if (y==0){
      inc(jNext,n);
      f00=(*this)[i][j];
      f10=(*this)[i][jNext];
      p1=x-0.5;
      return f00*(1-p1)+f10*p1;
   }
}
else{ // then x<0.5, since == was carried to return earlier
   if (y>0.5){  // OK
      dec(iNext);
      dec(jNext);
      f00=(*this)[i][jNext];
      f10=(*this)[i][j];
      f01=(*this)[iNext][jNext];
      f11=(*this)[iNext][j];
      p1=x+0.5;
      p2=y-0.5;
   }
   else if (y<0.5){
      inc(iNext,m);
      dec(jNext);
      f00=(*this)[iNext][jNext];
      f10=(*this)[iNext][j];
      f01=(*this)[i][jNext];
      f11=(*this)[i][j];
      p1=x+0.5;
      p2=y+0.5;
   }
```

```
         else if (y==0){
            inc(jNext,n);
            f00=(*this)[i][j];
            f10=(*this)[i][jNext];
            p1=x-0.5;
            return f00*(1-p1)+f10*p1;
         }
      }
      R b=f10-f00;
      R c=f01-f00;
      R d=f11-f00-b-c;
      return f00+b*p1+(c+d*p1)*p2;
         // f(x,y):=f00+b*x+c*y+d*x*y has to obey
         // f(0,0)=f00, f(0,1)=f01, f(1,0)=f10, f(1,1)=f11
         // indexes and arguments refer to the (x,y)-system for which
         // lower left corner is (0,0) and right upper corner is (1,1)
   }
   else{
      cpmerror("Image<T>::operator(): unvalid parameter method");
      return 0;
   }
}

template<class T>
void Image<T>::gamma(R val1, R val2, R valMin, R valMax, R gammaI)
{
   R2 b=minMax();

   R u1,u2;
   if (val1<=val2){
      u1=val1;
      u2=val2;
   }
   else{
      u1=val2;
      u2=val1;
   }

   R v1,v2;
   if (valMin<=valMax){
      v1=valMin;
      v2=valMax;
   }
   else{
      v1=valMax;
      v2=valMin;
   }

   Z i,j;
   Rh pix;
```

```
   Z m=Base::dim1(), n=Base::dim2();
   for (i=1;i<=m;i++) for (j=1;j<=n;j++){
      pix=getPix(i,j);
      pix=response(u1,u2,v1,v2,gammaI,pix);
      (*this)[i][j]=(T)pix;
   }
}

template<class T>
void Image<T>::gamma(R valMin, R valMax, R gammaI)
{
   R2 b=minMax();
   gamma(b.x1,b.x2,valMin,valMax,gammaI);
}

template<class T>
void Image<T>::gamma(R gammaI)
{
   R2 b=minMax();
   gamma(b.x1,b.x2,b.x1,b.x2,gammaI);
}

template<class T>
void Image<T>::sampleComplexFunction(const Graph& gr, C (*f)(const C&),
    R (*g)(const C&, const R_Vector&), const R_Vector& p )
   // The rectangle in object space which belongs to *this is considered
   // a m*n pixel matrix independently of the pixel structure which also
   // is determined by *this. We take samples of the function f (defined
   // on object space) at the positions of pixel centers.
   // The correspondence between position in object space and in the
   // resulting GrayImage is matrix oriented: GrayImage(1,1) corresponds
   // to the upper left corner of object space.
   // The complex values of f at the sampling positions are transformed
   // into real values via the transformation g under control of the
   // parameter vector p ( allows maximum flexibility due to variable
   // number of components.
{
   C z,w;
   R aspRatGraph=cpmabs(gr.getyUpper()-gr.getyLower());
   cpmassert(aspRatGraph!=0.,"");
   aspRatGraph=cpmabs(gr.getxRight()-gr.getxLeft())/aspRatGraph;
   setAspRat(aspRatGraph);
      // aspect ration of image should be that of the object space
      // area of interest. Then an orthogonal line system in the
      // complex plane will be shown as an orthogonal line grid if in
      // Frame::show(GrayImage a, R_Vector par, Z method, Z faithful )
        // the last parameter is set to 1.
   Z i,j,i_;
   R u,v;
   T r;
```

```
   Z m=Base::dim1(), n=Base::dim2();
   V<R> iLattice=ImagingTools::pixelCenters(gr.getyLower(),
      gr.getyUpper(),m);
      // matrix index i varies in y direction
   V<R> jLattice=ImagingTools::pixelCenters(gr.getxLeft(),
      gr.getxRight(),n);
      // matrix index j varies in x direction
   for (i=1;i<=m;i++){
      i_=m-i+1;
      v=iLattice[i_];
      for (j=1;j<=n;j++){
         u=jLattice[j];
         z=C(u,v);
         w=f(z);
         r=g(w,p);  // transformation C to R
         (*this)[i][j]=r;
      }
   }
}

template<class T>
void Image<T>::sampleComplexFunction(R xL, R yL, R dx, R dy,
      const F<C,C>& f, const F<C,R>& g)
{
   const Z nTacit=4;
      // controls frequency of messages in verbose mode

   cpmassert(dx>=0.,"");
   cpmassert(dy>0.,"");

   R xU=xL+dx;
   R yU=yL+dy;

   setAspRat(dx/dy);
      // aspect ration of image should be that of the object space
      // area of interest. Then an orthogonal line system in the
      // complex plane will be shown as an orthogonal line grid if in
      // Frame::show(GrayImage a, R_Vector par, Z method, Z faithful )
      // the last parameter is set to 1.
   Z m=Base::dim1(), n=Base::dim2();

   Z i,j,i_;
   R u,v;
   V<R> iLattice=ImagingTools::pixelCenters(yL,yU,m);
      // matrix index i varies in y direction
   V<R> jLattice=ImagingTools::pixelCenters(xL,xU,n);
      // matrix index j varies in x direction

   C z,w;
   T r;
```

```
   Z lineCounter=0;
   R mInv=1./m;

   for (i=1;i<=m;i++){
      i_=m-i+1;
      v=iLattice[i_];
        if (verbose==1)
      {   lineCounter++;
         if (lineCounter==nTacit){
            lineCounter=0;
            R fr=100.*i*mInv;
            cout<<endl<<"Image "<<fr<<" % completed";
         }
        }

      for (j=1;j<=n;j++){
         u=jLattice[j];
         z=C(u,v);
         w=f(z);
         r=g(w);  // transformation C to R
         (*this)[i][j]=r;
      }
   }
}

template<class T>
void Image<T>::
sampleComplexFunction(R xL, R yL, R dx, R dy, const F<C,R>& f)
{
   const Z nTacit=4;
      // controls frequency of messages in verbose mode

   cpmassert(dx>=0.,"");
   cpmassert(dy>0.,"");

   R xU=xL+dx;
   R yU=yL+dy;

   setAspRat(dx/dy);
      // aspect ration of image should be that of the object space
      // area of interest. Then an orthogonal line system in the
      // complex plane will be shown as an orthogonal line grid if in
      // Frame::show(GrayImage a, R_Vector par, Z method, Z faithful )
      // the last parameter is set to 1.
   Z m=Base::dim1(), n=Base::dim2();

   Z i,j,i_;
   R u,v;
   V<R> iLattice=ImagingTools::pixelCenters(yL,yU,m);
      // matrix index i varies in y direction
```

```
   V<R> jLattice=ImagingTools::pixelCenters(xL,xU,n);
      // matrix index j varies in x direction

   C z,w;
   T r;
   Z lineCounter=0;
   R mInv=1./m;

   for (i=1;i<=m;i++){
      i_=m-i+1;
      v=iLattice[i_];
        if (verbose==1)
      {   lineCounter++;
         if (lineCounter==nTacit){
            lineCounter=0;
            R fr=100.*i*mInv;
            cout<<endl<<"Image "<<fr<<" % completed";
         }
        }

      for (j=1;j<=n;j++){
         u=jLattice[j];
         z=C(u,v);
         r=f(z); // transformation C to R
         (*this)[i][j]=r;
      }
   }
}

template<class T>
void Image<T>::
insertPattern(R xL, R yL, R dx, R dy, const F<C,T>& f, const F<C,B>& g)
{
   const Z nTacit=4;
      // controls frequency of messages in verbose mode

   cpmassert(dx>=0.,"");
   cpmassert(dy>0.,"");

   R xU=xL+dx;
   R yU=yL+dy;

   setAspRat(dx/dy);
      // aspect ration of image should be that of the object space
      // area of interest. Then an orthogonal line system in the
      // complex plane will be shown as an orthogonal line grid if in
      // Frame::show(GrayImage a, R_Vector par, Z method, Z faithful )
      // the last parameter is set to 1.
   Z m=Base::dim1(), n=Base::dim2();
```

```
   Z i,j,i_;
   R u,v;
   V<R> iLattice=ImagingTools::pixelCenters(yL,yU,m);
      // matrix index i varies in y direction
   V<R> jLattice=ImagingTools::pixelCenters(xL,xU,n);
      // matrix index j varies in x direction

   C z,w;
   T r;
   Z lineCounter=0;
   R mInv=1./m;
   B valid;

   for (i=1;i<=m;i++){
      i_=m-i+1;
      v=iLattice[i_];
      if (verbose==1)
      {
         lineCounter++;
         if (lineCounter==nTacit){
            lineCounter=0;
            R fr=100.*i*mInv;
            cout<<endl<<"Image "<<fr<<" % completed";
         }
      }

      for (j=1;j<=n;j++){
         u=jLattice[j];
         z=C(u,v);
         valid=g(z);
         if (valid){   // thus only those parts are
            r=f(z); // transformation C to R
            (*this)[i][j]=r;
         }
      }
   }
}


template<class T>
void Image<T>::mark(Frame& fr, const Response& rsp, Z method,
   Z faithful)const
{
   Z mL=2;
   Word loc("Image<>::show(Frame,R2,Z,Z,Z)");
   CPM_MA
   Z i,j,m=Base::dim1(),n=Base::dim2();
   R pI, qI, width=1., hight=1.;
   if (faithful){ // we assume square pixels on screen
      R alpha_=getAspRat(); // aspect ratio of image
```

```
      R alpha=fr.getAspRat(); // aspect ratio of frame
      if (alpha_ > alpha){ // image has to be placed full width
         width=1.;
         hight=alpha/alpha_;
      }
      else{ // image has to be placed full hight
         width=alpha_/alpha;
         hight=1.;
      }
   }

   V<R> lattice1=ImagingTools::pixelCenters(0.,width,n);
   V<R> lattice2=ImagingTools::pixelCenters(0.,hight,m);

   Z2 index;
   R2 p;
   Z left,right,lower,upper;
   for (i=1;i<=fr.getLin();i++){
      index.x1=i;
      for (j=1;j<=fr.getCol();j++){
         index.x2=j;
         p=fr.getxFyF(index); // point in frame system
         if (p.x2>hight || p.x1>width) continue;
         ImagingTools::find(lattice1, p.x1, pI, left, right);
         ImagingTools::find(lattice2, p.x2, qI, lower, upper);
            // left ... upper refer to discretized xF, yF
         lower=1+m-lower;
         upper=1+m-upper;
            // now left ... upper refer to the matrix index grid
   // pixel values are converted to R for (optional) interploation
   // and input to rsp
         R a11=getPix(upper,left);
         R a12=getPix(upper,right);
         R a21=getPix(lower,left);
         R a22=getPix(lower,right);
         R val=ImagingTools::int2(a11,a12,a21,a22,pI,qI,method);
         Color c=rsp.color(val);
         fr.put(index,c);
      }
   }
 //  fr.display(); // 2005-08-16
   CPM_MZ
}

namespace{

Z multPix(Z n, R x)
{
   Z res=(Z)(x*n+0.5);
   if (res==0) return 1;
```

```
   if (res>n) return n;
   return res;
}

}

template<class T>
R2 Image<T>::minMax(R xL, R xU, R yL, R yU)const
   // Let res be the return value. Then res.x is the minimum
   // and res.x2 the maximum of the pixel values of *this
{
   const R veryLarge=1e30;
   R max=-veryLarge, min=veryLarge, a;
   cpmassert(xL>=0 && xU<=1,"");
   cpmassert(yL>=0 && yU<=1,"");
   cpmassert(xL<xU,"");
   cpmassert(yL<yU,"");
   Z m=getLin();
   Z n=getCol();
   Z j1=multPix(n,xL);
   Z j2=multPix(n,xU);
   Z i1=multPix(m,1.-yU);
   Z i2=multPix(m,1.-yL);
   Z i,j;
   for (i=i1;i<=i2;i++) for (j=j1;j<=j2;j++){
      a=getPix(i,j);
      if (a>max) max=a;
      if (a<min) min=a;
   }
   R2 res(min,max);
   return res;
}

template<class T>
R_Vector Image<T>::statistics(R xL, R xU, R yL, R yU)const
{
   const R veryLarge=1e30;
   R max=-veryLarge, min=veryLarge, a;
   cpmassert(xL>=0 && xU<=1,"");
   cpmassert(yL>=0 && yU<=1,"");
   cpmassert(xL<xU,"");
   cpmassert(yL<yU,"");
   Z m=getLin();
   Z n=getCol();
   Z j1=multPix(n,xL);
   Z j2=multPix(n,xU);
   Z i1=multPix(m,1.-yU);
   Z i2=multPix(m,1.-yL);
   Z i,j,count=0;
   R sum=0;
```

```
   for (i=i1;i<=i2;i++) for (j=j1;j<=j2;j++){
      a=getPix(i,j);
      sum+=(R)a;
      count++;
      if (a>max) max=a;
      if (a<min) min=a;
   }
   cpmassert(count>0,"");
   R mean=sum/count;
   R_Vector res(3);
   res[1]=min;
   res[2]=mean;
   res[3]=max;
   return res;
}

template<class T>
Image<T> Image<T>::cut(R xL, R xU, R yL, R yU)const
{
   if (verbose) cpmmessage("Image<>::cut(R,R,R,R) started");
   cpmassert(xL>=0 && xU<=1,"");
   cpmassert(yL>=0 && yU<=1,"");
   cpmassert(xL<xU,"");
   cpmassert(yL<yU,"");
   Z m=getLin();
   Z n=getCol();
   Z j1=multPix(n,xL);
   Z j2=multPix(n,xU);
   Z i1=multPix(m,1.-yU);
   Z i2=multPix(m,1.-yL);
   Z mc=1+i2-i1;
   Z nc=1+j2-j1;
   Z si=i1-1;
   Z sj=j1-1;

   Image<T> res(mc,nc);
   Z i,j;
   for (i=1;i<=mc;i++){
      for (j=1;j<=nc;j++){
         res.putPix(i,j,getPix(i+si,j+sj));
      }
   }
   if (verbose) cpmmessage("Image<>::cut(R,R,R,R) done");
   return res;
}

template<class T>
Image<T> Image<T>::cut(Z2 cLL, Z2 cUR)const
{
   Z2 cLLVVStyle(cLL[1],cLL[0]);
```

```
   Z2 cURVVStyle(cUR[1],cUR[0]);

                           //                cUR
   Z i1=cURVVStyle[0];     //   (i1,j1)       (i1,j2)
                           //                              matrix indexing
                           //   (i2,j1)       (i2,j2)
   Z i2=cLLVVStyle[0];     //  cLL

   Z j1=cLLVVStyle[1];
   Z j2=cURVVStyle[1];

   CpmRootX::order(i1,i2);
   CpmRootX::order(j1,j2);
   cpmassert(i1<i2 && j1<j2,""); // for utmost security
   Z m=getLin();
   Z n=getCol();
   cpmassert(i2<=m && j2<=n,"");
      // to catch inconsitent input of cLL, cUR

   Z mc=1+i2-i1;
   Z nc=1+j2-j1;
   Z si=i1-1;
   Z sj=j1-1;
   if (verbose){
      Word mess="Image<>::cut(Z2, Z2) creating";
      Word mess1=" m="+CpmRoot::toWord(mc)+" n="+CpmRoot::toWord(nc);
      mess=mess+mess1;
      cpmmessage(mess);
   }
   Image<T> res(mc,nc);
   Z i,j;
   for (i=1;i<=mc;i++){
      for (j=1;j<=nc;j++){
         res.putPix(i,j,getPix(i+si,j+sj));
            // notice mc+si=i2, nc+sj=j2, thus i2<=m and j2<=n
            // is a guarantee that the index is never out of range
      }
   }
   return res;
}

template<class T>
Image<T> Image<T>::cut(Z i1, Z i2, Z j1, Z j2)const
{
   CpmRootX::order(i1,i2);
   CpmRootX::order(j1,j2);
   Z m=i2-i1+1;
   Z n=j2-j1+1;
   Z i,j;
   Image<T> res(m,n);
```

```
   for (i=1;i<m;i++){
      for (j=1;j<=n;j++){
         T val=getPix(i+i1-1,j+j1-1);
         res.putPix(i,j,val);
      }
   }
   return res;
}

template<class T>
Image<T> Image<T>::clip(R clipL, R clipU )const
{
   R vMax=minMax().get2();
   T mi=clipL*vMax;
   T ma=clipU*vMax;
   if (verbose) cout<<endl<<"lower clip level = "<<mi;
   if (verbose) cout<<endl<<"upper clip level = "<<ma;
   Image<T> res=*this;
   Z m=getLin();
   Z n=getCol();
   Z i,j;
   T val;
   for (i=1;i<=m;i++){
      for (j=1;j<=n;j++){
         val=getPix(i,j);
         if (val<mi) res.putPix(i,j,mi);
         if (val>ma) res.putPix(i,j,ma);
      }
   }
   return res;
}

template<class T>
Image<T> Image<T>::transformToneScale(const F<T,T>& f)const
      // returns a new image that takes pixel value f(val)
      // where the original image had pixel value val.
{
   Z m=getLin();
   Z n=getCol();
   Z i,j;
   T val;
   Image<T> res(m,n);
   for (i=1;i<=m;i++){
      for (j=1;j<=n;j++){
         val=(*this)[i][j];
         res[i][j]=f(val);
      }
   }
   return res;
}
```

```
template<class T>
T Image<T>::getLocalMean(Z i, Z j, const Mask& a)const
{
   T res=T();
   Z p,d=a.dim();
   for (p=1;p<=d;p++) res+=(getPix(i+a.di(p),j+a.dj(p))*a.w(p));
      // no out of range problem due to getPix's robustness
      // assumes  T+=T and T*R, satisfied e.g. for T=C
   return res;
}

template<class T>
Image<T> Image<T>::conv(const Mask& a)const
{
   cpmmessage( "Image<>::conv(const Mask& ) started");
   if (a.isVoid()) {
      cpmmessage("Image<>::conv(const Mask& ) trivially done");
      return *this;
   }
   Z m=getLin(), n=getCol();
   Z i, j;
   Image<T> res(m,n);
   R prfrac=1./m;
   for (i=1;i<=m;i++){
      cpmprogress("Image<T>::conv()-loop",i*prfrac);
      for (j=1;j<=n;j++){
         res[i][j]=getLocalMean(i,j,a);
      }
   }
   cpmmessage( "Image<>::conv(const Mask& ) done");
   return res;
}


template<class T>
Image<T> Image<T>::conv(const R_Matrix& a, Z si, Z sj)
{
   cpmmessage( "Image<>::conv(const R_Matrix& , Z , Z) started");

   Z ma=a.dim1(),na=a.dim2(), ia, ja;
   if (ma==0 && na==0){
      cpmmessage( "Image<>::conv(const R_Matrix&,Z,Z) trivially done");
      return *this;
   }
   Z m=getLin(), n=getCol(), i, j;
   Z ci=1+ma/2, cj=1+na/2;
   Z i0=ci+si, j0=cj+sj;
   T val=T();
   Image<T> res(m,n);
```

```
     R prfrac=1./m;
   for (i=1;i<=m;i++){
      cpmprogress("Image<T>::conv()-loop",i*prfrac);
      for (j=1;j<=n;j++){
         val=0.;
         for (ia=1;ia<=ma;ia++){
            for (ja=1;ja<=na;ja++){
               val+=getPix(i+ia-i0,j+ja-j0)*a[ia][ja];
                  // if ma and na are odd and si=sj=0, the range should be
                  // i-ma/2, i+ma/2
                  // j-na/2, j+ma/2
                  // Thus i+1+ti=i-ma/2, thus ti=-ma/2-1
                  // Correspondingly tj=-na/2-1. For even ma or na this
                  // is taken over as definition. Formulae 1 and 2 are
                  // definitions of si,sj
            }
         }
         res.putPix(i,j,val);
      }
   }
   cpmmessage( "Image<>::conv(const R_Matrix& , Z , Z) done");
   return res;
}

template<class T>
Image<T>& Image<T>::operator *= (R r)
{
   Z m=getLin(), n=getCol(), i, j;
   for (i=1;i<=m;i++){
      for (j=1;j<=n;j++){
         putPix(i,j,getPix(i,j)*r);
      }
   }
   return *this;
}

template<class T>
Image<T>& Image<T>::operator += (R r)
{
   Z m=getLin(), n=getCol(), i, j;
   for (i=1;i<=m;i++){
      for (j=1;j<=n;j++){
         putPix(i,j,getPix(i,j)+r);
      }
   }
   return *this;
}

template<class T>
void Image<T>::frame(const T& t)
```

```
{
   Z m=getLin(), n=getCol(), i, j;
   i=1;
   for (j=1;j<=n;j++){
      putPix(i,j,t);
   }
   i=m;
   for (j=1;j<=n;j++){
      putPix(i,j,t);
   }
   j=1;
   for (i=1;i<=m;i++){
      putPix(i,j,t);
   }
   j=n;
   for (i=1;i<=m;i++){
      putPix(i,j,t);
   }
}

template<class T>
Image<T> Image<T>::fftFilter(const F<C,C>& resp)const
{
   using namespace CpmFourier;
   Z i,j,m=Base::dim1(),n=Base::dim2();
   VV<C> mat(m,n);
   for (i=1;i<=m;++i) for (j=1;j<=n;++j){
      R vij=(*this)[i][j];
      vij=cpmabs(vij);
         // image should be positive so cpmabs is only
         // a defensive provision
      mat[i][j]=C(::sqrt(vij));
         // amplitude is the square-root of the intensity
   }
   SD2 x(mat,true);
      // considering this as distribution in space domain
   Z mx=x.dim1();
   Z nx=x.dim2();
   R mI=1./(mx-1);
   R nI=1./(nx-1);
   FD2 y=x.fft();
      // going to frequency domain
   for (i=1;i<=mx;++i){
      R yc=-0.5+(i-1.)*mI;
         // yc from -1/2 to 1/2 for i 1 to mx
      for (j=1;j<=nx;++j){
         R xc=-0.5+(j-1.)*nI;
            // xc from -1/2 to 1/2 for i 1 to nx
         C arg(xc,yc);
         C fac=resp(arg);
```

```
            // multiplicative modification
         y(i,j)*=fac;
      }
   }
   SD2 xt=y.fft();
      // back to space domain amplitude
   Image<T> res(mx,nx);
   for (i=1;i<=mx;++i) for (j=1;j<=nx;++j){
      R rij=xt(i,j).absSqr();
      res[i][j]=rij;
         // back to intensity as square of abs
         // of the amplitude
   }
   return res;
}

template<class T>
Image<T> Image<T>::fourierFilter(const Fa<R2,R>& resp)const
{
   Z mL=1;
   static Word loc("Image<>::fourierFilter(Fa<R2,R>)");
   CPM_MA
   VV<C> a=fourierTransform();
// filtering the Fourier image
   Z m2=a.dim1(),n2=a.dim2();
   R fac,xi,xj;
   Z i,j,ic,jc;
   Z mNy=m2/2+1;
   Z nNy=n2/2+1;
   R ci=2./m2, cj=2./n2;
   for (i=1;i<=m2;i++){
      ic=(i<=mNy ? i : 2+m2-i);
         // this ensures that the components belonging to frequency
         // f will be multiplied by the same factor as those multiplied
         // by -f
      xi=(ic-1)*ci;
         // xi varies in [0,1]
      for (j=1;j<=n2;j++){
         jc=(j<=nNy ? j : 2+n2-j);  // as for i
         xj=(jc-1)*cj;
         fac=resp(R2(xi,xj));
         a(i,j)*=fac;
      }
   }
   if (verbose>11){
      // shows the spectrum (absolut value) of the filtered image
      Frame f;
      Image<Rh> aShow(m2,n2);
      R rv;
      R tiny=1e-5;
```

```
      R sc=absVal(a(1,1));   // normally the largest value
      cpmassert(sc>0.,"");
      sc=1./sc;
      for (i=1;i<=m2;i++) for (j=1;j<=n2;j++){
         rv=log( tiny+sc*absVal(a(i,j)) );
         aShow.putPix(i,j,rv);
      }
      aShow.mark(f);
      f.display();
      cpmwait(4);
   }
   Image<T> res(a);
   CPM_MZ
   return res;
}

template<class T>
VV<C> Image<T>::fourierTransform()const
{
   Z mL=1;
   static Word loc("Image<>::fourierTransform()");
   CPM_MA
   Z m=getLin(), n=getCol(), i, j;
   cpmassert(m>=2 && n>=2,loc);
   Z m2=2; while (m2<m) m2*=2;
   Z n2=2; while (n2<n) n2*=2;
   C zero;
   VV<C> ori(m2,n2,zero); // sufficiently large for fourier purposes
      // and initialized by zero
   for (i=1;i<=m;i++) for (j=1;j<=n;j++){
      R vij=getPix(i,j);
      ori(i,j)=C(vij);
   }
   CPM_MZ
   return fft(ori,1);
}

template<class T>
Image<T> Image<T>::cut(const Rect& rect)const
{
   R xL=rect.getxL();
   R xU=rect.getxU();
   R yL=rect.getyL();
   R yU=rect.getyU();
   Z m=getLin();
   Z n=getCol();
   Roi r(m,n,xL,xU,yL,yU);
   Image<T> res(r.m(),r.n());
   Z i,j;
   for (i=1;i<=r.m();i++){
```

```
      for (j=1;j<=r.n();j++){
         res[i][j]=getPix(r.i(i),r.j(j));
      }
   }
   return res;
}

template<class T>
R Image<T>::norm1(void)const
   // sum over the absolute values over all pixel values, divided by
   // number of pixels
{
   Z m=getLin();
   Z n=getCol();
   Z i,j;
   R sum=0.;   // even if T is a integer type !
   R val;
   if (verbose){ cout<<endl<<"MultImage<>::norm1 started";}
   for (i=1;i<=m;i++){
      if (verbose>9){ cout<<endl<<"treating line "<<i;}
      for (j=1;j<=n;j++){
         val=getPix(i,j);
         R valr=CpmRoot::absVal<T>(val);
         sum+=valr;
      }
   }
   R nPix=1.*m*n;
   cpmassert(nPix>0,"");
   sum/=nPix;
   return sum;
}

template<class T>
Histogram Image<T>::histogram(Z nBins, R BDown, R BUp )const

{
   Z m=getLin();
   Z n=getCol();
   Z i,j;
   R val;
   Histogram b(1,nBins,BDown,BUp);
   for(i=1;i<=m;i++){
      for(j=1;j<=n;j++){
         val=getPix(i,j);
         b.put(1,val);
      }
   }
   return b;
}
```

```
template<class T>
Img24 Image<T>::toImg24()const
{
   Z wR=w();
   Z hR=h();
   Img24 res(wR,hR);
   Z i,j;
   for (i=0;i<wR;++i){
      for (j=0;j<hR;++j){
         L val=(L)(*this)[j+1][i+1];
         res.putPel(i,j,rgb(val,val,val));
      }
   }
   return res;
}

} // namespace

#endif
```

96 cpmimagingtools.h

```
//? cpmimagingtools.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_IMAGINGTOOLS_H_
#define CPM_IMAGINGTOOLS_H_
/*
   Description: Defines some functions to be used in image classes
*/
#include <cpmva.h>
#include <cpmf.h>
#include <cpmc.h>
#include <cpminterval.h>
#include <cpmangle.h>
#include <cpmviewport.h>
#include <cpmalgorithms.h>

namespace CpmImaging{

   using CpmRoot::Z;
   using CpmRoot::Word;
   using CpmRoot::R;
   using CpmRoot::C;

   using CpmRootX::Z1;
   using CpmRootX::R1;
   using CpmRootX::B;

   using CpmLinAlg::R3;

   using CpmArrays::V;
   using CpmArrays::Va;

   using CpmGraphics::rgb;
   using CpmGraphics::ColRef;

   using CpmGeo::Iv;
   using CpmGeo::Angle;

   using CpmAlgorithms::CyclicAlarm;
   using CpmAlgorithms::NonCyclicAlarm;

//////////////////// class Color ////////////////////////////////////////

   enum Palette {
    BLACK,            // dark colors
    BLUE,
    GREEN,
```

```
    CYAN,
    RED,
    MAGENTA,
    BROWN,
    FAINTGRAY,
    LIGHTGRAY,
    DARKGRAY,           // light colors
    LIGHTBLUE,
    LIGHTGREEN,
    LIGHTCYAN,
    LIGHTRED,
    LIGHTMAGENTA,
    YELLOW,
    ORANGE,
    WHITE,
    DARKBLUE,    // dark colors
    DARKGREEN,
    DARKCYAN,
    DARKRED,
    DARKMAGENTA,
    INVISIBLE,   // special colors
      // If the color of a body is set 'invisible' this does not mean
      // that its screen image can not be seen. Most of my graphics
      // functions Camera::mark(...) then make an anaglyphic stereo
      // representation in 3D and the corresponding flat line
      // representation in 2D. In a sense, then the color of the body is
      // in fact invisible; the body is 'mere form'
    STEREOBLACK,
    STEREORED,
    STEREOGREEN,
    TRANSPARENT_ ,// without _ it clashes with a MS-name
    STEREO,
      // same as INVISIBLE
    WRITECOLOR,
    GRIDCOLOR
};

   class Color: public rgb{//data of CpmGraphics::rgb + gamma, aritmetics
      static R gamma;
         // comes into action only in converting C to Color

      void make(Palette c);
         // sets r,g,b according to a Palette entry
         // (Palette is an enum !)
         // actually this function defines what is the operational
         // meaning of a Palette entry

      static R valMax_;
   public:
      typedef Color Type;
```

```
typedef R ScalarType;
   // for arithmetics defined via CPM_PRODUCT_SCALAR_C

virtual Color* clone()const{ return new Color(*this);}
   // allows Color to be inserted as X in the polymorphic array
   // template Vp<X>
CPM_IO_V
CPM_ORDER
CPM_PRODUCT_SCALAR_C
CPM_SUM_C
Word nameOf()const{ return Word("Color");}

explicit Color(rgb const& c):rgb(c){}
Color():rgb(){}
Color(Z r_, Z g_, Z b_):rgb(r_,g_,b_){}
explicit Color(V<Z> const& v):rgb(v(1),v(2),v(3)){}
   // This is usefull since V<Z> is among the data types that can be
   // exchanged with class Record on which my ini-files are based.
Color(R r_, R g_, R b_):rgb((Z)r_,(Z)g_,(Z)b_){}
Color(Palette c){ make(c);}
   // so a Palette-value will directly be converted into a Color
   // make(c) does not call a normalizing constructor of rgb,
   // so it may set negative values as done for c=INVISIBLE

Color(Word w, R br):rgb((Z)br,(Z)br,(Z)br){}
   // first argument 'dummy' second gives the brightness
   // of gray

Color(R3 const& y);
   // construction from real triples
   // Usefull values satisfy 0<=y[i]<=1, i=1,2,3.
   // Other values get squeezed to 0 or 255.
   // This constructor is usefull in order to construct color
   // representations of arithmetical constructs.

R3 toR3()const;
   //: to R3
   // For each Color c, we have Color(c.toR3())==c.
   // The converse property can not hold: For arbitrary
   // y in R3 such that 0<=y[i]<=1, i=1,2,3 we are not sure
   // that Color(y).toR3() == y since the transition from
   // y to Color(y) waists bits.

explicit Color(Z colNumber);
   // so colors can be initialized also by numbers,
   // numbers <=0 create Color(INVISIBLE). Large numbers repeat
   // periodically the basic Palette (the colors from BLACK to
   // WHITE). Thus:
   // Color(0)=INVISIBLE,
```

```
   // Color(1)=BLACK, Color(2)=BLUE, Color(3)=GREEN,
   // Color(4)=CYAN, Color(5)=RED
   // Color(16)=ORANGE,
   // Color(17)=WHITE, Color(18)=BLACK,...
   // No automatic conversion.
   // Notice that there is another way of generating colors from
   // integers:
   // i|--> CpmImaging::rainBow.cyc(i);
   // For these repeated 8 colors the intended order is evident,
   // so it especially useful for displaying families of curves

static Z toZ(Palette p);
   // Defined such that for Palette p we have
   // Color(p)==Color(Color::toZ(p)) (for all p
   // for which there is Z i such that Color(i)==Color(p)
   // Thus INVISIBLE, BLACK, BLUE, ... WHITE

static Palette toPal(Z i);
   // Returns the Palette instance that belongs to Color(i)

explicit Color(const C& z, Iv minMax=Iv(0,1));
   // coding complex numbers as colors, the absolute value controls
   // luminance and the phase angle controls the hue. We use
   // maximally saturated colors only and the angle is 0 for red,
   // 120 for green, 240 for blue. Thus the order is in the
   // direction of increasing polar angle (0->360):
   // red,yellow,green,cyan,blue,purple,red
   // Notice: red   == real positive
   //         cyan  == real negative
   //         green == imaginary positive
   //         blue  == imaginary negative
   // other algorithms that I tested made negative real and
   // negative imaginary appear rather similar.
   // The absolute value of z gets transformed into brightness
   // via ImagingTools::response according to:
   // response(minMax.inf(), minMax.sup(), 0, 255, gamma,
   // z.absVal());
   // For gamma=0 one shows only the phase of z coded as color
   // and the absolute value of z does not matter. This is
   // sometimes usefull.

explicit Color(R r, Iv minMax=Iv(0,1));
   // coding real numbers as a color such that two pure colors get
   // connected over the white point.
   // Presently: (0,1/2,1)|--> red,white,blue
   // This just fits my EP-application

explicit Color(ColRef cr):rgb(cr){}

Z Red()const{return (r<0 ? 0 : r);}
```

```
      Z Green()const{return (g<0 ? 0 : g);}
      Z Blue()const{return (b<0 ? 0 : b);}

      R red()const{return (r<0 ? 0. : (R)r);}
      R green()const{return (g<0 ? 0. : (R)g);}
      R blue()const{return (b<0 ? 0. : (R)b);}

      bool visible()const{ return r>=0;}
         // notice that for Color(INVISIBLE) we have r=g=b=-1
      bool transparent()const{ return r==-137;}
      bool stereo()const{return r<0;}
      ColRef toColRef()const{ return ColRef(r,g,b);}
      static void setGamma(R gamma_){ gamma=gamma_;}
      static R getGamma(){ return gamma;}
      rgb toRGB()const{ return rgb(r,g,b);}
   };

   const CpmArrays::V<Color> rainBow("",BLACK,MAGENTA,BLUE,CYAN,
      GREEN,YELLOW,ORANGE,RED);

// some dark colors which look good for graphics made of white or yellow
// dots or lines
   const CpmImaging::Color darkSky(0,0,50);
      //: dark sky
   const CpmImaging::Color twiSky(30,0,50);
      //: twilight sky
   const CpmImaging::Color morSky(50,60,100);
      //: morning sky


//////////////////// class ImagingTools ////////////////////////////////

class ImagingTools{ // collection of helper functions

public:

// functions

static R int2(R f11, R f12, R f21, R f22, R p, R q, Z method);
// int2 stands for interpolation in 2 dimensions.
// The returned value is a weighted meen of the fij. Notice
// p+p_=1, q+q_=1, which leads to p*q_+p_*q_+p*q+p_*q=1.
// For interpretation: p and q follow x and y axis.
// The corners of the rectangle in which the interpolation point is
// placed and to which the reduced coordinates p and q refer are as
// follows:
//
//      f11      f12
//
//      f21      f22
```

```
//
// Method =1 means bilinear interpolation and method=0 means
// pixelation. The latter is useful if the number of pixels is to be
// compared to expectations. The bilinear formula is
// f11*p_*q+f12*p*q+f21*p_*q_+f22*p*q_ which implies for
// p=q=0 obviously f21 ( namely the coefficient of p_*q_ ). Thus
// the indexes of f are matrix indexes and p and q refer to normally
// oriented x,y-axes (y from bottom to top )

static CpmArrays::V<R> pixelCenters(R a, R b, Z n);
// The interval [a,b] is divided into n subintervalls of equal length
// and the n centers of these subintervals are returned as a R_Vector
// in ascending order

static void find(const CpmArrays::V<R>& lattice, R x, R& p, Z& i, Z& j);
// After function call, i is the largest integer for which lattice[i]
// is defined and for which lattice[i]<=x.
// p=(x-lattice[i])/(lattice[i+1]-lattice[i]).
// It is assumed that lattice[j+1]-lattice[j]>0 and independent of j,
// whenever none of the indexes is out of range.
// Extension 95-3-3: for values outside the lattice points, linear extra-
// polation may be useful.

static R response(R x1, R x2, R y1, R y2, R gamma, R x);
// Auxiliar function for e.g. Image<T>::gamma(...).
// The function x |---> f(x):=response(...,x) has the following
// properties:
// f(x)=y1 for x<=x1
// f(x)=y2 for x>=x2
// For gamma>=0: f(x)= y1+((x-x1)/(x2-x1))^gamma
// For gamma<0: f(x)=alpha+beta*log(x) with alpha and beta
// determined such that f(x1)=y1, f(x2)=y2
// Also x1 is forced to be positive and x2 is reduced if too large
// for gamma<0.

static R random(Z i);
// returns randomly distributed numbers in the interval [0,1]
// as a function of i=1,2,3,....

static Z random(Z i, Z k);
// returns a randomly distributed numbers in {1,2,...k}
// as a function of i=1,2,3,....

// data
static Z verbose;
// verbose=0, no information outputted during calculations

};

//////////////////// class Response ////////////////////////////////////
```

```
class Response{ // versatile conversion from R to screen color
   // handles the conversion from R into the input range
   // of screen color.
   // The idea is that each color channel may have its own
   // response, so the present class does not deal with cooperative
   // functions of color channels.
   Iv rx,ry;
   R1 gamma;
   Z1 mode;
   static Z1 levels;
      // number of intervals into which the whole
      // color-spectrum in function color() is divided
      // by 'Fraunhofer lines'
   typedef Response Type;
public:
// infrastructure for enabling I/O
// and polymorphic arrays of derived types
   CPM_IO_V
   CPM_NAM(Response)
   virtual Response* clone()const{ return new Response(*this);}
// constructors
   Response(){}
   Response(Iv rxI, Iv ryI=Iv(0,255), R gammaI=1, Z modeI=2):
      rx(rxI),ry(ryI),gamma(gammaI),mode(modeI){}
   Response(R xL, R xU):
      rx(Iv(xL,xU)),ry(Iv(0,255)),gamma(1),mode(Z(0)){}
// raison d'etre
   R operator()(const R& x)const;
      // Assume Response r, and  R x, then r(x) is the response of r
      // to x.
   Color color(R x)const;
      // for mode<=5 this returs Color(v,v,v) if v is the return value
      // of operator(), thus it returns a B/W color (unbunte Farbe).
      // For mode==6 it gives a 'spectral color' under the control of
      // rx and gamma. Colors correspond to x just as x would be a
      // photon energy lowest values correspond to red (also red numbers
      // in economy correspond to lowest, here negative, values of
      // earnings) highest values to violet. The whole range rx gets
      // subdivided into 10 subintervals by darker lines like the
      // Fraunhofer lines in the spectrum of solar light. Notice that
      // the spectral lines have sharp, pixeled, edge towards the
      // boundaries of rx, wheras the interior lines have diffuse edges.

// modifying functions
   void setRangeX(R xL, R xU){ rx=Iv(xL,xU);}
   void setRangeY(R xL, R xU){ ry=Iv(xL,xU);}
   void setGamma(R gammaI){ gamma=gammaI;mode=0;}
   R getGamma()const{ return gamma; }
   bool normalizeGamma(){ bool b=gamma>0; if (!b) gamma*=-1; return b;}
```

```
   void setMode(Z modeI){ mode=modeI;}
      Z getMode()const{ return mode;}
      // mode 0: response contolled by a traditional gamma
      // mode 1: response curve is arctan, gamma has no influence
      // mode 2: periodic response, gamma has no influence
      // mode 3: response curve is sin, gamma has no influence
      // mode 4: band pass response: for values outside rx we return 0
      // mode 5: asymmetric sine response, in between of 2 and 3
      // less aliasing than 2, but visibility of growth and decline as
      // for 2
      // mode 6: colormode; here the the new member-function color
      //    gives meaningful values, see function color() for the meaning
      //    of the colors. catchword: photon energy

   void setGammaResponse(){mode=0;}
   void setArcTanResponse(){mode=1;}
   void setPeriodicResponse(){ mode=2;}
   void setSineResponse(){ mode=3;}
   void setBandResponse(){ mode=4;}
   void setAsySineResponse(){ mode=5;}
   void setColorResponse(){ mode=6;}
   void setLevels(Z lev){ levels=lev;}

   bool getColorResponse(){ return mode>=6;}
   Iv getRangeX()const{ return rx;}
   Iv getRangeY()const{ return ry;}
};

//////////////////////////// class Roi ////////////////////////////////
// 97-5-18(?)

class Roi{  // region of interest

   // The following is the logic to copy the Roi-part of image *this into
   // a new image res

/*
      Z m=getLin();
      Z n=getCol();
      Roi r(m,n,xL,xU,yL,yU);
      Image<T> res(r.m(),r.n());
      Z i,j;
      for (i=1;i<=r.m();i++){
         for (j=1;j<=r.n();j++){
            res.putPix(i,j,getPix(r.i(i),r.j(j)));
         }
      }
*/
protected:
   Z mc,nc,si,sj;
```

```
public:
   Roi(Z m, Z n, R xL, R xU, R yL, R yU);
      // The part of the image (m=image.getCol(), n=image.getLin() )
      // which is described by the Roi has normal
      // coordinates (x,y) with respect to the original which satisfy
      //   xL <= x <= xU
      //   yL <= y <= yU
      // Normal coordinate system: origin lower-left, x-axis
      // to the right, y-axis vertically ascending. The upper-right
      // image corner has coordinates x=1, y=1.
   Roi(Z m, Z n, Iv ivx, Iv ivy);
      // same as previous constructor but with more convenient
      // arguments (ivx[1]=xL, ivx[2]=xU, ivy[1]=yL, ivy[2]=yU)
   Z m(void)const{ return mc;}
   Z n(void)const{ return nc;}
   Z i(Z i0)const{ return i0+si;}
   Z j(Z j0)constc{ return j0+sj;}
};

//////////////////////////////// class Rect ////////////////////////////////

class Rect{  // rectangular sub-area of the unit-square

protected:
   R xL,xU,yL,yU;
      // The points (x,y) of the rectangle satisfy
      // xL<=x<=xU, yL<=y<=yU
   typedef Rect Type;
public:
   CPM_IO
   CPM_NAM(Rect)
   Rect(R a=0, R b=1, R c=0, R d=1);
      // defines a sub-rectangle of the unit-square [0,1]x[0,1].
      // If the arguments are as intended, we have
      // 0<=a<b<=1
      // 0<=c<d<=1.
      // Then the sub-rectangle is simply [a,b]x[c,d]
      // If they are out of range or out of order, they will be brought
      // back in an obvious manner

   Rect(Iv iv1, Iv iv2);
      // 'modern form' defines a rectangle in terms of
      // intervals on the x-axis and the y-axis.
      // the section of iv1 with [0,1] defines [xL,xU]
      // and the section of iv2 with [0,1] defines [yL,yU]

   Rect(Z c1x, Z c1y, Z c2x, Z c2y, Z cx, Z cy);
      // defines the sub-rectangle of the unit-square [0,1]x[0,1]
      // which corresponds to graphics pixel locations for subarea
      // corners as can be read from PhotoStyler screen. Marking the
```

```
      // lower right corner of the whole image  with the pipette tool of
      // PhotoStyler (same with similar programs) we read the pixel
      // coordinates (cx,cy). Marking two corners c1 and c2 which span
      // the subarea of interest (i.e. to be described by the Rect under
      // construction) we get the pixel readings
      // (c1x, c1y)
      // (c2x, c2y)

   R getxL(void)const{ return xL;}
   R getxU(void)const{ return xU;}
   R getyL(void)const{ return yL;}
   R getyU(void)const{ return yU;}

   Rect sub(const Rect& rec)const;
      // returns the rectangle which is formed by taking rec of *this
      // in a way that *this is now considerd a unit square
};

/***** class Mask for defining local mean values of pixels**************/
//97-9-12
// 2008-01-30 Convolutions of images may also be defined in a way that
// the process is given by a R_Matrix which is more direct than with
// Mask

class Mask{ // tool for defining convolution
   // tool for defining linear combinations of pixelvalues in the
   // neighborhood of a given pixel.
   // 'The data are the interface' in this case.
   // The constructor assures reasonable initialisation.
   // Since members are constant, no inconsitencies can be created.
   // Default constructor gives a mask a0 such that
   // ImageUM<R>::getLocalMean(i,j,a0) =  ImageUM<R>::getPix(i,j)
   // 97-9-15: now members not constant, since with constant members
   // no operator = is defined automatically. Returned to the
   // orthodox standpoint where access to data should be define by proper
   // memberfunctions and not by public members. This actually improves
   // the readability of formulas for ImageUM<>::getLocalMean(). This
   // functions completely explains the quantities entering the
   // definition of the present class.
public:
// constructors
   Mask(V<Z> const& diI, V<Z> const& djI, V<R> const& wI):
      di_(diI),dj_(djI),w_(wI)
   { cpmassert(diI.dim()==djI.dim()&&djI.dim()==wI.dim(),"Mask(...)");}

   Mask():di_(1,0),dj_(1,0),w_(1,1){}

   explicit Mask(Z n);
      // a uniform (constant) n by n mask. n is internally chosen as odd
      // if n was even (next higher odd value).
```

```
   Mask(R rMax, const CpmFunctions::F<R,R>& f);
      // a radially symmetric mask of total weight 1, where the
      // individual weights are proportion to f(r), where r is
      // the distance from the center. No contributions for r > rMax,
      // Distance from between pixel centers of horizontally or
      // vertically adjacent pixels is 1. (and sqrt(2) for diagonally
      // adjacent ones)
      // implementation did not work till 03-03-25

static Mask parabolic(R rMax);
   // static function as a 'named constructor'. returns the mask
   // Mask(rMax,f) where f(r):=1-(r/rMax)^2. For rMax=2 the
   // functionality looks very good

static Mask bin3();
   //: binomial 3 (by 3)
   // The smallest from the family of binomial masks which represented
   // as a matrix is
   // ((1,2,1),(2,4,2),(1,2,1))/16

// access functions
   Z dim(void)const{return di_.dim();}
   Z di(Z i)const{ return di_[i];}
   Z dj(Z i)const{ return dj_[i];}
   R w(Z i)const{ return w_[i];}
   Mask invert(void)const;
      // returns a mask which is 'reflected at the origin'
   R rms(void)const;
      // returns the root of the mean (with respect to w) of r square.
      // We do not assume that the components of w_ are positive and/or
      // normalized in some way. We take  always the absolute value of
      // these components and normalize with respect to these positive
      // quantities
   bool isVoid()const{ return (di_.dim()==0 && dj_.dim()==0);}

protected:
   CpmArrays::Va<Z> di_;
   CpmArrays::Va<Z> dj_;
   CpmArrays::Va<R> w_;
};

//////////////////////// struct ShowControl ////////////////////

struct ShowControl{ // collection of display control parameters
   // combining the 'historically' grown screen
   // display control parameters into a single data structure
   mutable CyclicAlarm cyalClear;
   mutable CyclicAlarm cyalView;
   mutable CyclicAlarm cyalImg;
```

```
   mutable NonCyclicAlarm nocyalImg;
   mutable Z1 countImg;
   Z1 offSet;
   Z1 maxNumberImages;
   R1 tWait;
   R1 autoFrameFactor;
   B boxSel;
   Iv ivx,ivy,ivz;
   V<Z> listColorize;
   V<Z> invisibleParticles;
      // particles wit an id in this list will not be marked on screen
      // (introduced 2006-01-02 for creating graphics for printing)
   Word modality;
      // More flexible control than using several B's.
   Word imgFile;
   ShowControl(){}
   Word nameOf()const{ return Word("ShowControl");}
};

} // namespace

#endif
```

97 cpmimagingtools.cpp

```
//? cpmimagingtools.cpp
//? Status of work 2008-10-25.
//?
#include <cpmangle.h>
#include <cpmimagingtools.h>
#include <cpmzinterval.h>

using namespace CpmRoot;
using namespace CpmGeo;
using namespace CpmSystem;
using namespace CpmArrays;
using namespace CpmFunctions;
using namespace CpmImaging;

//////////////////////////// Color ////////////////////////////////

R Color::gamma=1.;
// should be 1

R Color::valMax_=256-1e-6;

bool Color::prnOn(ostream& str)const
{
   using CpmRoot::write;
   CpmRoot::writeTitle("Color",str);
   cpmp(r);
   cpmp(g);
   cpmp(b);
   return true;
}

bool Color::scanFrom(istream& str)
{
   cpms(r);
   cpms(g);
   cpms(b);
   return true;
}

Z Color::com(const Color& c)const
{
   if (r<c.r) return 1;
   if (r>c.r) return -1;
   if (g<c.g) return 1;
   if (g>c.g) return -1;
   if (b<c.b) return 1;
   if (b>c.b) return -1;
```

```
   return 0;
}

Color Color::operator+(const Color& c)const
{
   return Color(r+c.r,g+c.g,b+c.b);
}

Color Color::operator*(const R& x)const
{
   return Color(r*x,g*x,b*x);
}

void Color::make(Palette c)
   {
// The Palette colors make use only of a few code values
      static const Z b4=255;
      static const Z b3=191;
      static const Z b2=127;
      static const Z b1=66;
      static const Z b0=0;
      static const Z bright=200;
      static const Z dark=50;
      switch(c){
// dark colors
      case BLACK:       r=b0; g=b0; b=b0; break; // experiment
      case BLUE:        r=b0; g=b0; b=b4; break;
      case GREEN:       r=b0; g=b4; b=b0; break;
      case CYAN:        r=b0; g=b2; b=b2; break;
      case RED:         r=b4; g=b0; b=b0; break;
      case MAGENTA:     r=b2; g=b0; b=b2; break;
      case BROWN:       r=b2; g=b2; b=b0; break;
      case FAINTGRAY:    r=b3; g=b3; b=b3; break;
         // closer to white than LIGHTGRAY
      case LIGHTGRAY:   r=b2; g=b2; b=b2; break;
         // = 'dark white'
// darker colors
      case DARKBLUE:    r=b0; g=b0; b=b2; break;
      case DARKGREEN:   r=b0; g=b2; b=b0; break;
      case DARKCYAN:    r=b0; g=b1; b=b1; break;
      case DARKRED:     r=b2; g=b0; b=b0; break;
      case DARKMAGENTA: r=b1; g=b0; b=b1; break;
      case DARKGRAY:    r=b1; g=b1; b=b1; break; // = 'light black'
// bright colors
      case LIGHTBLUE:   r=b2; g=b2; b=b4; break;
      case LIGHTGREEN:  r=b2; g=b4; b=b2; break;
      case LIGHTCYAN:   r=b2; g=b4; b=b4; break;
      case LIGHTRED:    r=b4; g=b2; b=b2; break;
      case LIGHTMAGENTA:r=b4; g=b2; b=b4; break;
      case YELLOW:      r=b4; g=b4; b=b2; break; // light brown
```

```
      case ORANGE:      r=b4; g=b2; b=b0; break;
      case WHITE:       r=b4; g=b4; b=b4; break;
// specific colors for anaglyphic stereo representation
  //    case STEREOBLACK: r=87; g=31; b=b0; break; // black values
  // according to
         // experiments of Thomas Dera 2000-9-13
      case STEREOBLACK: r=0; g=0; b=0; break; // dark values
      // but not necessarily really black. r=0 is necessary
      // however
      case STEREORED:   r=b4; g=b0; b=b0; break;
      case STEREOGREEN: r=b0; g=b4; b=b4; break; // blue-green
// not a color
      case INVISIBLE:   r=-1; g=-1; b=-1; break;
      case STEREO:      r=-1; g=-1; b=-1; break; // same as INVISIBLE
         // even to me my old name INVISIBLE suggested the wrong
         // association after having not worked with it for a while.
         // could also be named ANAGLYPH
      case TRANSPARENT_: r=-137; g=-1; b=-1; break;
      case WRITECOLOR:   r=dark; g=dark; b=dark; break;
      case GRIDCOLOR:    r=bright; g=bright; b=bright; break;
         // should not appear in the previous list, so that curves
         // normally don't get the same color as the grid
      default:  r=0; g=0; b=0; break;
      }
   }

   namespace{
      const Z nUsed=17;
      // we use the first nUsed colors of Palette
   }

   Color::Color(Z colNumber)
   {
      Z mL=4;
      static Word loc("Color(Z)");
      CPM_MA
      Z i=colNumber-1;
      if (i<0) { r=-1; g=-1; b=-1;}
      else{ // else was missing till 2005-04-06
         i=i%nUsed;
            // now i in {0,1,....nUsed-1}
         make((Palette)(i));
      }
      CPM_MZ
   }

   Z Color::toZ(Palette p)
   {
      Z res=0;
      if (p!=INVISIBLE && p!=STEREO){
```

```
         res=1+(Z)p;
         if (res>nUsed) cpmwarning(
            "Color::toZ(Palette): colors beyond WHITE not expected here")\
            ;
      }
      return res;
   }

   Palette Color::toPal(Z i)
   {
      Z j=i-1;
      if (j<0) return STEREO;
      else return Palette(j%nUsed);
   }

namespace{ // anonymous
   const R xg=-0.5;
   const R yg=0.5*sqrt(3.);
   const R2 er(1,0);
   const R2 eg(xg,yg);
   const R2 eb(xg,-yg);
      // er+eg+eb=0,
      // er.absVal()==eg.absVal()==eb.absVal()==1
}

Color::Color(C const& z, Iv mm)
// no trigonometric functions called, fast
   {
      R2 v(z.real(),z.imag());
      R vr=v.normalize_();
      R r_=((v|er)+1.)*0.5;
      R g_=((v|eg)+1.)*0.5;
      R b_=((v|eb)+1.)*0.5;
      // r_+g_+b_=3/2, 0<=r_,g_,b_<=1
      R rho=ImagingTools::response(
         mm.inf(),mm.sup(),0,255, gamma,vr);
      r=(Z)(rho*r_);
      g=(Z)(rho*g_);
      b=(Z)(rho*b_);
   }

   Color::Color(R r, Iv mm)
   {
      static Color cL(RED);
      static Color c0(WHITE);
      static Color cU(BLUE);
      R p=2*(mm.ramp(r)-0.5);
      Color res;
      if (p<0){
         p*=-1;
```

```
         res=p*cL+(1-p)*c0;
      }
      else{
         res=p*cU+(1-p)*c0;
      }
      *this=res;
   }

   Color::Color(R3 const& y)
   {
      static const Iv iv(0,1);
      static const Z n=256;
      static const bool restrict=true; // named quantities
         // support understanding. Using literals as function
         // arguments falls short in this respect.
      r=iv.qun(y[1],n,restrict)-1;
      g=iv.qun(y[2],n,restrict)-1;
      b=iv.qun(y[3],n,restrict)-1;
   }

   R3 Color::toR3()const
   {
      static const IvZ ivz(0,255);
      static const Iv iv(0,1);
      return R3(iv.qunInv(r,ivz),iv.qunInv(g,ivz),iv.qunInv(b,ivz));
   }

//////////////////////////// ImagingTools ////////////////////////

Z ImagingTools::verbose=0;

R ImagingTools::int2(R f11, R f12, R f21, R f22, R p, R q, Z method)
// int2 stands for interpolation in 2 dimensions.
// The returned value is a weighted meen of the fij. Notice
// p+p_=1, q+q_=1, which leads to p*q_+p_*q_+p*q+p_*q=1.
// For interpretation: p and q follow the x and y axis respectively.
// The corners of the rectangle in which the interpolation point is
// placed and to which the reduced coordinates p and q refer are as
// follows:
//
//      f11      f12
//
//      f21      f22
//
// Method=1 means bilinear interpolation and method=0 means
// pixelation. The latter is useful if the number of pixels is to be
// compared to expectations.
{
   if (method==1){ // bilinear interpolation
      R p_=1-p;
```

```
      R q_=1-q;
      return f11*p_*q+f12*p*q+f21*p_*q_+f22*p*q_;
   }
   else if (method==0){ // pixel representation
      if (p<0.5){
         if (q<0.5) return f21; else return f11;
      }
      else{
         if (q<0.5) return f22; else return f12;
      }
   }
   else {
      cpmerror("Frame", "in2: method not implemented");
      return 1.;
   }
}

V<R> ImagingTools::pixelCenters(R a, R b, Z n)
// The interval [a,b] is divided into n subintervalls of equal length
// and the n centers of these subintervals are returned as a  real vector
// in ascending order
{
   Iv iv(a,b);
   return iv.centers(n);
}

void ImagingTools::find(const V<R>& lattice, R x, R& p, Z& i, Z& j)
// Positioning x within an equidistant 1-dimensinal lattice.
// After function call one of three situations is given:
//
// 1. (normal case)      lattice[i] <= x < lattice[j]
//
// 2. x<lattice[1], i=1, j=2
//
// 3. i=n-1, j=n, lattice[n]<=x   (n=dim(lattice))
//
// p=(x-lattice[i])/(lattice[j]-lattice[i]).
// It is assumed that lattice[j+1]-lattice[j]>0 and independent of j,
// whenever none of the indexes is out of range.
{
   Word loc("ImagingTools::find(...)");
   Z n=lattice.dim();
   if (n==1){p=0.;i=1;j=1;return;}
   R first=lattice[1];
   R last=lattice[n];
   cpmassert(first<last,loc);
   R step=(last-first)/(n-1.); // lattice has n components and n-1
      // gaps between values
   if (x<=first){
      i=1;
```

```
      j=2;
   }
   else if (x>=last){
      i=n-1;
      j=n;
   }
   else{
      i=1+(int)((x-first)/step);
         //would be the exact result if the lattice would
         // be exactly equidistant.
         // It is always a numerically perfect solution for
         // the interpolation applicalion since p gets computed
         // accordingly
      j=i+1;
   }
   R lower=lattice[i];
   R upper=lattice[j];
   cpmassert(lower<upper,loc);
   p=(x-lower)/(upper-lower);
      // is negative below first, and larger than 1 above last.
      // This is OK
}

R ImagingTools::response(R x1, R x2, R y1, R y2, R gamma, R x)
// The function x |---> f(x):=response(...,x) has the following
// properties
// f(x1)=y1, f(x2)=y2,
// f(x) is a power function of a linear transform of x.
// 97-4-14: for gamma=-1, f is an exponential function of a linear
// transform of x. This is valuable if the pixel values of the image are
// proportional to log exposure and the response is intended to be a
// power of exposure.
{
   using namespace CpmRootX;
   static const R maxLim=100;
   static const R minLim=0.01;
      // only needed for gamma<0
   if (x2>hugeNumber) x2=hugeNumber;
   if (x<=x1) return y1;
   if (x>=x2) return y2;
   if (x1==x2) return 0.5*(y1+y2);
   R p=(x-x1)/(x2-x1);
   if (gamma==1.){
      return y1+p*(y2-y1);
   }
   else if (gamma==0){
      return y2;
   }
   else if (gamma==0.5){
      return y1+sqrt(p)*(y2-y1);
```

```
   }
   else if (gamma>0){
      return y1+pow(p,gamma)*(y2-y1);
   }
   else{ // gamma<0
      if (x1<minLim) x1=minLim;
      if (x2>maxLim) x2=maxLim;
      if (x<=x1) return y1;
      if (x>=x2) return y2;
      R beta=(y2-y1)/(log(x2)-log(x1));
      R alpha=y2-beta*log(x2);
      return alpha+beta*log(x);
   }
}

R ImagingTools::random(Z i)
{
   return CpmRoot::randomR(i);
}

Z ImagingTools::random(Z i, Z k)
{
   return CpmRoot::ranVal(k,i);
}

///////////////// class Response ////////////////////////////////////

Z1 Response::levels(20);

R Response::operator()(const R& x)const
{
   using CpmRoot::Pi;
   static R piInv=1./Pi;
   static R piHalf=0.5*Pi;

   if (mode==0){
      return
         ImagingTools::response
            (rx.inf(),rx.sup(),ry.inf(),ry.sup(),gamma,x);
   }
   else if (mode==1){
      R p=rx.linear(x,-1,1);
      p=::atan(p);
      p=(p+piHalf)*piInv;
      return ry.inf()+p*ry.absVal();
   }
   else if (mode==2){
      R p=rx.periodic(x);
      return ry.inf()+p*ry.absVal();
   }
```

```
   else if (mode==3){
      R p=rx.linear(x,-cpmpi,cpmpi);
      return ry.inf()+0.5*(::sin(p)+1.)*ry.absVal();
   }
   else if (mode==4){
      R y=rx(x);
      if (y>0) return ry.inf();
      else{
         R p=-y/rx.radius();
         return ry.inf()+p*ry.absVal();
      }
   }
   else if (mode==5){
      R p0=rx.periodic(x); // varies from 0 to 1
      R pexp=2.5;
      R p=::pow(p0,pexp);
      p*=cpmpi; // varies from 0 to pi
      p=::sin(p); // first half-wave of sine used
      return ry.inf()+p*ry.absVal();
   }
   else{
      cpmerror("Response::operator()(R): unvalid mode");
      return 0;
   }
}

Color Response::color(R x)const
{
   if (mode<=5){
      R y=operator()(x);
      return Color(y,y,y);
   }
   else if (mode==6){
      R yL=0.; // don't change
      R yU=1.; // don't change
      R y=ImagingTools::response(rx.inf(),rx.sup(),yL,yU,gamma,x);
      if (y<yL) return Color(DARKGRAY); //should not happen
      if (y>yU) return Color(WHITE); // should not happen
      Iv rg(yL,yU);
      R phi1=CpmRoot::Pi*y; // assumes yL=0, yU=1
      R phi2=1.75*phi1;
      R arg=sin(levels*phi1);
      arg*=arg;
      arg*=arg;
      R val=1-arg;
      C z;
      z.polar(val,phi2);
      Color col(z,rg);
      return col;
   }
```

```
   else{
      return Color();
   }
}

bool Response::prnOn(ostream& str)const
{
 //  CpmRoot::writeTitle("Response",str);
   cpmwat;
   cpmp(rx);
   cpmp(ry);
   cpmp(gamma);
   cpmp(mode);
   return true;
}

bool Response::scanFrom(istream& str)
{
   cpms(rx);
   cpms(ry);
   cpms(gamma);
   cpms(mode);
   return true;
}

/***************** class Roi Region of Interest ************************/

namespace{
Z mult_pix(Z n, R x)
{
   R s=x*n+0.5;
   cpmassert(s>=0,"mult_pix(Z,R)");
   Z res=(Z)s;
   if (res<1) return 1;
   if (res>n) return n;
   return res;
}
}

Roi::Roi(Z m, Z n, R xL, R xU, R yL, R yU)
{
   Word loc("Roi(Z m, Z n, R xL, R xU, R yL, R yU)");
   cpmassert(xL>=0 && xU<=1,loc);
   cpmassert(yL>=0 && yU<=1,loc);
   cpmassert(xL<xU,loc);
   cpmassert(yL<yU,loc);

   Z j1=mult_pix(n,xL);
   Z j2=mult_pix(n,xU);
   Z i1=mult_pix(m,1.-yU);
```

```
   Z i2=mult_pix(m,1.-yL);
   mc=1+i2-i1;
   nc=1+j2-j1;
   si=i1-1;
   sj=j1-1;
}

Roi::Roi(Z m, Z n, Iv ivx, Iv ivy)
{
   *this=Roi(m,n,ivx[1],ivx[2],ivy[1],ivy[2]);
}



///////// class Rect Rectangular Subareas of the unit square /////////////

Rect::Rect(Iv iv1, Iv iv2)
{
   Iv iv0(0,1);
   iv1=iv1&iv0;
   iv2=iv2&iv0;
   xL=iv1.inf();
   xU=iv1.sup();
   yL=iv2.inf();
   yU=iv2.sup();
}

Rect::Rect(R a, R b, R c, R d)
      // defines a sub-rectangle of the unit-square [0,1]x[0,1].
      // If the arguments are as intended, we have
      // 0<=a<b<=1
      // 0<=c<d<=1.
      // Then the sub-rectangle is simply [a,b]x[c,d]
      // If they are out of range or out of order, they will be brought
      // back in an obvious manner
{
      if (a<=b){
         xL=a;
         xU=b;
      }
      else{
         xL=b;
         xU=a;
      }

      if (c<=d){
         yL=c;
         yU=d;
      }
      else{
```

```
         yL=d;
         yU=c;
      }

      if (xL<0) xL=0;
      if (xU>1) xU=1;
      if (yL<0) yL=0;
      if (yU>1) yU=1;
}

Rect::Rect(Z c1x, Z c1y, Z c2x, Z c2y, Z cx, Z cy)
{
   Word loc("Rect(Z c1x, Z c1y, Z c2x, Z c2y, Z cx, Z cy)");
   cpmassert(cx>0,loc);
   cpmassert(cy>0,loc);
   R cxInv=1./cx;
   R cyInv=1./cy;
   R a=c1x*cxInv;
   R b=c2x*cxInv;
   R c=1.-c1y*cyInv;
   R d=1.-c2y*cyInv;
   *this=Rect(a,b,c,d);
}

namespace{
void auxSub(R a1, R a2, R b1, R b2, R& c1, R& c2)
   // determining subinterval (c1,c2) of (0,1) as subinterval '(b1,b2)'
   // of (a1,a2). The 'subinterval (b1,b2)' is given 'relative to
   // (a1,a2)' i.e. considering (b1,b2) as a unit-interval.
{
   R d=a2-a1;
   c1=a1+b1*d;
   c2=a1+b2*d;
}
}

Rect Rect::sub(const Rect& rec)const
   // returns the rectangle which is formed by taking rec of *this
   // in a way that *this is now considerd a unit square
{
   R x1,x2,y1,y2;
   auxSub(xL,xU,rec.xL,rec.xU,x1,x2);
   auxSub(yL,yU,rec.yL,rec.yU,y1,y2);
   return Rect(x1,x2,y1,y2);
}

bool Rect::prnOn(ostream& str)const
{
   cpmwat;
   cpmp(xL);
```

```
   cpmp(xU);
   cpmp(yL);
   cpmp(yU);
   return true;
}

bool Rect::scanFrom(istream& str)
{
   cpms(xL);
   cpms(xU);
   cpms(yL);
   cpms(yU);
   return true;
}

//////////////////// class Mask ////////////////////////////////////////

Mask Mask::invert(void)const
{
   return Mask(-di_,-dj_,w_);
}

R Mask::rms(void)const
{
   Word loc("CpmImaging::Mask::rms(void)");
   R res=0;
   R wTot=0.;
   R r2,dx,dy,wi;
   for (Z i=1;i<=dim();i++){
      dx=di_[i];
      dy=dj_[i];
      r2=dx*dx+dy*dy;
      wi=cpmabs(w_[i]);
      res+=wi*r2;
      wTot+=wi;
   }
   cpmassert(wTot>0.,loc);
   res/=wTot;
   cpmassert(res>=0.,loc);
   return sqrt(res);
}

Mask::Mask(Z n)
{
   if (n==0){ // trivial mask that will make a convolution trivial
      di_=V<Z>(0);
      dj_=V<Z>(0);
      w_=V<R>(0);
      return;
   }
```

```
   Z nOdd=(n%2 ? n : n+1);
      // n is odd then the modulo 2 operation % gives 1
   Z nD=nOdd*nOdd;
   cpmassert(nD>0,"Mask::Mask(Z n)");
   R weight=1./nD;

   Va<R> w1(nD,weight);
   w_=w1;

   Va<Z> di1(nD);
   Va<Z> dj1(nD);
   Z i,j,k=1;
   Z nShift=(nOdd+1)/2;
   for (i=1;i<=nOdd;i++){
      for (j=1;j<=nOdd;j++){
         di1[k]=i-nShift;
         dj1[k]=j-nShift;
         k++;
      }
   }
   di_=di1;
   dj_=dj1;
}

Mask Mask::bin3()
{
   V<Z> di("",-1,0,1,-1,0,1,-1,0,1);
      // here we need 9 entries, this kind of list
      // constructor is defined for up to 10 entries.
   V<Z> dj("",1,1,1,0,0,0,-1,-1,-1);
   V<R> wp("",1.,2.,1.,2.,4.,2.,1.,2.,1.);
   Va<R> w(wp);
   R i16=1./16;
   w*=i16;
   return Mask(di,dj,w);
}

Mask::Mask(R rMax, const F<R,R>& f)
{
   Word loc("Mask::Mask(R rMax, const F<R,R>& f)");
   if (rMax<=0.){ // trivial mask that will make a convolution trivial
      di_=V<Z>(0);
      dj_=V<Z>(0);
      w_=V<R>(0);
      return;
   }
   Z i,j,nD=0;
   Z lim=(Z)rMax;
   R rMax2=rMax*rMax;
   R val;
```

```
   // counting the contributions needed
   for (i=-lim;i<=lim;i++){
      for (j=-lim;j<=lim;j++){
         val=i*i+j*j;
         if(val<=rMax2) nD++;
      }
   }
   Va<R> w1(nD);
   Va<Z> di1(nD);
   Va<Z> dj1(nD);
   Z k=0;
   R r;
   for (i=-lim;i<=lim;i++){
      for (j=-lim;j<=lim;j++){
         val=i*i+j*j;
         if(val<=rMax2){
            k++;
            r=sqrt(val);
            val=f(r);
            cpmassert(val>=0.,loc);
            w1[k]=val;
            di1[k]=i;
            dj1[k]=j;
         }
      }
   }
   R wSum=0.;
   for (k=1;k<=nD;k++){
      wSum+=w1[k];
   }
   cpmassert(wSum>0,loc);
   wSum=1./wSum;
   w_=w1*wSum;
   di_=di1;
   dj_=dj1;
}

namespace{
   R fAux(const R& r, const R& r0)
   {
      R rRel=r/r0;
      return rRel>1 ? 0 :1-rRel*rRel;
   }
}

Mask Mask::parabolic(R rMax)
{
   return Mask(rMax,F1<R,R,R>(rMax)(fAux));
}
```

98 cpmimg24.h

```
//? cpmimg24.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_IMG24_H_
#define CPM_IMG24_H_

/*
   Description: Declaration and partial definition of class
   Img24 which serves as memory for graphics windows and also
   allows to use all graphical functions for console applications
   to represent screen images as image files of *.ppm format.

*/

#include <cpmviewport.h>
#include <cpminterval.h>

namespace CpmGraphics {

   using CpmRoot::Z;
   using CpmRoot::L;
   using CpmRoot::R;
   using CpmRoot::Rh;
   using CpmRoot::Word;
   using CpmRootX::B;
   using CpmRootX::R1;
   using CpmLinAlg::R2;
   using CpmLinAlg::R3;
   using CpmGeo::Iv;
   using CpmArrays::Vl;
   using CpmArrays::VV;
   using CpmArrays::VVV;
   using CpmArrays::IvZ;

////////////////////////// enum ColSty ////////////////////////////////////////

enum ColSty {SET_COLOR,ADD_COLOR,MIX_COLOR,STEREO};
      // color style

//////////////////////// class TriAng ////////////////////////////////////////

class TriAng{
// triangles in the plane, geometry only as needed for graphics
   // the plane is considered as R2
   // No restrictions concerning the points.
   // Lean class dealing only with aspects relevant to graphics
```

```
//independent quantities
   R2 p1_,p2_,p3_;
public:
   Word nameOf()const{ return Word("TriAng");}
      //: name of
   TriAng(R2 const& p1, R2 const& p2, R2 const& p3):
      p1_(p1),p2_(p2),p3_(p3){}
      // constructor from three points

   TriAng shf(R2 const& s)const{ return TriAng(p1_+s,p2_+s,p3_+s);}
      //: shift

   IvZ i1()const;
      //: i1
      // returns the smallest IvZ which contains the set
      // {p1_.x1, p2_.x1, p3_.x1}

   IvZ i2()const;
      //: i2
      // returns the smallest IvZ which contains the set
      // {p1_.x2, p2_.x2, p3_.x2}

   R area()const;
      //: area
      // this area is positive iff the rotation sense
      // p1 -> p2 -> p3 is positive (i.e. counter clock wise)

   bool out(R2 const& p)const;
      //: out
      // returns true iff p is outside the triangle *this
      // Source: see function barCor

   R3 barCor(R2 const& p)const;
      //: barycentric coordinates
      // returns the barycentric coordinates of p with respect
      // to the triangle *this.
      // Source: G. Aumann, K. Spitzmueller:
      // Computerorientierte Geometrie, BI 1993,  p.43
};

//////////////////////////////////// class Img24 ///////////////////////

class Img24{ // class of 24-bit (R,G,B)-images
// Notice that in the CpmGraphics system, the generation and display of
// screen graphics always make uses of Img24.

// Data (all private)

   Z width;
   // Width of the bitmap
```

```
Z height;
// height of the bitmap

Vl< Vl<ColRef> > bh_;
   // indexing corresponds to considering the image rectangle as
   // a mathematical matrix; indexing is however starting at 0.

Rec displayBuffer;
   // see cpmviewport.h for class Rec. As the name says, this data
   // member holds (in his linear image memory) the pixel values
   // of *this which can be displayed via our viewport(The size of
   // *this may be much larger than any display).

B readyForDisplay;
   // true says that the data from bh_ have been mapped consistently
   // to displayBuffer so that calling displayBuffer.dis() shows
   // the part of *this which fits the viewport.

ColRef c0;
   // Base color of the image with which the bitmap gets filled
   // on initialization in all cases that don't provide
   // pixel data.

ColSty sty;
   // sets the color mixing mode of graphical actions
   // The meaning of color adding in case of sty==ADD_COLOR
   // is determined by the static public variable
   // rgb::addingMethod for which presenty values 0,1,2
   // have a meaning

B antiAliasLines;
   // influences only function line(...) in a way that is clear
   // from the name

B thickLines;
   // influences only function line(...) in a way that is clear
   // from the name.
   // There is only a single method
   // of thickening which makes horizontal and vertical lines
   // 3 pixels wide and thus effectively covers pixel irregularities
   // in fine-grained polygons. Graphics generated with thick lines
   // can be converted (with some image processor) into PNG and
   // the included in LATEX-documents. Without thickening graphics
   // which contain horizontal or vertical lines look very
   // poor in pdflatex-generated documents since some of the lines
   // are visible, others not.

R1 gamma;
   // controls how to add colors
```

```
// private functions

   xy dis(R x, R y)const // discretize
      // returning the discrete representative of the
      // continuous location (x,y). Meaningful only for
      // (x,y) \in W:=[0,width]x[0,height]
   {
      return xy(x==width ? width-1 : (Z)x,
         y==height ? height-1 : (Z)y );
   }

   Z disX(R x)const
      // For all R x, R y the term
      // bh_[disY(y)][disX(x)] is guaranteed to be
      // well-defined.
   {
      if (x>=width) return width-1;
      else if (x<=0) return 0;
      else return (Z)x; // x always positiv, has to be fast
   }

   Z disY(R y)const
      // see Z disX(R x)const
   {
      if (y>=height) return height-1;
      else if (y<=0) return 0;
      else return (Z)y;
   }

   L outcode(R x, R y)const
      // Sutherland's outcode. Setting bits intelligently
   {
      L res=0;
      if (x<0) res+=8;        // 8=2^3
      if (x>width) res+=4;    // 4=2^2
      if (y<0) res+=2;        // 2=2^1
      if (y>height) res+=1;   // 1=2^0
      return res;
   }

   bool clip(R& x1, R& y1, R& x2, R& y2)const;
      // The points (x1,y1) and (x2,y2) determine a closed line
      // segment S. The image *this determines the closed
      // rectangle W:=[0,width]x[0,height] in R^2 (sic, no -1 !).
      // We consider D := section of sets S and W.
      // There are two cases:
      // (1) D == void set
      // (2) else
      // The function returns false in the first case and true else.
```

```
      // In the second case the function potentially modifies the
      // arguments (which is possible since these are references) in such
      // a way that D is the closed line segment determined by (x1,y1)
      // and
      // (x2,y2).

    Img24 intPolAux(Z mI, Z nI, Z method)const;
      // tool for implementing function interpolate

    Rec toRec();
      //: to rectangle

    Img24(Rec const& r);
   // construcor from graphical rectangles with linear byte memory

    typedef Img24 Type;

public:
   CPM_IO

   Img24();
   // trivial constructor which does not allocate memory

   explicit Img24(ColRef c, R fracX=1, R fracY=1);
      // Image of uniform color c which fills the viewport
      // according to the fractions given for x and y.
      // sty=SET_COLOR, gamma=Viewport::getGamma()

   Img24(Z w, Z h, rgb c=rgb(), ColSty cs=SET_COLOR);
      // constructor which allocates memory for x*y pixels and initialize
      // id with Pix(r,g,b)
      // Z w: Width of the bitmap
      // Z h: Height of the bitmap
      // We use SET_COLOR as the default so that classes Img24
      // and Viewport, behave as exchangable as possible

explicit Img24(Word const& Filename);
   // constructor from a *.ppm (binary portable pixmap,
   // classified P6 or P3).
   // P6 has all data beyond 'depth' in binary format; i.e.
   // each character counts, no one is overread as whitespace, new-line
   // or something like that. Therefore P6 is faster to read and to write
   // and creates smaller files. Example of usage:
   //    Img24 im2("pisces.ppm");

explicit Img24(const VVV<L>& vv);
   // generation from 'standard imaging data type'

explicit Img24(const VV<L>& vv);
   // generation of a gray image from 'standard imaging data type'
```

```
explicit Img24(const VV<ColRef>& vv);
   // generation from a 'standard imaging data type'

explicit Img24(VVV<R> const& vv);
   // Generation from floating point pixeldata. Such data can be
   // retreived from instances of my CpmImaging::MultiImage<T> for T=R
   // (see cpmmultiimage.h) which is capable of many image processing
   // operations such as Fourier filtering.
   // For vv.dim3()<3 a gray image will be created
   // (using the first image plane only). And for vv.dim3()>=3
   // a color image will be created (using the lower 3 image
   // planes only). If the values don't fit the range [0,255]
   // they will scaled by a linear transformation (same for all
   // image planes) to attain minimum 0 and maximum 255.
   // Small negative values (>-0.5) as may result from image
   // processing in floating point numbers are dealt with by a slight
   // shift and no multiplicative scaling. So natural color images
   // don't get automaticly boosted in a way that the value 255
   // appears in one channel at least.

VVV<L> toVVV()const;
   // creation of an equivalent 'standard imaging data type'
   // Let res be the return value. Then res[1] is the red
   // image matrix (i.e. res[1][1][1] left upper corner,
   // res[1][res.dim2()][res.dim3()] right lower corner).
   // res[2] is the green one, and res[3] the blue one.

VV<ColRef> toVVColRef()const;
   // creation of an equivalent 'standard imaging data type'
   // Let res be the return value. Then res[1][1] is the
   // color of the left upper corner,

VV<R> toVVR(R wR=1, R wG=1, R wB=1, Iv val=Iv(0,1))const ;
   // returns a real image matrix as a gray level image
   // (obtained by forming an weighted average of
   // the r,g,b-plane with weights (wR,wG,wB)/(wR+wG+wB).
   // The values of the image are such that (r,g,b)=(0,0,0)
   // corresponds to val.inf() and (r,g,b)=(255,255,255)
   // to val.sup()

Vl<ColRef> const& operator[](Z i)const{ return bh_[i];}
   // returns the i-th line of the image (i=0,...height-1)
   // in a form that allows to pass it over to image
   // processing algorithms in a fast way (notice that
   // we don't allow to change it

// Handling color style
   void setCol(){ sty=SET_COLOR;}
   void addCol(){ sty=ADD_COLOR;}
```

```
   void stereo(){ sty=STEREO;}
   ColSty getSty()const{ return sty;}
   void setSty(ColSty sty_){ sty=sty_;}

// Handling anti-aliasing, thick lines, and assumed display gamma
   bool getAntiAliasLines()const{ return antiAliasLines;}
   void setAntiAliasLines(bool aal=true){ antiAliasLines=B(aal);}

   void setThickLines(bool b=true){ thickLines=B(b);}
   bool getThickLines()const{ return thickLines;}

   R getGamma()const{ return gamma;}
   void setGamma(R g){ gamma=R1(g);}

// Handling base color
   void setColor(rgb c){ c0=ColRef(c);}
      // notice that filling the image with this color is the
      // task of function clear.
   rgb getColor()const{ return rgb(c0);}

// Initializing change of image size.
   void resize(Z widthNew, Z heightNew);
   // We re-set the size of the image data matrix and
   // initializes it with with c0. Input <0 will be handled as 0.
   // Thus, no logical exceptions. For large input, storage allocation
   // may fail however.
   // This action takes place even if width==widthNew and
   // height==heightNew.

// Descriptors for image size
   bool isVoid()const{ return width<=0 || height<=0;}

   Z getLin()const{ return height;}
      // number of pixel lines

   Z getCol()const{ return width;}
      // number of pixel columns

   Z dim1()const{ return height;}
   Z dim2()const{ return width;}
      // for uniformity with matrices. Considers the image
      // as a pixel matrix in the natural manner
      // For uniformity with Image<>

   Z w()const{ return width;}
   Z h()const{ return height;}

   IvZ i1()const{ return IvZ(0,width-1);}
      // index-range in 1-direction
```

```
   IvZ i2()const{ return IvZ(0,height-1);}
      // index-range in 2-direction

   Iv r1()const{ return Iv(0.,width);}
      // real range in 1-direction, the interval is to be considered
      // as [0,width)

   Iv r2()const{ return Iv(0.,height);}
      // real range in 2-direction, the interval is to be considered
      // as [0,height)

   R xr(R xi)const{ return xi*width;}
      // x from a 'relative coordinate' xi which varies in [0,1]

   R yr(R eta)const{ return eta*height;}
      // y from a 'relative coordinate' eta which varies in [0,1]

    bool out(R x, R y)const
      // Returns true if (x,y) is outside the image area
      // A=[0,width]x[0,height]
   {
      if (x<0) return true;
      if (y<0) return true;
      if (x>width) return true;
      if (y>height) return true;
      return false;
   }

   bool out(Z i, Z j)const
      // Returns true if (i,j) is outside the image area
      // [0,width-1]x[0,height-1]
   {
      if (i<0) return true;
      if (j<0) return true;
      if (i>=width) return true;
      if (j>=height) return true;
      return false;
   }

// Showing image data

   void display(Z sx=0, Z sy=0);
      // Displays *this on Viewport in a way that the left upper corner
      // of the image rectangele (on screen) is the screenpoint xy(sx,sy)

// Reading image data

    rgb getPel(Z i, Z j)const
   // Save reading access for discrete location. If (i,j)
   // is outside the image area, the background color of the
```

```
   // image gets returned.
   {
      if (out(i,j)) return rgb(c0);
      return rgb(bh_[j][i]);
   }

   rgb getPel(R x, R y)const
   // Save reading access for continuous location. If (x,y)
   // is outside the image area, the background color of the
   // image gets returned.
   {
      if (out(x,y)) return rgb(c0);
      xy z=dis(x,y);
      return rgb(bh_[z.y][z.x]);
   }

// Changing image data

   void clear(void);
   // Fills the Bitmap with base color c0

   void paint(rgb c){ setColor(c); clear();}
   // Fills the Bitmap with base color c and changes c0 to c

   void frame(rgb c=rgb(255,0,0));
   // sets the rim pixels to color c

   void putPel(Z x, Z y, rgb c, R w=1)
   // Puts a pixel in a 24 bit bitmap at the position x, y to
   // the color specified in c.
   // Meaningful is 0<=x<width, 0<=y<height. If (x,y) violate this
   // there is no problem: the function simply does nothing in this case.
   // pix is a struct with the elements r, g, b.
   // colorStyle: SET_COLOR the new color is painted over the old one,
   //             ADD_COLOR the new color is added to the old one
   //                adding identical colors increases brightness
   //             STEREO adds colors in a simple manner which makes
   //                sense for colors being either (255,0,0) or
   //                (0,255,255) as is the case in anaglyphic
   //                represetation
   //             MIX_COLOR the new color is mixed with the old one
   //                mixing identical colors gives no change
   // It is only in the last case that the weight parameter w gets used.
   // w near 1 indicates dominance of the new color and w near 0
   // indicates dominance of the old color (e.g. background color).
   // Since access is over operator[] of class Vl<>, indexes out of range
   // will stop the program with an error depending on
   // defined(CPM_RANGE_CHECK).
   // (x,y)=(0,0) is the upper left corner and
   // (x,y)=(width-1,height-1) is the lower right one (graphics pixel
```

```
// coordinates).
// Thus the x-axis is as usual and the y-axis goes downwards
// inlining for utmost efficiency
{
   if (out(x,y)) return;
   if (sty==SET_COLOR) bh_[y][x]=c;
   else{ ColRef cB=bh_[y][x]; bh_[y][x]=mix(cB,c,w);}
   readyForDisplay=B();
}

void putPel(R x, R y, rgb pix, R w=1)
   // Defined for all (x,y). But does nothing for (x,y) outside
   // the image area
{
   if (out(x,y)) return;
   xy z=dis(x,y);
   putPel(z.x,z.y,pix,w);
}

void lineThin(R x1, R y1, R x2, R y2, rgb c);
// Draws the clipped part of the line from x1, y1 to x2, y2.
// Clipping is performed by the Cohen-Sutherland algorithm.
// If the line is horizontal, a fast implementation based
// on non-public function horLine is chosen.
// The color of the line, specified in r, g, b, is combined with the
// already existing color of each pixel according to the value of sty.
// Depending on the attribute antiAliasLines the representation
// reduces aliasing significantly if gamma has a value corresponding to
// the system (e.g. 2.2).
// The R-typed arguments actually enter ony through their integer
// parts disX(x1),disY(y1),disX(x2),disY(y2).
// This gives good-looking lines, but at the edges of polygons ugly
// patterns like
//              x
//            x
//          x x
//        x
// may appear. Notice that the C+- graphical system allows to create
// arbitrarily large Img24 objects and write them to file as
// ppm-images. If such large images get processed by a, say, 1.5 pixel
// wide unsharp mask, one easily obtains print quality polygons.
// Also employing function line(...) instead of lineThin(...)
// together with thickLines==true  conceals irregularities of polygons.

void line(R x1, R y1, R x2, R y2, rgb c)
   // as lineThin, but thickness controlled by data member thickLines
{
   lineThin(x1,y1,x2,y2,c);
   if (thickLines){
      lineThin(x1+1,y1,x2+1,y2,c);
```

```
      lineThin(x1-1,y1,x2-1,y2,c);
      lineThin(x1,y1+1,x2,y2+1,c);
      lineThin(x1,y1-1,x2,y2-1,c);
   }
}

void fillRectangle(R x1, R y1, R x2, R y2, rgb c);
// Fills a rect with the RGB color specified in c. x1, y2 is the
// upper left corner (beginning with 0, 0) and x2, y2 is the lower
// right corner of the recangle to fill. The color specified in r, g,
// b is not added to the existing background color but taken as it is.
// Now no longer implemented by putPel but by much faster horLine.

void fill(TriAng const& tri, rgb c);
// Fills a triangle tri with the color specified in c.

void fillEllipse(R x1, R y1, R x2, R y2, rgb c);
// The function paints a solid ellipse into the rectangle defined
// by x1, y1 (left, top) and x2, y2 (right, bottom).
// Actually only the part of the ellipse wich lies in the
// image area gets painted. The color c, gets added to the existing
// background color.

bool horLine(Z x1, Z x2, Z y, rgb c, R w=1);
// Draws a line from (x1,y) to (x2,y). The color of the line,
// specified in r, g, b, is combined with the already existing color
// of each pixel according to the value of sty.
// Depending on the attribute antiAliasLines the representation
// reduces aliasing significantly if gamma has a value corresponding to
// the system (e.g. 2.2)
// The weight parameter w has the same meaning as for putPel.
// Returns true if at least one pixel was addected, else false.

void horIns(xy p, Vl<ColRef> const& line,
   ColRef cr=ColRef());
   // Horizontal insert, no restrictions to the length of the line.
   // Only the part that fits the frame will be used. Insertion
   // is always in color setting mode (the old image is locally
   // replaced by the color data from line. p is the first point in
   // which the replacement becomes active, and the replacement goes
   // of course from left to right. If cr has not the default value,
   // it causes an admixture of this color in a way that was intro-
   // duced for implementing writing letters. Special usage, not of
   // general relevance.

void insert(xy p, Img24 const& img , ColRef cr=ColRef());
   // the image img replaces the part of *this which is determined
   // by the size of img and the prescription that p is the
   // left upper corner of the replacement area. If due to
   // the size of img and/or the location of p *this can't
```

```
      // incorporate the whole of img, an appropriate outcut
      // of img will be used for this purpose.
      // Since 2005-09-01 implementation is based on the fast
      // function horIns and no longer by looping over individual
      // pixels.
      // ColRef argument seems to play a role in my font generation
      // system. Finction has to be deduced from code. Not needed
      // in most applications.

// storing image data on file

//   Word write(const Word& Filename="img24_write", bool binary=true);
   Word write(const Word& Filename="img24_write", bool binary=false);

   // The function writes a bitmap to Filename+"extension".
   // Return value is the file name with proper extension added by the
   // function.
   // "extension"=".ppm"
   // The binary=true gives compact files of portable pixmap P6 format
   // that can be used to create a new  image via the constructor
   // Img24(const Word&). binary=false gives portable pixmap P3 format
   // in a form that can be easily interpreted by a text editor. A
   // complete legende gives date of generation and also the value of the
   // data member text. Explanatory text is preceded by a '#' so that a
   // ppm-reader overreads them and can display the image
   // directly. My constructor reads both variants correctly.
   // Note 2008-03-06 at least working under MS compiler, binary sometimes
   // gets gets reconstructed badly and (missing R-plane ?) and the
   // PhotoImpact program fails to read some of the frames. The binary=
   // false setting works correctly. So P3 should be preferred over P6 in
   // my setting. The Mingw-executable works (at least in this case)
   // perfectly also for binary=true

// Auxiliary functions

   static ColRef miX(ColRef const& cB, ColRef const& c,
      ColSty const& s, R const& g, R const& w);
      // this is the primary definition of making
      // display color from existing background color cB
      // and the color c that 'is aimed for' in the
      // graphical action calling this function.
      // g is a gamma.

   ColRef mix(ColRef const& cB, ColRef const& c, R w)const
      // making this a member function; notice that making
      // F-typed function objects as needed in the fast
      // definition of horLine, one needs miX and not mix
      // Here Img24::gamma comes in.
   {
      if (sty==STEREO) return cB.add(c);
```

```
      else return miX(cB,c,sty,gamma,w);
   }

   Img24 interpolate(Z m, Z n, Z method)const;
      // Returns a m*n Image created from *this by a interpolation
      // method selected by the parameter method.
      // method =1 means bilinear interpolation and method=0 means
      // pixelation. The latter is useful if the number of pixels is to
      // be compared to expectations.
      // Also works well for m << height and n << width in which case
      // a combination of interpolation and averaging is done.

   Img24 pair(Img24 const& img, Z gap=2)const;
      //: pair
      // returns a paired image made from *this and img.
      // Assumes that this two images have the same height.
      // The width of the result is the sum of the widths of the
      // components. No interpolation. gap is the width of
      // a separating gap between the images

   Word nameOf()const{ return "Img24";}

   static ColRef biLin(ColRef f11, ColRef f12, ColRef f21, ColRef f22,
      R w11, R w12, R w21, R w22, Z method);
   // law for bi-linear interpolation (for method==1, else pixelation)

};

} // namespace

#endif
```

99 cpmimg24.cpp

```
//? cpmimg24.cpp
//? Status of work 2008-10-25.
//?
#include <cpmimg24.h>
#include <cpmsystemdependencies.h>
#include <cpmsystem.h>
#include <cpmviewport.h>
#include <cpmgreg.h>
#include <cpmimagingtools.h>

using namespace CpmSystem;
using namespace CpmRoot;
using CpmSystem::IFileStream;
using CpmSystem::OFileStream;
using CpmGraphics::Rec;
using CpmGraphics::Viewport;
using CpmArrays::IvZ;
using CpmArrays::V;
using CpmArrays::Vo;
using CpmArrays::X2;
using CpmFunctions::F;
using CpmFunctions::F1;
using CpmFunctions::F4;
using CpmImaging::ImagingTools;
using CpmLinAlg::Z2;
using CpmLinAlg::R2;
using CpmLinAlg::R3;

namespace CpmGraphics {

//////////////////////// class TriAng ////////////////////////////////////

   R TriAng::area()const
   {
      R res=(p1_.x2+p3_.x2)*(p3_.x1-p1_.x1)
           +(p3_.x2+p2_.x2)*(p2_.x1-p3_.x1)
           +(p1_.x2+p2_.x2)*(p1_.x1-p2_.x1);
      return res*0.5;
   }

   bool TriAng::out(R2 const& p)const
      // fast implementation which needs not to compute
      // full barycentric coordinates and also needs not always
      // consider all points
   {
      using CpmRootX::R1;
      R a1=TriAng(p,p2_,p3_).area();
```

```
      R a2=TriAng(p1_,p,p3_).area();
      R s1=R1(a1).sign();
      if (s1!= R1(a2).sign()) return true;
      R a3=TriAng(p1_,p2_,p).area();
      if (s1!= R1(a3).sign()) return true;
      return false;
   }

   R3 TriAng::barCor(R2 const& p)const
   {
      Word loc("TriAng::baryCenCor(R2)");
      R a=TriAng(p1_,p2_,p3_).area();
      cpmassert(a!=0,loc);
      R aInv=cpminv(a);
      R a1=TriAng(p,p2_,p3_).area();
      R a2=TriAng(p1_,p,p3_).area();
      R a3=TriAng(p1_,p2_,p).area();
      return R3(a1*aInv,a2*aInv,a3*aInv);
   }

   IvZ TriAng::i1()const
   {
      return IvZ(p1_.x1,p2_.x1)|IvZ(p2_.x1,p3_.x1);
   }

   IvZ TriAng::i2()const
   {
      return IvZ(p1_.x2,p2_.x2)|IvZ(p2_.x2,p3_.x2);
   }

//////////////////////////// class Img24 ////////////////////////////////

ColRef Img24::miX(ColRef const& cB, ColRef const& c,
   ColSty const& s, R const& g, R const& w)
{
   if (s==SET_COLOR) return c;
   if (s==STEREO) return cB.add(c);
   R w1=1,w2=1;
   if (s!=ADD_COLOR){ w1=1-w; w2=w;}
   // w is the weight of the requested color not of
   // the background! Remember that 1 is the default
   // for w, which would be nonsense if this would
   // be the other way round.
   return rgb::mix((rgb)cB,(rgb)c,w1,w2,g);
}

// We set gamma 1 per default (and not equal to
// Viewport::gamma) since this is much faster

Img24::Img24():
```

```
   width(0),height(0),c0(),sty(SET_COLOR),gamma(1){}

Img24::Img24(ColRef c, R fracX, R fracY):
   c0(c),sty(SET_COLOR),gamma(Viewport::getGamma())
{
   Z mL=3;
   static Word loc("Img24(ColRef,R,R)");
   width=(Z)(Viewport::win().w()*fracX);
   height=(Z)(Viewport::win().h()*fracY);
   if (width && height){
      bh_=Vl<Vl<ColRef> >(height,Vl<ColRef>(width,c0));
   }
   cpmmessage(mL,loc&" done");
}

Img24::Img24(Z x, Z y, rgb c, ColSty cs):
   c0(c),sty(cs),gamma(1)
{
   Z mL=3;
   static Word loc("Img24(Z,Z,rgb,ColSty)");
   cpmmessage(mL,loc&" started: x="&cpmwrite(x)&" y="&cpmwrite(y));
   if (x<0){
      x=0;
      cpmwarning("Img24(Z x,Z y,rgb,ColSty): input x<0 replaced by x=0");
   }

   if (y<0){
      y=0;
      cpmwarning("Img24(Z x,Z y,rgb,ColSty): input y<0 replaced by y=0");
   }

   width=x;
   height=y;
   if (width && height){
      bh_=Vl<Vl<ColRef> >(height,Vl<ColRef>(width,c0));
   }
      // notice that mem_ does not hold all the same byte, so that a
      // specially simple initialization of mem_ is not at hand
   cpmmessage(mL,loc&" done");
}

Img24::Img24(const VVV<L>& vv):width(vv.dim3()),height(vv.dim2()),
   sty(SET_COLOR),gamma(1)
{
   Z mL=4;
   cpmmessage(mL,"Img24(const VVV<L>& ) started");
   if (vv.dim1()!=3) cpmerror("Img24(const VVV<L>& vv): vv.dim()!=3");
   bh_=Vl<Vl<ColRef> >(height,Vl<ColRef>(width,c0));
   for (Z i=1;i<=height;i++){
      for (Z j=1;j<=width;j++){
```

```
         L r=vv[1][i][j];
         L g=vv[2][i][j];
         L b=vv[3][i][j];
         rgb c(r,g,b);
         bh_[i-1][j-1]=c; // matrix style
      }
   }
   cpmmessage(mL,"Img24(const VVV<L>& ) done");
}

namespace {

   rgb con(R x, R y, R z, R alpha, R beta)
   {
      R xn=x*alpha+beta;
      R yn=y*alpha+beta;
      R zn=z*alpha+beta;
      return rgb((Z)xn,(Z)yn,(Z)zn);
   }
}

Img24::Img24(VVV<R> const& vv):width(vv.dim3()),height(vv.dim2()),
   sty(SET_COLOR),gamma(1),c0()
{
   Z mL=3;
   Word loc("Img24::Img24(VVV<R>)");
   CPM_MA
   Z nP=vv.dim1();
   cpmassert(nP>0,loc);
   if (nP<3) nP=1; else nP=3;
   R vHuge=1e64; //
   R vMin=vHuge, vMax=-vMin;
   Z i,j,k;
   for (i=1;i<=height;++i){
      for (j=1;j<=width;++j){
         for (k=1;k<=nP;++k){
            R v=vv[k][i][j];
            if (v>vMax) vMax=v;
            if (v<vMin) vMin=v;
         }
      }
   }
   bool direct=vMin>-0.5 && vMax<255.5;
   R alpha=1, beta=0;
   if (direct){
      if (vMin<0) beta=-vMin;
      // here we only shift a bit to eliminate tiny negative values
      // which may arrise from image processing algorithms with
      // floating point arithmetics.
   }
```

```
   else{ // image is out of range [0,255]
      R diff=vMax-vMin;
      if (diff==0.){ // image is constant. It will be
         // scaled to gray
         alpha=128./vMax;
         // if vMax would be 0, we had vMax=vMin=0. All values would
         // be zero, and we would have the case 'direct'
      }
      else{ // scaled to exact range [0,255]
         alpha=255./diff;
         beta=-alpha*vMin;
      }
   }

   bh_=Vl<Vl<ColRef> >(height,Vl<ColRef>(width,c0));
   for (i=1;i<=height;++i){
      for (j=1;j<=width;++j){
         R v1=vv[1][i][j],v2=v1,v3=v1;
         if (nP==3){
            v2=vv[2][i][j];
            v3=vv[3][i][j];
         }
         rgb c=con(v1,v2,v3,alpha,beta);
         bh_[i-1][j-1]=c; // matrix style
      }
   }
   CPM_MZ;
}

Img24::Img24(const VV<L>& vv):width(vv.dim2()),height(vv.dim1()),
   sty(SET_COLOR),gamma(1)
{
   Z mL=4;
   cpmmessage(mL,"Img24(const VV<L>& ) started");

   bh_=Vl<Vl<ColRef> >(height,Vl<ColRef>(width,c0));
   for (Z i=1;i<=height;i++){
      for (Z j=1;j<=width;j++){
         L r=vv[i][j];
         rgb c(r,r,r);
         bh_[i-1][j-1]=c; // matrix style
      }
   }
   cpmmessage(mL,"Img24(const VV<L>& ) done");
}

Img24::Img24(const VV<ColRef>& vv):width(vv.dim2()),height(vv.dim1()),
   sty(SET_COLOR),gamma(1)
{
   Z mL=4;
```

```
   Word loc("Img24(VV<ColRef>)");
   cpmmessage(mL,loc&" started");
   bh_=Vl<Vl<ColRef> >(height,Vl<ColRef>(width,c0));
   for (Z i=1;i<=height;i++){
      for (Z j=1;j<=width;j++){
         bh_[i-1][j-1]=vv[i][j]; // matrix style
      }
   }
   cpmmessage(mL,loc&" done");
}

void Img24::resize(Z x, Z y)
{
   Z mL=3;
   static Word loc("Img24::resize(Z,Z)");
   cpmmessage(mL,loc&" started: x="&cpm(x)&" y="&cpm(y));
   if (x<0){
      x=0;
      cpmwarning(loc&": input x<0 replaced by x=0");
   }
   if (y<0){
      y=0;
      cpmwarning(loc&": input y<0 replaced by y=0");
   }
   width=x;
   height=y;
   if (width && height){
      bh_=Vl<Vl<ColRef> >(height,Vl<ColRef>(width,c0));
   }
   readyForDisplay=B();
   CPM_MZ
}

void Img24::frame(rgb c)
{
   line(0,0,width-1,0,c);
   line(0,height-1,width-1,height-1,c);
   line(0,0,0,height-1,c);
   line(width-1,0,width-1,height-1,c);
   readyForDisplay=B();
}

VV<ColRef> Img24::toVVColRef()const
{
   VV<ColRef> res(height,width);
   for (Z i=1;i<=height;i++){
      for (Z j=1;j<=width;j++){
         res[i][j]=bh_[i-1][j-1];
      }
   }
```

```
   return res;
}

VVV<L> Img24::toVVV()const
{
   VVV<L> res(3,height,width);
   for (Z i=1;i<=height;i++){
      for (Z j=1;j<=width;j++){
         rgb c=getPel(j-1,i-1);
         res[1][i][j]=(L)c.r;
         res[2][i][j]=(L)c.g;
         res[3][i][j]=(L)c.b;
     }
   }
   return res;
}

VV<R> Img24::toVVR(R wR, R wG, R wB, Iv val)const
{
   VV<R> res(height,width);
   R wInv=cpminv((wR+wG+wB)*255);
   R val1=val.inf();
   R val2=val.sup();
   R dval=val2-val1;
   for (Z i=1;i<=height;i++){
      for (Z j=1;j<=width;j++){
         rgb c=getPel(j-1,i-1);
         R vR=R(c.r);
         R vG=R(c.g);
         R vB=R(c.b);
         R v=wR*vR+wG*vG+wB*vB;
         v*=wInv;
         if (val1==0. && val2==1.)
            res[i][j]=v;
         else
            res[i][j]=val1+dval*v;
     }
   }
   return res;
}

Img24::Img24(const Word& filename):c0(),sty(SET_COLOR),gamma(1)
{
   Z mL=1;
   Word loc("Img24(Word)");
   cpmmessage(mL,loc&" started");
   IFileStream in(filename);
   if (!in()){
      cpmerror("Img24(Word): bad stream opening "&filename);
      return;
```

```
}
string w1;
string indicator1("P3");
string indicator2("P6");
in()>>w1;
cpmmessage(loc&": file type read as "&Word(w1));
in.skipComments();
   // several programs that create ppm-files introduce comment-lines
   // which have to be overread
if (w1!=indicator1 && w1!=indicator2){
   ostringstream ost;
   ost<<"Img24("<<filename<<
      "): image file is not of type ppm P6 or P3";
   ost<<" instead we read "<<w1;
   cpmerror(Word(ost.str()));
}
in()>>width;
in.skipComments();
in()>>height;
in.skipComments();
Z depth;
in()>>depth;
in.skipComments();
if (depth!=255){
   ostringstream ost;
   ost<<"Img24("<<filename<<"): image file has depth="<<depth;
   cpmerror(Word(ost.str()));
}
bh_=Vl<Vl<ColRef> >(height,Vl<ColRef>(width));

if (w1==indicator2) in().unsetf(std::ios_base::skipws);
   // P6
if (w1==indicator1){ // P3, non-binary reading
   unsigned short r,g,b;
   for (Z i=0;i<height;i++){
      for (Z j=0;j<width;j++){
         in()>>r;
         in()>>g;
         in()>>b;
         bh_[i][j]=rgb(r,g,b);
      }
   }
}
else{ // P6
   L r,g,b;
   for (Z i=0;i<height;i++){
      for (Z j=0;j<width;j++){
         in()>>r;
         in()>>g;
         in()>>b;
```

```
            bh_[i][j]=rgb(r,g,b);
         }
      }
   }
   cpmmessage(mL,loc&" done");
}

void Img24::insert(xy p, Img24 const& img , ColRef cr)
// fast insert of image lines as whole
{
   Z mL=3;
   Word loc("Img24::insert(xy,Img24)");
   CPM_MA
   xy tp(0,0);
   xyxy tr(tp,width,height);
   Z iw=img.w(), ih=img.h();
   xyxy ir(p,iw,ih);
   ir=tr&ir; // set section, now ir fits the frame
      // of *this or it is void
   if (ir.isVoid()&& cpmverbose>1){
      cpmwarning(loc&": nothing to insert");
      return;
   }
   iw=ir.w();ih=ir.h();
   for (Z j=0;j<ih;++j){
      horIns(p,img[j],cr);
      p.y++;
   }
   readyForDisplay=B();
   CPM_MZ
}

void Img24::display(Z sx, Z sy)
// that this is fast is important since Frame::display
// is based on it
{
   Z mL=3;
   static Word loc("Img24::display(Z,Z)");
   CPM_MA
   if (!readyForDisplay){
      displayBuffer=Rec(bh_);
      Z dw=displayBuffer.w();
      Z dh=displayBuffer.h();
      Z excess_x=sx+dw-Viewport::win().w();
      Z excess_y=sy+dh-Viewport::win().h();
      if (excess_x>0){
         sx-=excess_x;
         if (sx<0) sx=0;
      }
      if (excess_y>0){
```

```
         sy-=excess_y;
         if (sy<0) sy=0;
      }
         // if the properly shifted image does not fit
         // Viewport::win() we reduce the shift to zero, so that
         // as much as possible can be seen
      xy shift(sx,sy);
      displayBuffer+=shift;
      readyForDisplay=B(true);
   }
   displayBuffer.dis();
   CPM_MZ
}

Img24::Img24(Rec const& rec):width(rec.w()),height(rec.h()),
   sty(SET_COLOR),gamma(1)
{
   xy p1=rec[1];
   displayBuffer=rec;
   displayBuffer-=p1;
   bh_=rec.fold();
   readyForDisplay=B(true);
}

Word Img24::write(const Word& filename, bool binary)
{
   Z mL=3;
   Word loc("Img24::write(Word,bool)");
   cpmmessage(loc&" started");
   Word ffn=filename&".ppm";
   // writing a portable pixmap; very simple
   const Z maxVal=255;
   cpmmessage(mL,"width="&cpmwrite(width)&", height="&cpmwrite(height));
   if (binary){
      OFileStream out(ffn);
      out()<<"P6"<<'\n'<<width<<' '<<height<<'\n'<<maxVal<<'\n';
      for (Z i=0; i<height;i++){
         for (Z j=0;j<width;j++){
            ColRef val=bh_[i][j];
            out()<<val.r<<val.g<<val.b;
         }
      }
   }
   else{ // not binary, explicit "publication mode"
       const string indicator="P3";
      // indicator for ASCII portable pixmap
      Word title="# Portable pix map made by CpmGraphics::Img24:write(";
      title&=filename;
      title&=") ";
      CpmTime::Greg greg;
```

```
      greg.now();
      Word date="# Date: ";
      date&=greg.getDate(false);
      Word textOut="#";
      bool safe=true;
      bool binary=false;
      OFileStream out(ffn,safe,binary);
      out()<<indicator<<endl;
      out()<<title<<endl;
      out()<<date<<endl;
      out()<<"# Text:"<<endl;
      out()<<textOut<<endl;
      out()<<"# Image data: width, height, maximum code value"<<endl;
      out()<<width<<" "<<height<<endl;
      out()<<maxVal<<endl;
      Z countMax=5;
      Z count=0;
      for (Z i=0; i<height;i++){
         for (Z j=0;j<width;j++){
            ColRef val=bh_[i][j];
            out()<<(Z)val.r<<" "<<(Z)val.g<<" "<<(Z)val.b<<"   ";
            count++;
            if (count==countMax){
               count=0;
               out()<<endl;
            }
         }
      }
      out()<<endl;
   }
   cpmmessage(loc&" done");
   return ffn;
}

bool Img24::clip(R& ax, R& ay, R& bx, R& by)const
// Cohen-Sutherland algorithm,modified from two internet sources
{
   static L X_TOO_SMALL=8;
   static L X_TOO_LARGE=4;
   static L Y_TOO_SMALL=2;
   static L Y_TOO_LARGE=1;
   L c1 = outcode( ax, ay);
   L c2 = outcode( bx, by);
   L c;
   R minusDeltaY, deltaX, d, x, y;
   while((c1 | c2) != 0){
      if ((c1 & c2) != 0) return false ;
      minusDeltaY = ay-by;
      deltaX = bx-ax;
      d = ax*by-ay*bx;
```

```
      if (c1 == 0)
         c = c2;
      else
         c = c1;
      if ((c & X_TOO_SMALL) != 0) {
         x = 0;
         y = -(minusDeltaY * x + d) / deltaX;
      }
      else if ((c & X_TOO_LARGE) != 0) {    //c ist rechts
         x = width;
         y = -(minusDeltaY * x + d) / deltaX;
      }
      else if ((c & Y_TOO_LARGE) != 0) {      //c ist oben
         y = height;
         x = -(deltaX*y + d) / minusDeltaY;
      }
      else if ((c & Y_TOO_SMALL) != 0) {   //c ist unten
         y = 0;
         x = -(deltaX * y + d)/minusDeltaY;
      }
      if (c == c1) {
         ax = x;
         ay = y;
         c1 = outcode(ax, ay);
      }
      else {
         bx = x;
         by = y;
         c2 = outcode(bx, by);
      }
   }
   return true;
}

void Img24::lineThin(R x1, R y1, R x2, R y2, rgb c)
{
 //  Z mL=3;
   static Word loc("Img24::lineThin(R,R,R,R,rgb)");
 //  CPM_MA
   if (!CpmRoot::isVal(x1)) return;
   if (!CpmRoot::isVal(y1)) return;
   if (!CpmRoot::isVal(x2)) return;
   if (!CpmRoot::isVal(y2)) return;
      // input may be irregular; then it is OK to do nothing
   static R tiny=0.001;
   bool b=clip(x1,y1,x2,y2);
   if (!b) return;

   ColSty styMem=sty;
   if (antiAliasLines) sty=MIX_COLOR;
```

```
Z i,j,i1=disX(x1), i2=disX(x2), j1=disY(y1), j2=disY(y2);
   // x1,x2,y1,y2 no longer needed
if (i1==i2 && j1==j2){
   bh_[j1][i1]=c;
}
else{
   Z pdi=i2>=i1 ? i2-i1 : i1-i2;
   Z pdj=j2>=j1 ? j2-j1 : j1-j2;
   if (pdj<=pdi){ // flat case. Since at least one of pdi, pdj is
      // !=0, we know pdi>0, thus i1!=i2. We want to work over
      // increasing i, this can be achieved by exchanging the
      // role of 1 and 2:
      if (i2<i1){
         cpmswp<Z>(i1,i2);
         cpmswp<Z>(j1,j2);
      } // no 'else' needed
      // now a loop 'for (i=i1;i<=i2;++i)' is OK
      R m=R(j2-j1)/(i2-i1);
         // the definition m=(y2-y1)/(x2-x1) gives bad curves
      R yC=antiAliasLines ? j1+0.5 : j1+tiny;
      for (i=i1;i<=i2;++i){
         Z jC=disY(yC);
         if (antiAliasLines){
            R frac=yC-jC-0.5; // ranges in [-0.5,0.5)
          //  cpmassert(frac>=-0.5 && frac<0.5,loc); //tested to hold
            Z jC1= frac>=0 ? disY(yC+1) : disY(yC-1);
            if (frac<0) frac*=-1;
            putPel(i,jC,c,1-frac);
            putPel(i,jC1,c,frac);

         }
         else{
            if (sty==SET_COLOR) bh_[jC][i]=c; else putPel(i,jC,c);
         // fast access
         }
         yC+=m;
      } // i-loop
   } // flat case
   else{ // steep case
      if (j2<j1){
         cpmswp<Z>(i1,i2);
         cpmswp<Z>(j1,j2);
      } // no 'else' needed
      // now a loop 'for (j=j1;j<=j2;++j)' is OK
      R m=R(i2-i1)/(j2-j1);
      R xC=antiAliasLines ? i1+0.5 : i1+tiny;
      for (j=j1;j<=j2;++j){
         Z iC=disX(xC);
         if (antiAliasLines){
```

```
               R frac=xC-iC-0.5;
              // cpmassert(frac>=-0.5 && frac<0.5,loc);//tested to hold
               Z iC1 = frac>=0 ? disX(xC+1) : disX(xC-1);
               if (frac<0) frac*=-1;
               putPel(iC1,j,c,frac);
               putPel(iC,j,c,1-frac);
            }
            else{
              if (sty==SET_COLOR) bh_[j][iC]=c; else putPel(iC,j,c);
            }
            xC+=m;
         } // j-loop
      } // steep case
   } // if (i1==i2 && j1==j2){...}else{...}
   if (antiAliasLines) sty=styMem;
   readyForDisplay=B();
 //  CPM_MZ
}

namespace{

ColRef fStereo(ColRef const& cB, ColRef const& c){ return cB.add(c);}

} // namespace

bool Img24::horLine(Z x1, Z x2, Z y, rgb c, R w)
{
  // return;
   if (!i2().hasElm(y)) return false;
      // nothing to do
   IvZ iv=i1()&IvZ(x1,x2);
      // restriction of line to width
   ColRef cr=c;
   if (sty!=STEREO){
      F<ColRef,ColRef> fc=F4<ColRef,ColRef,ColSty,R,R,ColRef>
         (cr,sty,gamma,w)(miX);
      bh_[y].fa_(fc,iv);
   }
   else{ // should be fast, so we don't hand-over arguments
      // that are not needed
      F<ColRef,ColRef> fc=F1<ColRef,ColRef,ColRef>(cr)(fStereo);
      bh_[y].fa_(fc,iv);
   }
   readyForDisplay=B();
   return true;
}

namespace{

ColRef fSet(ColRef const& cB, ColRef const& c)
```

```
{
   return c;
}

ColRef cWrite;
   // notice that we can't make cWrite a part
   // of the argument list of fSet2, so we pass
   // it 'over the head of this function by initializing
   // it in the function body of horIns; very
   // powerful trick which looks quite acceptable to me
   // when encasulated in a namespace as done here.
ColRef fSet2(ColRef const& cB, ColRef const& c)
{
   static R i255=1./255;
   if (c.r<255){ // c is assumed to come from
      // a monochrome line to be inserted
      R w1=i255*c.r;
      R w2=1-w1;
      return ColRef((Z)(w1*cB.r+w2*cWrite.r),
                    (Z)(w1*cB.g+w2*cWrite.g),
                    (Z)(w1*cB.b+w2*cWrite.b));
   }
   else return cB;
}

}// namespace

void Img24::horIns(xy p, Vl<ColRef> const& line, ColRef cr)
// isn't this elegant ?
// safty features added 2005-09-12
// Argument cr added 2005-09-15
{
   if (p.y<0 || p.y>=height || p.x>=width) return;
      // readyForDisplay needs no change
   Z ld=line.dim();
   if (ld==0) return;
   IvZ rl=IvZ(p.x,p.x+ld-1); // range of line
   IvZ i=i1()&rl;
   if (i.isVoid()) return;
      // end of trivial returns
   cWrite=cr;
   if (cr==ColRef()){ // use fSet
      if (p.x>=0)
         bh_[p.y].ff_(fSet,line,p.x);
         // even if the line is too long for the image width
         // no cutting is needed
      else // line manipulation only if needed
         bh_[p.y].ff_(fSet,line.eliFirst(-p.x),0);
   }
   else{ // use fSet2
```

```
      if (p.x>=0)
         bh_[p.y].ff_(fSet2,line,p.x);
      else
         bh_[p.y].ff_(fSet2,line.eliFirst(-p.x),0);
   }
   readyForDisplay=B();
}

void Img24::clear(void)
{
   Z mL=3;
   Word loc("Img24::clear(void)");
   CPM_MA
   if(width && height){
      bh_=Vl<Vl<ColRef> >(height,Vl<ColRef>(width,c0));
   }
   if (c0.isGray()){
      L gb=c0.r;
      xyxy r(xy(0,0),width,height);
      displayBuffer=Rec(r,gb);
      readyForDisplay=B(true);
   }
   else readyForDisplay=B();
   CPM_MZ
}

void Img24::fillRectangle(R x1, R y1, R x2, R y2, rgb c)
{
   CpmRootX::order<R>(x1,x2);
   CpmRootX::order<R>(y1,y2);
   Z zx1=(Z)(x1);
   Z zx2=(Z)(x2);
   Z zy1=(Z)(y1);
   Z zy2=(Z)(y2);
   for (Z j=zy1;j<=zy2;j++) horLine(zx1,zx2,j,c);
}

void Img24::fill(TriAng const& tri, rgb c)
{
   IvZ iv1=tri.i1();
   iv1=iv1&i1();
   if (iv1.isVoid()) return;
   IvZ iv2=tri.i2();
   iv2=iv2&i2();
   if (iv2.isVoid()) return;
   R shift=0.0;
   for (Z i=iv1.b();i<=iv1.e();++i){
      for (Z j=iv2.b();j<=iv2.e();++j){
         R2 p(shift+i,shift+j);
         bool out=tri.out(p);
```

```
         if (!out){
            putPel(i,j,c);
         }
      }
   }
}

void Img24::fillEllipse(R x1, R y1, R x2, R y2, rgb c)
// one should at least mark the center, that very small ellipses don't
// rely on numerical noise for getting shown
{
   CpmRootX::order<R>(x1,x2);
   CpmRootX::order<R>(y1,y2);
   xy z1=dis(x1,y1), z2=dis(x2,y2);
   Z zx1=z1.x, zy1=z1.y, zx2=z2.x, zy2=z2.y;

   if ((IvZ(zx1,zx2)&i1()).isVoid()) return;
   if ((IvZ(zy1,zy2)&i2()).isVoid()) return;

   R w2=(x2-x1)*0.5;
   R w22=w2*w2;
   R h2=(y2-y1)*0.5;
   R h22inv=1.;
   if (y1!=y2){
      R h22=h2*h2;
      h22inv=1./h22;
   }
   R x0=w2+x1;
   R y0=h2+y1;

   bool marked=false;

   for(Z j=zy1; j<=zy2; j++){
      R rz=j-y0;
      R rz2=rz*rz*h22inv;
      R wj=(1-rz2)*w22;
      if (wj<0) continue;
      wj=sqrt(wj);
      Z iL=(Z)(x0-wj);
      Z iU=(Z)(x0+wj);
      bool b=horLine(iL,iU,j,c);
      if (b==true) marked=true;
  }
   if (!marked){
      R xm=(x1+x2)*0.5, ym=(y1+y2)*0.5;
      putPel(xm,ym,c); // these 2 lines added 2006-11-09
   }
}

bool Img24::prnOn(ostream& str)const
```

```
{
   cpmwt("Img24 begin");
   cpmp(width);
   cpmp(height);
   cpmp(bh_);
   cpmp(c0);
   cpmp((Z)sty);
   cpmp(antiAliasLines);
   cpmp(gamma);
   cpmwt("Img24 end");
   return true;
}

bool Img24::scanFrom(istream& str)
{
   Z mL=3;
   Word loc("Img24::scanFrom(istream)");
   CPM_MA
   readyForDisplay=B();
      // since we read only the primary image data
      // and not the derived data displayBuffer
   cpms(width);
   cpmmessage(mL,"reading width done");
   cpms(height);
   cpmmessage(mL,"reading height done");
   cpms(bh_);
   cpmmessage(mL,"reading bh_ done");
   cpms(c0);
   cpmmessage(mL,"reading c0 done");
   Z styAux;
   cpms(styAux);
   cpmmessage(mL,"reading styAux done");
   sty=(ColSty)styAux;
   cpms(antiAliasLines);
   cpmmessage(mL,"reading antiAliasLines done");
   cpms(gamma);
   cpmmessage(mL,"reading gamma done");
   CPM_MZ
   return true;
}

ColRef Img24::biLin(ColRef f11, ColRef f12, ColRef f21, ColRef f22,
   R w11, R w12, R w21, R w22, Z method)
{
   if (method==1){
      R p11=w11*w21;
      R p12=w11*w22;
      R p21=w12*w21;
      R p22=w12*w22;
      Z r=(Z)(f11.r*p11 + f12.r*p12 + f21.r*p21 + f22.r*p22);
```

```
      Z g=(Z)(f11.g*p11 + f12.g*p12 + f21.g*p21 + f22.g*p22);
      Z b=(Z)(f11.b*p11 + f12.b*p12 + f21.b*p21 + f22.b*p22);
      return ColRef(r,g,b);
   }
   else{
      Z i= w11<=w12 ? 2 : 1;
      Z j= w21<=w22 ? 2 : 1;
      if (i==1 && j==1) return f11;
      else if (i==1 && j==2) return f12;
      else if (i==2 && j==1) return f21;
      else return f22;
   }
}

Img24 Img24::intPolAux(Z nI, Z mI, Z method)const
   // See function int2
{
   Z mL=3;
   Word loc("Img24::intPolAux(Z,Z,Z)");
   CPM_MA
   Z m=h();
   Z n=w();
   if (m==mI && n==nI){
      cpmmessage(mL,loc&": no size change, nothing to do");
      return *this;
   }
   Z i,j;
   ColRef f11,f12,f21,f22;
   Iv iv(0,1);
   V<R> ci=iv.centers(m);
   V<R> cj=iv.centers(n);
   V<R> cIi=iv.centers(mI);
   V<R> cIj=iv.centers(nI);

   Img24 res(nI,mI);
   for (i=1;i<=mI;i++){
      R xiI=cIi[i];
      X2<R2,Z2> qi=iv.intPolInfo(xiI,m);
      R2 wi=qi.c1();
      Z2 zi=qi.c2();
      R wi1=wi.x1;
      R wi2=wi.x2;
      Z i1=zi.x1-1;
      Z i2=zi.x2-1;
      for (j=1;j<=nI;j++){
         R xjI=cIj[j];
         X2<R2,Z2> qj=iv.intPolInfo(xjI,n);
         R2 wj=qj.c1();
         Z2 zj=qj.c2();
         R wj1=wj.x1;
```

```
         R wj2=wj.x2;
         Z j1=zj.x1-1;
         Z j2=zj.x2-1;
         f11=bh_[i1][j1];
         f12=bh_[i1][j2];
         f21=bh_[i2][j1];
         f22=bh_[i2][j2];
         res.bh_[i-1][j-1]=biLin(f11,f12,f21,f22,wi1,wi2,wj1,wj2,method);
      }
   }
   CPM_MZ
   return res;
}

Img24 Img24::interpolate(Z nI, Z mI, Z method)const
   // See function int2
{
   Z mL=3;
   Word loc("Img24::interpolate(Z,Z,Z)");
   CPM_MA
   Z m=h();
   Z n=w();
   if (mI>m/2 && nI>n/2){ // traditional case, interpolation to
      // similar or larger resolution
      cpmmessage(mL,loc&" no averaging requested");
      return Img24::intPolAux(nI,mI,method);
   }
   else{ // new case: interpolating to lower resolution
         // based on averaging; works well for characters
         // in fonts
      R mIinv=1./mI, nIinv=1./nI;
      Z mf=cpmround(m*mIinv), nf=cpmround(n*nIinv);
      Z mA=mI*mf, nA=nI*nf;
      Img24 resA=Img24::intPolAux(nA,mA,method);
         // image of similiar resolution than the original
         // but allowing schematic averaging
      Img24 res(nI,mI);
      Z i,j,iR,jR,k,l;
      R nmInv=1./(mf*nf);
      for (i=1;i<=mI;++i){
         iR=(i-1)*mf;
         for (j=1;j<=nI;++j){
            R3 sum;
            jR=(j-1)*nf;
            for (k=0;k<mf;++k){
               for (l=0;l<nf;++l){
                  ColRef val=resA[iR+k][jR+l];
                  sum.x1+=val.r;
                  sum.x2+=val.g;
                  sum.x3+=val.b;
```

```
                }
            }
            sum*=nmInv;
            Z s1=cpmround(sum.x1);
            Z s2=cpmround(sum.x2);
            Z s3=cpmround(sum.x3);
            (res.bh_)[i-1][j-1]=ColRef(s1,s2,s3);
         }
      }
      cpmmessage(mL,loc&" averaging done");
      CPM_MZ
      return res;
   }
}

Img24 Img24::pair(Img24 const& img, Z gap)const
{
   Z mL=3;
   Word loc("Img24::pair(Img24)");
   CPM_MA
   Z hn=h();
   cpmassert(hn==img.h(),loc);
   Z wn=w()+img.w()+gap;
   Img24 res(wn,hn);
   xy p1(0,0);
   xy p2(w()+gap,0);
   res.insert(p1,*this);
   res.insert(p2,img);
   CPM_MZ
   return res;
}

} // namespace
```

100 cpminifilebasapp.h

```
//? cpminifilebasapp.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_INIFILEBASAPP_H
#define CPM_INIFILEBASAPP_H

/*

Purpose: Provide tools for building application programs

   2007-08-07 Usage of path-names changed. Cooperation with Ruby
      programs suggested to use valid directory names x (for which
      Dir.chdir(x) works in situations were the present program had
      path names ending in '/'. Now the harmonization is done:
      pathname = directory name.
      This caused major changes in ini-files.

*/
#include <cpmf.h>
#include <cpmp.h>
#include <cpmapplication.h>
#include <cpmrecordhandler.h>

namespace CpmApplication{
   // Building an application framework within which one has
   // convenient graphical and messaging capabilities available.

   using CpmRoot::Word;
   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRootX::RecordHandler;
   using CpmArrays::V;
   using CpmArrays::Vp;
   using CpmFunctions::F;
   using CpmFunctions::F1;

   extern bool docu;
      // documentation flag

   extern Z dimApp;
      // 2 or 3 represents a dimension selection

////////////////////////  IniFileBasedApplication  ///////////////////////
// Proven tool to for creating non-interactive applications which
// get all input from files and create output to files (probably along
// with dynamical screen output) including proper run documentation.
//
```

```
// Derive your application class from IniFileBasedApplication and
// re-define the virtual function void doTheWork(void). Primary data
// input is provided by a text file capable of access based on sections
// and keys.
//
// Supported data types for input are bool, Z, R, Word and homotypic
// lists of these.
//
// Originally this file had to be a Windows configuration file which
// always come with the file extension 'ini' which explains that I
// still refer to this primary input file as 'the ini-file'. The
// ini-file is the unique source to all data that the class can
// hold and which determine the 'state' of a class instance.
// In technical terms it is the attribute
//
//    RecordHandler rch;
//
// which holds the state, that was created by reading the ini-file. Here
// RecordHandler is a flexible container class and RecordHandler itself
// is a direct descendant of a container class Record. Notice,
// that the state information may contain file names or URLs. These then
// open access to additional information. And also this information may
// enter the computations organized as member functions of the class.
//
// The syntax of ini-files will be explained in cpmrecord.h together with
// class Record, see particularly the explanation to the default
// constructor
//
//      Record(){};
//
// This is natural since this syntax can be considered as being
// indirectly defined in terms of a member function of this class:
// a text file named fn is a valid ini-file if and only if
//       iniFileTest(fn)==true
// for the following function definition:
//
/*
      bool iniFileTest(Word const& aFile)
      {
         IFileStream ofs(aFile);
         Record rc;
         return rc.readFormatted(ofs());
      }
*/
//
// According to my experience over years the 'ini-files' are very
// convenient to work with in programs that are subject to continuous
// improvement and adaption to new requirements.
// Since August 2003 there is the possibility to organize ini-files
// hierarchically: Any in-file may contain enties of the form
```

```
// F anIniFile
// This helps to keep the input for complex programs organized,
// even if these need thousands of input parameters (which would
// be a challenge for GUI-based programs).

// A convenient method for creating a scientific program (application)
// is to derive a class from the class IniFileBasedApplication and to
// re-define the  virtual function doTheWork in a way that unfolds and
// utilizes the information given in rch. The actual execution of the core
// part of the program (the raison d'être of the program) is then
// initiated by calling the function run() of this class. This  function
// is not affected by the derivation - it is already defined in the
// present class .
// In a way it is indirectly virtual since it calls the virtual
// function doTheWork(). This is a common pattern in non-trivial C++
// programs.

Z setDimApp(Word const& prog, Word const& path="");
// sets dimApp from the arguments that can also be used to create an
// application. The reason is that one sometimes needs the 'mode'
// parameter from the ini file in order to start the right application.
// returns the value of 'selection.mode' of the ini-file and
// sets dimApp equal to this value if it is 2 or 3

class IniFileBasedApplication{
//creating file-based applications which document their input and output

protected:

   Z size;
   Z rank;
      // to support parallelism

 //  Word path_;
      // glue(path_,programName&".ini") is assumed to be a file that
      // can be opened to read in the data defining the program.

   RecordHandler rch;
      // The only non-trivial constructor of the present class
      // reads data from a file that is given by providing
      // its name and (optionally path) as argument. These data
      // load the storage area of data element rch which is capable
      // of holding data of types B,Z1,R1,Word,V<B>,V<Z1>,V<R1>,V<Word>
      // and provides bidirectional access to data of the related types
      // bool,Z,R,string,V<bool>,V<Z>,V<R>,V<string>.

      // The value (state) of rch is the essential determinant of
      // any meaningful instantiation of the class.

      // Due to the polytypic nature of the data type RecordHandler
```

```
      // rch can easily hold all information needed to determine the
      // run of an arbitrarily complex program.

      // So, as already stated, a proven design pattern for a scientific
      // application is to derive an application class from the present
      // class and re-define the virtual function doTheWork in a way that
      // unfolds and utilizes the information given in rch.

    Word programTitle_;
      // Typically a descriptive name for the fuctionality of the
      // present instance of the class. A whole 'program' such as
      // my 'PaLa' combines several classes derived from
      // class IniFileBasedApplication and is referred to by the
      // title held in CpmApplication::title_.
      // The present attribute does not get initialized by calling a
      // constructor but using member function setTitle(Word).

// dependent data: values will be initialized during calls to the
// constructor or to function run() from information in rch.

   V<Word> inFiles;
      // list of files that define input values

   V<Word> outFiles;
      // list of files that are created on output
      // The names of the files that actually get created have runName
      // appended

// From here on we follow the good practice to alienate the names by an
// underscore in order to not unintentionally use the same name in
// programming member functions. The previous 5 members occur so often
// in existing code that it would be a major activity to change the
// names.

   Z sel_;
      // a selector for the 'depth' of auto-generated documentation
      // files.
      // Control of auto-generated documentation files is based on the
      // three entries sel_, mode_, runName_. These are the rules:
      // 1. for sel_<0 no documentation files will be created.
      // 2. for sel_==0 a light documentation file is created that
      //    covers all the input so that the program run can be repeated.
      // 3. for sel_>0, in addition to the light documentation file
      //    a more detailed one will be created that also can made to
      //    list computational results.
      // 4. All documentation files have names which contain runName_
      //    appended to the name of the calling program
      // 5. Also the parameter mode_ will be appended to the
      //    names of the documentation files if sel_>0
```

```
   Z mode_;
      // allows the application run in different modes, which can be
      // discerned by looking to their runname. The model for this
      // parameter is a dimension parameter which switches in a modeling
      // program from 2D to 3D modeling.

   Word runName_;
      // identifier for the program run that will be appended to
      // the names of documentation files.
      // If the value of this quantity is "auto" this value will
      // be replaced by an autogenerated name.

// end of documentation control

   Word programName_;
      // a name of the program, here only a tag. We assume that a
      // ini-file exists the name of which is
      // path&dirSep&programName_&".ini"

   V<Z> appendInFiles_;
      // a list with as many entries as inFiles; controls
      // whether an inputfile will be appended to documentation file
      // programName&"-"&runName&"_docu.txt"
      // 0 for not append, 1 for append
      // For most inputfiles, documentation is recommended. For programs
      // that simply view large moviefiles (which in this application are
      // inputfiles) it is essential however to be in a position to
      // disable the time-consuming copying of the input file

   V<Z> appendOutFiles_;
      // a list with as many entries as outFiles; controls
      // whether an outputfile - which will be created anyway -
      // additionally will be appended to file
      // programName&"-"&runName&"_docu.txt"
      // 0 for not append, 1 for append

   R tStart_;
      // time as given by cpmtime() at program start

   R tStop_;
       // time as given by cpmtime() at program stop

   R tWaitAnnounceScreen_;
      // time for reading the introduction screen

   R tWaitWorkScreen_;
      // wait time after doTheWork. Typically available for
      // ispecting the last screen produced by
      // function doTheWork.
      // For this to work one needs to have a statement
```

```
      //    cpmwait(tWaitWorkScreen_);
      // after the last screen display statement of doTheWork().

   void documentTheWork();// not const !
      // could, for instance, change runName by appending a code for the
      // execution time

   virtual void announceTheWork();
      // typically a start screen

   virtual void doTheWork(void) // was const till 2001-10-03
       // re-defining this function in a derived class
       // defines the specific behavior of the application
       // thus defined.
   { cpmwarning(
      "IniFileBasedApplication: trivial version of doTheWork() called");
   }

public:
   typedef IniFileBasedApplication Type;

   virtual IniFileBasedApplication* clone()const
   { return new IniFileBasedApplication(*this);}

   IniFileBasedApplication(void):mode_(0),sel_(0){}
      // notice that most data have default constructor

   explicit IniFileBasedApplication(Word const& prog);
      // Creates an instance of IniFileBasedApplication from data
      // that are obtained by reading the file named
      // Message::ext(prog&".ini")=: fn which will be referred to as the
      // ini-file of the application.
      // Finally invokes Record::readFromFile(fn,true);
      // and this, in turn, invokes Record::readFormatted(istream& str)
      // with str obtained by opening file fn. So the
      // function Record::readFormatted(istream& str) finally determines
      // the syntax to be followed by the ini-file.

      // Example of the structure of this file
/****************** example for ini file *******************************
// text for explanation

about
   W file=...
   Ws line1=What is done here
   Ws line2=...
   Ws line3=...

selection
   Z sel=1
```

```
   Z cpmverbose=1
   Z mpiverbose=0
   Z cpmdbg=1
   Z mode=2
   W runName=021211a
   ...
input
   Z numberOfFiles=3
   W files1=movie1_station_xx.txt
   // irrelevant
   W files2=movie1_station_xx.txt
   // irrelevant
   W files3=movie3_station_xx.txt
   // irrelevant
   Z doc1=0
   Z doc2=0
   Z doc3=0
output
   Z numberOfFiles=3
   W files1=movie1
   W files2=movie2
   W files3=movie3
   Z doc1=0
   Z doc2=0
   Z doc3=0
   ...
****************************** end of file example *****************/

   void setTitle(Word const& w){ programTitle_=w;}
      // see explanation to data member programTitle_
   Word getTitle()const{ return programTitle_;}
      // see explanation to data member programTitle_
   Word getRunName()const{ return runName_;}
   Word getProgramName()const{ return programName_;}
   Z nInFiles()const{ return inFiles.dim();}
   Z nOutFiles()const{ return outFiles.dim();}
   void setModeFromInFiles();
      // In some programs based on IniFileBasedApplication the inFiles
      // are assumed to be the outFiles of other programs also based
      // on IniFileBasedApplication. These outfiles have the mode
      // parameter normally appended, and thus allow the mode-value
      // which fits the nature of these files to be read from
      // the file names of the inFiles under consideration. This
      // mechanism (which changes the values of mode and runName)
      // will be executed by calling the present function

   void run(void);
      // runs the application and documents input and output

   void sizeAndRank(){
```

```
      size=CpmMPI::Cpm_com.getSize();
      rank=CpmMPI::Cpm_com.getRank();
   }
};

//////////////////////////// class Task ////////////////////////////////////

class Task{ // tool for implementing a menu-like application structure
   // A class that makes it easy to define function main_() of a program
   // so that it allows a menu-like structure
   Word name_;
      // device that allows to call f_ as an autonomous
      // action within class Task
   F<Word,Z> f_;
      // a representation of potentially extreme complicated
      // code that is to be executed in performing the task
 public:
   Word nameOf()const{ return Word("Task");}
   Task(){}
      // does nothing
   Task(Word const& name, F<Word,Z> const& f):name_(name),f_(f){}
      // creates a task from the attributes
   Task(Word const& name, Z (*f)(Word const&) ):name_(name),f_(f){}
      // creates a task from the attributes
   Z work()const{ return f_(name_);}
      // executes the task, for which the constructors
      // provided the code
   F<Word,Z> get_f()const{ return f_;}
      // helps to use proven earlier code that works with
      // the functions instead of the full data of a Task
      // (see implementation of comAsRoot)
   static Task subMen(Word const& name, F<Word,Z> const& f);
      // making a menu by providing a list of ini-files for
      // the code provided by function f. The list of ini-files
      // is assumed to be defined in a file name&".ini". This
      // will be searched in directories ../control, ./control, ./,
      // (tested in this order) relative to the execution directory ./
      // The structure of the expected ini-file is as in the following
      // example:
/*
// file pala2.ini
// Particle system enclosed by walls

topics
   Z begin=1
   Z end=15
      // selecting the first and the last index of the
      // ini-file that will be done one after the other
      // (which is particularly convenient for demo programs)
```

```
   Ws topics01=danceOfGrains  danceOfRubberGrains  magneticChains
   Ws topics02=vertical_momentum  thermalization  allInteractions
   Ws topics03=allInteractionsEtot  larmor_cycles  plasma
   Ws topics04=crystallization  crystallization2  brownian_motion
   Ws topics05=osmosis1  osmosis2  FaradayCage
   Ws topics06=top2Dev
      // a list of names of ini-files that can easily be extended
      // and for which the access index (starting with 1) is obvious
      // from formatting if one chooses to place always the same number
      // of names in a line
      // (The maximum number of topics is 16 so far)

path
   W path=./control
   // path saying where the ini-files danceOfGrains.ini,
   // danceOfRubberGrains.ini, ... the names of which are given
   // in topics1, topics2, ... will be taken from

control

   R tWait=2
      // wait time after
*/
   static Task subMen(Word const& name, Z (*f)(Word const&) )
      { return subMen(name, F<Word,Z>(f));}
      // see previous function
   static Task men(Word const& name, V<Task> const& tasks);
      // Creates a root-level menu from a ini-file that must have the
      // structure of a commandline as in the following example
/*
// top.ini
command_line
   // obligatory header

// 1: Rigid body dynamics
// 2: Particle system with walls
// 3: Movie viewer

Ws args=2 2 2 ./control/ 4
   // after the '=' write just as you would write on the command line
   // after the name of the program to be called.
   // e.g. 3 2 10 c:/pala/control/ 15
   // where
   // 3  is the subprogram selector,
   // 2  is the value of cpmverbose until it becomes reset by
   //    additional input from a ini-file
   // 10 is the value of tWait, the time to see the final graphics screen
   //    in seconds.
   // c:/pala/control/ is a location descriptor where to find
   //    top1.ini, top2.ini, ...    that should be used
```

```
   // after the '=' write just as you would write on the command line
   // after the name of the program to be called.
   // e.g. 3 2 10 c:/pala/control/ 15
   // where
   // 3  is the subprogram selector,
   // 2  is the value of cpmverbose until it becomes reset by
   //    additional input from a ini-file
   // 10 is the value of tWait, the time to see the final graphics screen
   //    in seconds.
   // c:/pala/control/ is a location descriptor where to find
   //    top1.ini, top2.ini, ...    that should be used
   // 15 is a further wait time, now for the introduction screen
   //    that a user may want to read accurately for the first time
   //    but that shine up only for very few seconds in routine usage.
*/
};

/*
// Typical application (assuming
// Z top1(Word const& path),
// Z top2(Word const& iniFile),
// Z top3(Word const& path)   )

Z main_()
{
// we define tasks and combine them
   Task top2=Task::subMen("top2",top2c);
   Z nT=3;
   V< Task > vTask(nT);
   vTask[1]=Task("top1",top1);
   vTask[2]=top2;
   vTask[3]=Task("top3",top3);
   Task rt=Task::men("top",vTask);
// finally we execute them (by executing their
// combined version)
   return rt.work();
}

as an example for the complex code imported by the
functions topi I give top2:

Z top2(Word const& iniFile)
{
   using namespace CpmApplication;
   SysControl();
   setDimApp(iniFile);
   if (dimApp==2){
      CpmTop2::Sys2Appl a(iniFile);
      a.run();
   }
```

```
   else{
      CpmTop3::Sys2Appl a(iniFile);
      a.run();
   }
   return 2;
}
*/

//////////////////////////////////////////////////////////////////////

//////////////////// menueGenerator() ////////////////////////////////////
/*                              Example of usage:
int main(void)
{
   Z nMenue=4;
   Word title="Hilfsprogramme zur fotografischen Belichtung. Ulrich\
    Mutze\
    1999";

   V<Word> items(nMenue);
   items[1]="Berechnung der Belichtungszeit fuer fotografische\
    Aufnahmen"\
   ;
   items[2]="Berechnung der Leuchtdichte aus fotografischen\
    Belichtungsdaten";
   items[3]="Berechnung des Logarithmus zur Basis 2";
   items[4]="Ablesung logarithmischer Skalen";

   V< F<Z,Z> > f(nMenue);
   f[1]=F<Z,Z>(test3b); // functions to be called on menue item.
   // Functions
   // where of
      // type void f(void) before, change to Z f(Z) is trivial
   f[2]=F<Z,Z>(test3a);
   f[3]=F<Z,Z>(test0);
   f[4]=F<Z,Z>(test2);

   return menueGenerator(title,items,f,1);
}
*/

   Z menueGenerator(Word const& title, const V<Word>& items,
      const V< F<Z,Z> >& f, Z lang=2 );
// title holds a title for the program part generated by means of this
// function items holds the titles of the main menue items f holds
// the functions to be executed for the various menue items. If
// these are actually coded as normal functions of prototype Z g(Z),
// then e.g. f[3]=F<Z,Z>(g) is a valid initialization of component
// of f.
// lang=1 für deutsche Texte, lang=2 for English texts
```

```
} // namespace

#endif
```

101 cpminifilebasapp.cpp

```
//? cpminifilebasapp.cpp
//? Status of work 2008-10-25.
//?
#include <iomanip>
#include <cpminifilebasapp.h>
#include <cpmvo.h>
#include <cpmgreg.h>
#include <cpmdimdef.h>
#include <cpmsystemdependencies.h>
#include <cpmgraph.h>

using namespace CpmRoot;
using namespace CpmArrays;
using namespace CpmFunctions;
using namespace CpmSystem;

using CpmRootX::RecordHandler;

using CpmGraphics::Frame;
using CpmGraphics::Graph;
using CpmGraphics::Color;

using CpmApplication::IniFileBasedApplication;
using CpmApplication::args_;
using CpmApplication::Task;
using CpmSystem::glue;

bool CpmApplication::docu=true;
Z CpmApplication::dimApp=2;

//////////////////////////// IniFileBasedApplication ///////////////////

Z CpmApplication::setDimApp( Word const& program, Word const& path)
{
   const Z mL=1;
   Word loc="CpmApplication::setDimApp(";
   loc&=program;
   loc&=")";
   CPM_MA
   V<Word> proIni("",program&".ini"); // only one file name
   Word pathExt=Message::ext(path);
   RecordHandler rch(proIni, pathExt,
      "called from CpmApplication::setDimApp");
   Word sec("selection");
   Z mode_;
   rch.read(sec,"mode",mode_);
   if (mode_!=2 && mode_!=3){
```

```
      cpmmessage("mode="&cpm(mode_)&" not useful as space dimension");
   }
   else dimApp=mode_;
   CPM_MZ
   return mode_;
}

IniFileBasedApplication::IniFileBasedApplication(
Word const& program):
size(1),rank(1),
rch(program&".ini"),
   // Here already the initialization does substantial work.
   // It loads the data member rch ( for record handler) with the data
   // read from ini-file Message::ext(program&".ini")
sel_(0),mode_(0),
inFiles(0),outFiles(0),appendInFiles_(0),appendOutFiles_(0),
runName_("xxx")
{
   const Z mL=1;
   Word loc="IniFileBasedApplication(";
   loc&=program;
   loc&=")";
   CPM_MA
   size=CpmMPI::Cpm_com.getSize();
   rank=CpmMPI::Cpm_com.getRank();
   programName_=program;
   Word iniFileName(Message::ext(program&".ini"));

   Word sec("selection");
   Z ped=0; // degree of pedantry
   tWaitAnnounceScreen_=4;
   rch.read(sec,"tWaitAnnounceScreen",tWaitAnnounceScreen_,ped);
      // needs not to be there
   ped=1;
   rch.read(sec,"cpmverbose",cpmverbose,ped);
   rch.read(sec,"cpmdbg",cpmdbg,ped);
      // = cpmdbg controls whether cpmassert warns or stops
   CpmTime::Greg date;
   date.now();
   Word datw=date.getDate();
   cpmcerr<<endl<<"date: "<<datw<<endl;
   cpmdata<<endl<<"date: "<<datw<<endl;

   cpmmessage(" Listing all data from "&iniFileName);
   bool writeComments=false;
   rch.writeFormatted(cpmcerr,writeComments);
   cpmmessage(" End of listing all data from "&iniFileName);

   rch.read(sec,"sel",sel_,ped);
   if (sel_<0) docu=false;
```

```
   rch.read(sec,"runName",runName_);
   if (runName_=="auto"){
      runName_=Message::getRunId();
   }
// coupling the mode parameter to the dimension options set for
// compilation. We do not enforce the definition of CPM_DIMDEF2
// or CPM_DIMDEF3 and create only a warning if it is not defined.
   rch.read(sec,"mode",mode_,ped);
   if (mode_!=2 && mode_!=3){
      cpmmessage("mode="&cpm(mode_)&" not useful as space dimension");
   }
   else{ // in this case mode_ is interpreted as a dimension selector
      dimApp=mode_;
      cpmmessage("mode="&cpm(mode_)&"=dimApp");
#if defined(CPM_DIMDEF2)&&defined(CPM_DIMDEF3)
      cpmmessage("both CPM_DIMDEF2 and CPM_DIMDEF3 were defined");
      cpmmessage("mode and CPM_DIMDEF match");
#elif defined(CPM_DIMDEF2)
      cpmmessage("only CPM_DIMDEF2 was defined");
      if (mode_==3) cpmwarning("mode and CPM_DIMDEF do not match");
#elif defined(CPM_DIMDEF3)
      cpmmessage("only CPM_DIMDEF3 was defined");
      if (mode_==2) cpmwarning("mode and CPM_DIMDEF do not match");
#else
      cpmwarning("Neither CPM_DIMDEF3 nor CPM_DIMDEF2 was defined");
#endif
      if (sel_!=0) runName_&=cpmwrite(mode_);
         // mode indicator appended to run name
   }
// consistent creation of input-filenames
// no opening of files !
   sec="input";
   Z i,numberOfFiles=0;
   rch.read(sec,"numberOfFiles",numberOfFiles,ped);
   if (numberOfFiles>0){
      inFiles=V<Word>(numberOfFiles);
      appendInFiles_=V<Z>(numberOfFiles,0);
      for (i=1;i<=numberOfFiles;++i){
         rch.read(sec,"files"&cpm(i),inFiles[i],ped);
         rch.read(sec,"doc"&cpm(i),appendInFiles_[i],ped);
      }
   }
   Word indictorName="_"&runName_&".txt";
// consistent creation of output-filenames
// no opening of files !
   sec="output";
   bool succ=rch.read(sec,"numberOfFiles",numberOfFiles,ped);
   if (succ){ // then read outFiles and
      // appendOutFiles_from ini-file
      if (numberOfFiles>0){
```

```
         outFiles=V<Word>(numberOfFiles);
         appendOutFiles_=V<Z>(numberOfFiles);
         for (i=1;i<=numberOfFiles;++i){
            rch.read(sec,"files"&cpm(i),outFiles[i],ped);
            outFiles[i]=outFiles[i]&indictorName;
            rch.read(sec,"doc"&cpm(i),appendOutFiles_[i],ped);
            cpmmessage(loc&": outFiles["&cpm(i)&"] = "&outFiles[i]);
         }
      }
   }
   else{ // Then create a outFiles and appendInFiles_
      // automatically such that there is one outFile
      // per inFile and the docu status is the same for
      // in- and out-files
      if (numberOfFiles>0){
         outFiles=V<Word>(numberOfFiles);
         appendOutFiles_=V<Z>(numberOfFiles);
         Word pre("out_");
         for (i=1;i<=numberOfFiles;++i){
           outFiles[i]=pre&inFiles[i].baseName().appBefExt(indictorName);
            appendOutFiles_=appendInFiles_;
            cpmmessage(loc&": outFiles["&cpm(i)&"] = "&outFiles[i]&
               " !auto-created !");
         }
      }
   }
   // creating (for output) and opening of these files has to occur in
   // function doTheWork. In function documentTheWork, the output files
   // will be tried to open. If this is not possible a message on cpmcerr
   // will document this. The reason can be that function doTheWork did
   // not create the file.
   if (cpmverbose>=mL && rank==1&&!cpmsilent()){
      cpmdata<<endl<<"Input values as read from file:"<<endl;
      rch.getRecord().writeFormatted(cpmdata,writeComments);
   }
   tStart_=cpmtime();
   CPM_MZ
}

namespace{
   Word chain(V<Word> const& vw)
      // putting words together with a separating blank
   {
      Word res;
      for (Z i=vw.b();i<vw.e();++i) res&=(vw[i]&" ");
      res&=vw.last();
      return res;
   }
}
```

```
void IniFileBasedApplication::announceTheWork()
{
   if (tWaitAnnounceScreen_<=0) return;
// defining colors
   Color cBack(Z(0),Z(0),Z(60));
      // background color
   Color cProg(Z(255),Z(255),Z(255));
      // color for program title and the last line
   Color cTopics(Z(255),Z(255),Z(0));
      // color for the topics under consideration
   Color cSources(Z(50),Z(50),Z(120));
      // color for the text concerning sources
   Color cInfo(Z(100),Z(100),Z(240));
      // color for the text concerning info

   Word file=programName_;
      // name of the ini-file but without file extension
      // This name may be overridden by reading a name from the
      // ini-file

   Word beg0="        ";
   Word wdef=beg0&" ... ";
   V<Word> line1(1,wdef);
   V<Word> line2(1,wdef);
   V<Word> line3(1,wdef);
   V<Word> line4(1,wdef);
   V<Word> line5(1,wdef);

   Word sec="about";
// optionally overriding preset values
   cpmrhf(file);
   cpmrhf(line1);
   cpmrhf(line2);
   cpmrhf(line3);
   cpmrhf(line4);
   cpmrhf(line5);

// The topic lines enter primarily as lists of words
// which will be combined into a single Word containing spaces.
   Word beg1="Topics: ";
   V<Word> topics(0);
   topics<<beg1+chain(line1);
   topics<<beg0+chain(line2);
   topics<<beg0+chain(line3);
   topics<<beg0+chain(line4);
   topics<<beg0+chain(line5);

   V<Word> lines(0);
   V<Color> colors(0);
   Word tit=CpmApplication::title_;
```

```
lines<<("Program: "&tit);
colors<<cProg;

lines<<"";
colors<<cProg;

lines<<"Editing the input-file "+file+
   " allows controlling the program.";
colors<<cInfo;

lines<<"See this file and "+tit+"_readme.txt"+" for information.";
colors<<cInfo;

lines<<"";
colors<<cProg;

for (Z i=topics.b();i<=topics.e();++i){
   lines<<topics[i];
   colors<<cTopics;
}
lines<<"";
colors<<cProg;

lines<<
   "This program is based on the C+- class system by Ulrich Mutze,";
colors<<cSources;

lines<<
   "code modified from Numerical Recipes in C by William H. Press,";
colors<<cSources;

lines<<
   "Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery,";
colors<<cSources;

lines<<"the OpenGL Utility Toolkit (GLUT) by Mark J. Kilgard,";
colors<<cSources;

lines<<"and the C++ programming language by Bjarne Stroustrup.";
colors<<cSources;

lines<<"";
colors<<cProg;

lines<<"The program will start when the status bar shows tLeft = 0.";
colors<<cProg;

Frame fr(cBack);
fr.setText(lines,colors);
```

```
      // placing lines of text reasonably into
      // the available space (adjusts font size to needs)
   bool writeOnShow=(cpmverbose>1);
   fr.display(writeOnShow);
   tWaitWorkScreen_=4;
   using CpmApplication::args_;
   if (args_.dim()>=4) tWaitAnnounceScreen_=args_[4].toR();
   if (args_.dim()>=6) tWaitWorkScreen_=args_[6].toR();
      // from the command line or the ini-file one may have
      // a different request that, then, has priority
   cpmwait(tWaitAnnounceScreen_,2);
      // show the remaining time on pane 2
}

void IniFileBasedApplication::documentTheWork()
{
   Z mL=1;
   Word loc="IniFileBasedApplication::documentTheWork()";
   CPM_MA
   tStop_=cpmtime();
   R tRun=tStop_-tStart_;
   using CpmRootX::toString;
   ostringstream ost;
   Z i,nIn=inFiles.dim();
   Z nOut=outFiles.dim();
   CpmTime::Greg date;
   date.now();
   Word dn=date.getDate(false);
   ost<<"Date: "<<dn;
   ost<<endl<<"run time in seconds: "<<tRun<<endl;
   ost<<"Cpm documentation file for selection "<<sel_
      <<" of program "<<programName_
      <<endl<<" and run "<<runName_<<endl
      <<"What follows is obtained by concatenating all input files"
      <<" and all output files"<<endl;
   Word inFiles0=rch.getFileName();
   IFileStream iTemp1(inFiles0);
   ost<<endl<<("Input file "&inFiles0&": ")<<endl<<toString(iTemp1());
   for (i=1;i<=nIn;i++){
      ost<<endl<<"Input file #"<<i<<": "<<inFiles[i]<<endl;
      if (appendInFiles_[i]==0){
            ost<<" not shown";
      }
      else{
         IFileStream iTemp2(inFiles[i]);
         ost<<toString(iTemp2());
      }
   }
   if(nOut>0){
      for (i=1;i<=nOut;i++){
```

```
         ost<<endl<<"Output file #"<<i<<": "<<outFiles[i];
         if (appendOutFiles_[i]==0){
            ost<<" not shown";
         }
         else{
            bool safe=false;
            IFileStream oTemp(outFiles[i],safe);
               // IFile is OK here since we'll read
               // the outfiles which have been created (or not)
            if (!oTemp()){
               Word mes=loc;
               mes&=": unable to open "&outFiles[i]&
                  ";\n was probably not created in function doTheWork";
               cpmmessage(mes);
               ost<<" not created";
            }
            else{
               ost<<endl<<toString(oTemp());
            }
         }
      }
   }
   Word docuFileName=programName_&"_"&runName_&"_docu.txt";
   OFileStream out(docuFileName);
   out()<<ost.str();
   cpmdata<<endl<<endl<<"// Full content of rch:"<<endl;
   bool wrtCom=(sel_==0 ? false : true);
   rch.getRecord().writeFormatted(cpmdata,wrtCom);
   out()<<endl<<endl<<"// Full content of rch:"<<endl;
   rch.getRecord().writeFormatted(out(),wrtCom);
   // added 2004-09-14 since with the new hierarchical feature of
   // ini-files a mere copy of the top-level ini-file does not allow
   // to trace all data that made the program run as it did.
   // So we include a concise (reordered) record of all data here.
   // Even the comments are written in order not to loose comments
   // explaining newly introduced quantities.
   CPM_MZ
}

void IniFileBasedApplication::run()
{
   cpmurgent("IniFileBasedApplication::run() started");
   cpmstatus(programName_&" started",4); //
   announceTheWork();
   doTheWork();
   bool writeComments=(sel_>0 ? true : false);
   if (docu && rank==1&&!cpmsilent()){ // at least docu of input
      // since this can conveniently used for continuation of work based
      // on approved runs, one should not loose the comments in
      // doing so
```

```
      Word leanDocuFileName=programName_&"_"&runName_&"_.ini";
      OFileStream ldf(leanDocuFileName);
      rch.getRecord().writeFormatted(ldf(),writeComments);
      if (sel_>0) documentTheWork(); // also docu of results
   }
   cpmurgent("IniFileBasedApplication::run() done");
      // to get timing data also for cpmverbose=0. For comparing
      // efficiency of compilers it is wise reduce messaging to
      // a minimum, but timing is of impotance then
   cpmstatus(programName_&" done",4); //
}

void IniFileBasedApplication::setModeFromInFiles()
      // In some programs based on IniFileBasedApplication the inFiles
      // are assumed to be the outFiles of other programs also based
      // on IniFileBasedApplication. These outfiles have the mode
      // parameter normally appended, and thus allow the mode-value
      // which fits the nature of these files to be read from
      // the file names of the inFiles under consideration. This
      // mechanism (which changes the values of mode and runName)
      // will be executed by calling the present function
   {
      Word loc("IniFileBasedApplication::setModeFromInFiles()");
      Word t2="2.txt";
      Word t3="3.txt";
      Z tL=5; // tL==t2.dim()==t3.dim()
      Z d=inFiles.dim();
      cpmassert(d>0,loc);
      Word tail=inFiles[1].tail(tL);
      Z res;
      bool b(true);
      if (tail==t2) res=2;
      else if(tail==t3) res=3;
      else
         b=false;

      for (Z i=2;i<=d;i++){
         if (inFiles[i].tail(tL)!=tail)
            b=false;
      }
      if (b){
         mode_=res;
         Word mo=(mode_==2 ? "2" : "3");
         Word rnt=runName_.tail(1);
         if (rnt=="2" || rnt=="3"){
               // if runName ends in 2 or 3, it should get the end which
               // fits mode
            if (rnt!=mo) runName_=runName_.cut(1)&mo;
         }
      }
```

```
   }

namespace{

bool ini_args_(V<Word> const& fl)
{
   Z mL=1;
   Word loc("CpmApplication::ini_args_(V<Word>)");
   RecordHandler rch(fl,"",loc);
   if (!rch.isVal()){
      cpmwarning(loc&
         ": RecordHandler not valid, commandline not extended");
      CPM_MZ
      return false;
      // Thus no action if file fn does not exist
   }
   V<Word> args=rch.comLine();
   args_=CpmRootX::combine(args_,args);
   CPM_MZ
   return args_.dim()>0;
}

bool ini_args_(Word const& fl)
{
   Z mL=1;
   Word loc("CpmApplication::ini_args_(Word)");
   RecordHandler rch(fl);
   if (!rch.isVal()){
      cpmwarning(loc&
         ": RecordHandler not valid, commandline not extended");
      CPM_MZ
      return false;
      // Thus no action if file fn does not exist
   }
   V<Word> args=rch.comLine();
   args_=CpmRootX::combine(args_,args);
   CPM_MZ
   return args_.dim()>0;
}

Z menFun(Word const& pn, V< F<Word,Z> > const& appList)
{
   Word loc("menFun("&pn&",V< F<Word,Z> >)");
   Z mL=1;
   CPM_MA
   Word iniFile=pn&".ini";
   Word dummy;
   Z i;
   bool succ=ini_args_(iniFile);
   if (!succ) cpmerror("ini-file "&iniFile
```

```
      &" not found");
   Z sel=1;
   R tWait=1;
   Z na=args_.dim();
   cpmmessage(mL,"args_.dim()="&cpm(na));
   if (na>=2) sel=args_[2].toZ();
   if (na>=3) cpmverbose=args_[3].toZ();
   if (na>=4) tWait=args_[4].toR();
   cpmmessage("sel="&cpmwrite(sel));
   cpmmessage("cpmverbose="&cpmwrite(cpmverbose));
   Z nn=appList.dim();
   if (sel<1) cpmerror("main_(): sel<1");
   else if (sel>nn){ // doing all
      cpmmessage("sel="&cpm(sel)&
         " asks for doing the whole list");
      for (i=1;i<=nn;++i){
         cpmstatus("menu item "&cpm(i)&" started",4);
         appList[i](dummy);
         cpmstatus("menu item "&cpm(i)&" done",4);
      }
   }
   else{ // doing only the selected item
      cpmmessage("sel="&cpm(sel)&
         " asks for doing a single list component");
      cpmstatus("menu item "&cpm(sel)&" started",4);
      appList[sel](dummy);
      cpmstatus("menu item "&cpm(sel)&" done",4);
      // this calls the selected program
   }
   cpmwait(tWait);
   CPM_MZ
   return sel;
}

Z subMenFun(Word const& dummy, Word const& pn,  F<Word,Z>  const& f)
{
   Word loc("subMenFun("&pn&", F<Word,Z>)");
   Z mL=1;
   CPM_MA
   dummy; // not needed as the name says
// trying to find data
   Word iniFile=pn&".ini";
   RecordHandler rch(iniFile);
   bool succ=rch.isVal();
   if (!succ){
      cpmerror(loc&": no valid ini-file found");
      return 0; // will never be reached
   }
   else{
      cpmmessage(loc&": got valid data from "&rch.getFileName());
```

```
   }
// extracting task-related file names
   Word sec("tasks");
   Z begin=1,end=1;
   rch.read(sec,"begin",begin);
   rch.read(sec,"end",end);
   Z ped=0;
   V<Z> ignore(0);
   rch.read(sec,"ignore",ignore,ped);
   Z di=ignore.dim();
   ped=2;
   succ=false;
   V<Word> tasks;
   V<Word> varTask;
   succ=rch.read(sec,"tasks01",tasks,ped);
   ped=1; // further tasks may not be there
// maximum of 20 task groups prepared; since each task group
// may have as many tasks as one may write names in a line of the
// ini-file, this is not really a restriction. However, having only
// a small constant number, say 3, in a line, limits the total
// number (to 60 for that number, which is probably more than ever
// needed.
   if (succ){
      succ=rch.read(sec,"tasks02",varTask,ped);
      if (succ) tasks&=varTask;
   }
   if (succ){
      succ=rch.read(sec,"tasks03",varTask,ped);
      if (succ) tasks&=varTask;
   }
   if (succ){
      succ=rch.read(sec,"tasks04",varTask,ped);
      if (succ) tasks&=varTask;
   }
   if (succ){
      succ=rch.read(sec,"tasks05",varTask,ped);
      if (succ) tasks&=varTask;
   }
   if (succ){
      succ=rch.read(sec,"tasks06",varTask,ped);
      if (succ) tasks&=varTask;
   }
   if (succ){
      succ=rch.read(sec,"tasks07",varTask,ped);
      if (succ) tasks&=varTask;
   }
   if (succ){
      succ=rch.read(sec,"tasks08",varTask,ped);
      if (succ) tasks&=varTask;
   }
```

```
if (succ){
   succ=rch.read(sec,"tasks09",varTask,ped);
   if (succ) tasks&=varTask;
}
if (succ){
   succ=rch.read(sec,"tasks10",varTask,ped);
   if (succ) tasks&=varTask;
}
if (succ){
   succ=rch.read(sec,"tasks11",varTask,ped);
   if (succ) tasks&=varTask;
}
if (succ){
   succ=rch.read(sec,"tasks12",varTask,ped);
   if (succ) tasks&=varTask;
}
if (succ){
   succ=rch.read(sec,"tasks13",varTask,ped);
   if (succ) tasks&=varTask;
}
if (succ){
   succ=rch.read(sec,"tasks14",varTask,ped);
   if (succ) tasks&=varTask;
}
if (succ){
   succ=rch.read(sec,"tasks15",varTask,ped);
   if (succ) tasks&=varTask;
}
if (succ){
   succ=rch.read(sec,"tasks16",varTask,ped);
   if (succ) tasks&=varTask;
}
if (succ){
   succ=rch.read(sec,"tasks16",varTask,ped);
   if (succ) tasks&=varTask;
}
if (succ){
   succ=rch.read(sec,"tasks17",varTask,ped);
   if (succ) tasks&=varTask;
}
if (succ){
   succ=rch.read(sec,"tasks18",varTask,ped);
   if (succ) tasks&=varTask;
}
if (succ){
   succ=rch.read(sec,"tasks19",varTask,ped);
   if (succ) tasks&=varTask;
}
if (succ){
   succ=rch.read(sec,"tasks20",varTask,ped);
```

```
      if (succ) tasks&=varTask;
   }
   Z nTasks=tasks.dim();
   cpmassert(begin>=1,loc);
   cpmassert(begin<=nTasks,loc);
   cpmassert(end>=1,loc);
   cpmassert(end<=nTasks,loc);
   sec="control";
   R tWait=0;
   rch.read(sec,"tWait",tWait);
   Z i,sum=0;
   for (i=begin;i<=end;++i){
      bool vi=true;
      if (di>0){
         for (Z j=1;j<=di;++j){
            if (i==ignore[j]){
               vi=false;
               break;
            }
         }
      }
      if (vi){
         Word iniFileForTask=tasks[i];
         cpmmessage(iniFileForTask&" considered as task");
         sum+=f(iniFileForTask);
            // the tasks determine the name of the ini-file which
            // will deliver data
         cpmwait(tWait);
      }
   }
   CPM_MZ
   return sum;
}

}// namespace


Task Task::subMen(Word const& name, F<Word,Z> const& f)
{
   F<Word,Z> f1=F2<Word,Word,F<Word,Z>,Z>(name,f)(subMenFun);
   return Task(name,f1);
}

Task Task::men(Word const& name, V<Task> const& tasks)
// That this works shows the power of the function type
{
   Z td=tasks.dim();
   V< F<Word,Z> > fs(td);
   for (Z i=1;i<=td;++i) fs[i]=tasks[i].get_f();
   F<Word,Z> f=F1< Word, V< F<Word,Z> >, Z >(fs)(menFun);
```

```
   return Task(name,f);
}

//////////////////////////////////////////////////////////////////////

Z CpmApplication::menueGenerator(Word const& title, const V<Word>& items,
 const V< F<Z,Z> >& f, Z lang )
{
   Z i;
   Word menue1, menue2, askForEnter, askForExit, choice1, choice2,
    choice3;
   if (lang==1){
      menue1="Folgende Moeglichkeiten stehen zur Auswahl:";
      menue2="Nach dieser Berechnung stehen folgende Moeglichkeiten zur\
       Auswahl:";
      askForEnter="Eingabe der Wahlnummer:  ";
      askForExit="Programm beenden";
      choice1="Programm\
       beenden............................................. 0";
      choice2="Rueckkehr zum\
       Hauptmenue.................................... 1";
      choice3="Rueckkehr zum letzten\
       Menuepunkt........................... 2";
   }
   else {
      menue1="Main Menue: the following selections are possible";
      menue2="Continuation Menue: the following selections are possible";
      askForEnter="Enter selection:  ";
      askForExit="EXIT from program";
      choice1="EXIT from\
       program............................................ 0";
      choice2="Return to Main\
       Menue...................................... 1";
      choice3="Return to begin of current\
       item........................... 2";
   }

   V<Word> itemsNew=items&askForExit;
   Z nMenue=itemsNew.dim();
   Vo<Z> sizes(nMenue);
   for (i=1;i<=nMenue;i++) sizes[i]=itemsNew[i].dim();
   Vo<Z> orderedSizes=sizes.decreasingCopy();
   Z maxSize=orderedSizes[1];
   maxSize+=2;

MAINPOINT: cout<<endl<<title;
   cout<<endl<<menue1;
   Z fillsize;
   for (i=1;i<=nMenue;i++){
      cout<<endl;
```

```
      cout<<itemsNew[i];
      fillsize=maxSize-sizes[i];
      for (Z j=1;j<=fillsize;j++) cout<<".";
      cout<<" "<<i;
   }
   cout<<endl<<askForEnter;
   Z sel; cin>>sel;
   if (sel==nMenue) goto END;
SUBPOINT:
   f[sel](1);
   cout<<endl<<menue2;
   cout<<endl<<choice1;
   cout<<endl<<choice2;
   cout<<endl<<choice3;
   cout<<endl<<askForEnter;
   Z sel2; cin>>sel2;
   switch(sel2){
      case 0:  goto END;
      case 1:  goto MAINPOINT;
      case 2:  goto SUBPOINT;
      default: goto END;
   }
END:
   return 1;
}
```

102 cpminterfaces.h

```
//? cpminterfaces.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_INTERFACES_H_
#define CPM_INTERFACES_H_
/*

   Description: Declaration macros for interfaces for
      CPM classes. No macro argument, assuming that the
      client class will typedef her name as Type; some macros
      assume that the client class also defines ScalarType
      and CloneBase

*/
#include <cpmnumbers.h>
   // includes cpmdefinitions.h
#include <cpmbasicinterfaces.h>
#include <cpmmpi.h>
   // contains valid (though trivial) declarations even for
   // CPM_USE_MPI not defined

// Notice that using types Z and R in macros is only safe as
// CpmRoot::Z and CpmRoot::R since they might be used in a context
// where a corresponding using directive is not available.

// defining member functions send and rec(ieve) for communication between
// processes if the message passing interface library is being used
// (for parallel programming)
//using CpmMPI::Com;

#if defined(CPM_USE_MPI)

   #define CPM_MPI_O\
   bool send(Z dest, CpmRoot::Z tag, CpmMPI::Com comm=CpmMPI::Com()\
   )const\
      { \
         ostringstream ost;\
         CpmRoot::Z wrtPrcMem=CpmRoot::wrtPrc;\
         CpmRoot::wrtPrc=18;\
         bool b=prnOn(ost);\
         CpmRoot::wrtPrc=wrtPrcMem;\
         return b ? comm.sendStr(ost.str(),dest,tag): false;\
      }

   #define CPM_MPI_I\
      bool rec(CpmRoot::Z source, CpmRoot::Z tag, CpmMPI::Com\
       comm=CpmMPI::Com())\
```

```
      {\
         std::string s;\
         bool res=comm.recStr(s,source,tag);\
         std::istringstream ist(s);\
         bool b=scanFrom(ist);\
         return b ? res : false;\
      }

#else // !defined(CPM_USE_MPI)

   #define CPM_MPI_O\
   bool send(CpmRoot::Z dest, CpmRoot::Z tag, CpmMPI::Com comm=\
   CpmMPI::Com())const{ dest; tag; comm; return true;}

   #define CPM_MPI_I\
      bool rec(CpmRoot::Z source, CpmRoot::Z tag, CpmMPI::Com\
       comm=CpmMPI::Com()){ source; tag; comm; return true;}

#endif

// input and output functions; the function read is the common interface
// to both built-in types and Cpm classes to read from commented files
// ( which does not work for R and Z via >>)

// all IMPL-MACROS get completed by their corresponding
// MPI-functionality.
// Then it is guaranteed that all classes define proper inter-process
// communication.
// Notice that for non-class types like Z and R, there is no re-definition
// of operators >> an <<.

#define CPM_IO_IMPL\
   friend std::ostream& operator<<(std::ostream& out, Type const& x)\
      { x.prnOn(out); return out;}\
   friend std::istream& operator>>(std::istream& in, Type& x)\
      { x.scanFrom(in); return in;}\
      CPM_MPI_O\
      CPM_MPI_I

// declaration macros without public qualifications
// with and without virtual qualification.

#define CPM_IO_V\
   virtual bool prnOn(std::ostream&)const;\
   virtual bool scanFrom(std::istream&);\
   CPM_IO_IMPL

#define CPM_IO\
   bool prnOn(std::ostream&)const;\
   bool scanFrom(std::istream&);\
```

```
   CPM_IO_IMPL

// sometimes one needs the O-part (= output part) in isolation:

#define CPM_O_IMPL\
   friend std::ostream& operator<<(std::ostream& out, Type const& x)\
      { x.prnOn(out); return out;}\
   CPM_MPI_O

#define CPM_O_V\
   virtual bool prnOn(std::ostream&)const;\
   CPM_O_IMPL

#define CPM_O\
   bool prnOn(std::ostream&)const;\
   CPM_O_IMPL

// help to unify defining member functions
//    bool prnOn(ostream& str)const;
//    bool scanFrom(istream& str);
// in classes with many data members that define prnOn and scanFrom.
// Notice that the ; is missing, so that the call hs to provide it.
// E.g.
/*
   bool prnOn(ostream& str)const
   {
      cpmp(x1);
      cpmp(x2);
      cpmp(x3);
      return true;
   }

   bool scanFrom(istream& str)
   {
      cpms(x1);
      cpms(x2);
      cpms(x3);
      return true;
   }
*/
// defining the nameOf function easily, no ';' at the end
#define CPM_NAM(X) Word nameOf()const{ return Word(#X);}
#define CPM_NAM_V(X) virtual Word nameOf()const{ return Word(#X);}
// declarations that need a ';' behind. The rule is that any CPM_...
// needs no ; behinf it, but cpm... needs it.
#define cpmwt(X) if (!CpmRoot::writeTitle((X),str)) return false
   // wt for 'write title'
#define cpmwat if (!CpmRoot::writeTitle(nameOf().str(),str)) return false
   // wt for 'write automatic title', needs no argument.
#define cpmwbt if (!CpmRoot::writeTitle((nameOf()&" begin").str(),str))\
```

```
 return false
   // wbt for 'write begin title'
#define cpmwet if (!CpmRoot::writeTitle((nameOf()&" end").str(),str))\
 return false
   // wet for 'write end title', needs no argument.
#define cpmp(X) if (!CpmRoot::prnOn((X),str)) return false
   // p for print
#define cpms(X) if (!CpmRoot::scanFrom((X),str)) return false
   // s for scan
#define cpmrh(X) rch.read(sec,#X,X)
   // regularly occurring idiom for reading from a RecordHandler
   // read handler. Stops the program if the quantity cannot be
   // read.
#define cpmrhf(X) rch.read(sec,#X,X,1)
   // regularly occurring idiom for reading from a RecordHandler
   // read handler, option pedantic = 1, so that only a warning
   // gets issued if the quantity cannot be read.
   // 'f' for 'floppy'
#define cpmwh(X) rch.write(sec,#X,X)
   // regularly occurring idiom for writing to a RecordHandler
   // write handler
#define cpmrc(X) rc_.get(sec,#X,X)
   // regularly occurring idiom for reading from the attribute
   // Recordable::rc_

// universal interface to basic character string class string
// This functionality is sometimes called marshalling and
// toStr is also called 'dump' and fromStr also 'load'

#define CPM_WORD\
   CpmRoot::Word toWord()const\
   { std::ostringstream ost; prnOn(ost);\
   return CpmRoot::Word(ost.str());}

// order-related stuff now in basicinterfaces

// multiplicative structure with respect to scalars of type ScalarType.
// version where mutating member functions (actually operator *=)  are
// primary
#define CPM_PRODUCT_SCALAR_M\
   Type& operator *=(ScalarType const& s);\
   Type& operator /=(ScalarType const& s)\
   {return operator *=(CpmRoot::inv<ScalarType>(s));}\
   Type operator *(ScalarType const& s)const\
   { Type res=*this; return res*=s;}\
   friend Type operator *(ScalarType const& s, Type const& x)\
   { Type res=x; return res*=s;}\
   Type operator /(ScalarType const& s)const\
   { Type res=*this; return res/=s;}
```

```
// version where constant member functions (actually operators *) are
// primary
#define CPM_PRODUCT_SCALAR_C\
   Type operator *(ScalarType const& s)const;\
   Type operator /(ScalarType const& s)const\
   { return *this*CpmRoot::inv<ScalarType>(s);}\
   friend Type operator *(ScalarType const& s, Type const& x)\
    { return x*s;}\
   Type& operator *=(ScalarType const& s){ return *this=*this*s;}\
   Type& operator /=(ScalarType const& s){ return *this=*this/s;}

// linear structure with respect to scalars of type ScalarType.
// version where mutating member functions (actually operator *=, +=) are
// primary
#define CPM_SCALAR_M\
   Type& operator *=(ScalarType const& s);\
   Type& operator +=(ScalarType const& s);\
   Type& operator -=(ScalarType const& s){return operator +=(-s);}\
   Type& operator /=(ScalarType const& s)\
   {return operator *=(CpmRoot::inv<ScalarType>(s));}\
   Type operator *(ScalarType const& s)const\
   { Type res=*this; return res*=s;}\
   friend Type operator *(ScalarType const& s, Type const& x)\
   { Type res=x; return res*=s;}\
   Type operator +(ScalarType const& s)const\
   { Type res=*this; return res+=s;}\
   Type operator -(ScalarType const& s)const\
   { Type res=*this; return res-=s;}\
   Type operator /(ScalarType const& s)const\
   { Type res=*this; return res/=s;}

// version where constant member functions (actually operators *,+) are
// primary
#define CPM_SCALAR_C\
   Type operator *(ScalarType const& s)const;\
   Type operator +(ScalarType const& s)const;\
   Type operator -(ScalarType const& s)const{ return *this+(-s);}\
   Type operator /(ScalarType const& s)const\
   { return *this*CpmRoot::inv<ScalarType>(s);}\
   Type& operator *=(ScalarType const& s){ return *this=*this*s;}\
   Type& operator +=(ScalarType const& s){ return *this=*this+s;}\
   Type& operator -=(ScalarType const& s){ return *this=*this-s;}\
   Type& operator /=(ScalarType const& s){ return *this=*this/s;}

// minimum infra structure for programming unbiased test for the validity
// of expected mathematical identities

#define CPM_TEST\
   Type ranVal(CpmRoot::Z j=0)const;\
   Type testVal(CpmRoot::Z)const;\
```

```
   CpmRoot::Z hashVal(void)const;

#define CPM_TEST_X\
   CPM_TEST\
   CpmRoot::R disVal(Type const&)const;\
   CpmRoot::R absVal(void)const;\
   CpmRoot::R absSqr(void)const;

// CPM_TEST_ALL, comprises all infra-structure functions
// which are defined in cpmnumbers.h for the bb-types.
#define CPM_TEST_ALL\
   CPM_TEST_X\
   Type inv()const;\
   Type con()const;\
   Type net(CpmRoot::Z)const;

// CPM_TEST_MOD is a modification of CPM_TEST which turned out
// to be needed in a proper implementation of class Q where it would
// be unnatural to have absVal R-valued instead of Q-valued. Also
// implementing absVal in terms of absSqure would not be suitable in
// this case.
#define CPM_TEST_MOD\
   CPM_TEST\
   CpmRoot::R disVal(Type const&)const;\
   Type absVal(void)const;\
   Type absSqr(void)const;

// descriptors
#define CPM_DESCRIPTORS\
   virtual Word nameOf(void) const;\
   virtual Word toWord(void)const;

// inversion
// lean inversion
#define CPM_INVERSION\
   Type operator !(void)const;\
   Type& operator /=(Type const& x){ return operator*=(!x);}\
   Type operator /(Type const& x)const\
   { return *this*!x;}

// more complex version of inversion which also brings the zero into the
// game (which in a group needs not to be a defined concept)
#define CPM_DIVISION\
   Type net(CpmRoot::Z i=0)const;\
   Type inv(void)const;\
   Type operator !(void)const{ return inv();}\
   Type& operator /=(Type const& x){ return operator*=(x.inv());}\
   Type operator /(Type const& x)const\
   { return *this*x.inv();}\
   friend Type net(Type const& x, CpmRoot::Z i=0)\
```

```
   { return x.net(i);}\
   friend Type inv(Type const& x){ return x.inv();}

#define CPM_DIFFERENCE\
   Type neg(void)const;\
   Type& operator -=(Type const& x){ return operator+=(x.neg());}\
   Type operator -(void)const{ return neg();}\
   Type operator - (Type const& x)const\
   { return *this+x.neg();}

#define CPM_SUM_PLAIN\
   Type& operator +=(Type const& x);\
   Type operator + (Type const& x)const;

#define CPM_DIFFERENCE_PLAIN\
   Type& operator -=(Type const& x);\
   Type operator -(void)const;\
   Type operator - (Type const& x)const;

// product, where the mutating member function *= is primary
#define CPM_PRODUCT_M\
   Type& operator *=(Type const& x);\
   Type operator * (Type const& x)const\
   { Type res=*this; return res*=x;}

// product, where the constant member function operator * is primary
#define CPM_PRODUCT_C\
   Type operator * (Type const& x)const;\
   Type& operator *=(Type const& x){ return *this=(*this)*x;}

// sum, where the mutating member function += is primary
#define CPM_SUM_M\
   Type& operator +=(Type const& x);\
   Type operator + (Type const& x)const\
   { Type res=*this; return res+=x;}

// sum, where the constant member function + is primary
#define CPM_SUM_C\
   Type operator + (Type const& x2)const;\
   Type& operator +=(Type const& x){ return *this=(*this)+x;}

// sum, where the constant member function + is primary
#define CPM_DIFF_C\
   Type operator - (Type const& x2)const;\
   Type& operator -=(Type const& x){ return *this=(*this)-x;}

// complex conjugation
#define CPM_CONJUGATION\
   Type con(void)const;\
   friend Type con(Type const& x){ return x.con();}\
```

```
   Type operator~(void)const{ return con();}

// lean complex conjugation without friend function
// (friend functions are better avoided since they cause
// problems with todays MS compilers when templates are
// heavily involved)
#define CPM_CONJ \
Type con(void)const;\
Type operator~(void)const{ return con();}

// Scalar Product and real valued dependents

#define CPM_DOT_PRODUCT \
ScalarType operator |(Type const&)const;\
CpmRoot::R absSqr()const\
{ return CpmRoot::absVal<ScalarType>(*this|*this);}\
CpmRoot::R absVal()const\
{  return ::sqrt(absSqr());}\
CpmRoot::R disVal(Type const& x)const\
{\
   CpmRoot::R a=absVal();\
   CpmRoot::R b=x.absVal();\
   CpmRoot::R d=(*this-x).absVal();\
   return CpmRoot::distFunc(a,b,d);\
}

#define CPM_NORMALIZE \
CpmRoot::R nor_()\
{\
   CpmRoot::R nOrig=absVal();\
   if (nOrig>0){\
      CpmRoot::R nInv=1./nOrig;\
      operator*=(ScalarType(nInv));\
   }\
   return nOrig;\
}

// linear structure with respect to scalars of type ScalarType.
// Inversion in ScalarType is not considered; has to be added in a class
// where this is needed (e.g in C)

#define CPM_LINEAR\
   CpmRoot::Z dim()const;\
   Type& operator +=(Type const& a);\
   Type& operator -=(Type const& a);\
   Type& operator *=(ScalarType const& r);\
   Type& neg(void);\
   friend CpmRoot::Z dim(Type const& x){ return x.dim();}\
   friend Type operator +(Type const& x1, Type const& x2)\
      { Type res=x1; return res+=x2;}\
```

```
   friend Type operator -(Type const& x1, Type const& x2)\
      { Type res=x1; return res-=x2;}\
   friend Type operator -(Type const& x)\
      { Type res=x; return res.neg();}\
   friend Type operator *(Type const& x, ScalarType const& s)\
      { Type res=x; return res*=s;}\
   friend Type operator *(ScalarType const& s, Type const& x)\
      { Type res=x; return res*=s;}

// a lean and collection of arithmetic operations for linear spaces with
// scalar product. This is now the basic interface of
// VVa<>, VVVa<>, and VVVVa<>.
#define CPM_LIN\
   CPM_SUM_C\
   CPM_DIFF_C\
   Type operator-(void)const;\
   Type operator*(ScalarType const& r)const;\
   Type& operator *=(ScalarType const& r)\
      { return *this=*this*r;}\
   CPM_DOT_PRODUCT\
   CPM_CONJ
#endif
```

103 cpminterval.h

```
//? cpminterval.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_INTERVAL_H_
#define CPM_INTERVAL_H_
/*

    Description:
      open intervals on the real axis endowed with the lattice structure
      meet and join (here operator | and operator &)

*/

#include <cpmlinalg.h>
   // it is considered natural to treat geometry
   // on the basis of linear algebra. Presently
   // CpmLinAlg only needed for being able to use R2
#include <cpmx.h>
#include <cpmzinterval.h>

namespace CpmGeo{

   using namespace CpmStd;

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::Word;
   using CpmArrays::IvZ;
   using CpmLinAlg::Z2;
   using CpmLinAlg::R2;
   using CpmArrays::V;
   using CpmArrays::Vl;
   using CpmArrays::X2;

//////////////////////// class Iv ////////////////////////////////////////

class Iv{ // open intervals in R
   // could be extended to interval arithmetics, therefore by the
   // clone macro polymorphic containers are supported

//independent quantities
   R left;
   R right;
// dependent quantities
   R center;
   R halfWidth;
   R absInv; // 1/absVal() or 0
```

```
// giving the dependent quantities the right values
   void update();
   void updateLight(); // does not order left,right
public:
   typedef Iv Type;
   typedef Iv CloneBase;
   virtual Iv* clone()const{ return new Iv(*this);}
   CPM_IO
   CPM_ORDER
   Word nameOf()const{ return Word("Iv");}
   Iv(R a, R b, bool ordered=false):left(a),right(b)
      { if (!ordered) update(); else updateLight();}
   Iv():left(0),right(0),center(0),halfWidth(0),absInv(0){}

   explicit Iv(R a):left(-cpmabs(a)),right(cpmabs(a)){updateLight();}
      // Interval [-|a|,|a|]

   explicit Iv(R2 const& r2):left(r2[1]),right(r2[2]){update();}
      // making an interval out of a pair of numbers

   Iv(const V<R>& vr);
      // generates the interval determined by the smallest
      // and the largest of the components vr

   bool isVoid()const{ return left==right;}
      // This shows that our intervals here are not fully
      // characterized when considered point sets. There
      // is only one void point set in R but many Iv's
      // with left==right

   R absVal()const{ return 2*halfWidth;}
      // length of interval = diameter of the set
   R radius()const{ return halfWidth;}
      // in one dimension an interval also is a sphere
   R rad()const{ return halfWidth;}
      //: radius

   R inf()const{ return left;}

   R sup()const{ return right;}

   R operator[](Z i)const{ if (i<=1) return left; else return right;}
      // since Iv and R2 are often to be used interchangingly
      // it is convenient to get to the data by the same syntax

   R mean()const{ return center;}
      // center

   R x(R p)const{ return left+p*absVal();}
      // returns the left end for p=0, the center for p=0.5,
```

```
      // and the right end for p=1.

   R operator()(R x)const{return cpmabs(x-center)-halfWidth;}
      // geometrical indicator function
      // zero on the boundary, positive outside
      // negative inside; gradient of the function gives the
      // outside-directed normal vector on the boundary

   Iv operator|(const Iv& iv)const;
      // minimum open interval that contains the union
      // join or l.u.b. (lowest upper bound) in lattice terminology

   Iv operator&(const Iv& iv)const;
      // section, intersection
      // meet or g.l.b (greates lower bound) in lattice terminology

   bool contains(R x, bool leftOpen=true, bool rightOpen=true)const
      // answers the question whether point x is an element of the
      // interval *this. The second and the third argument allow to
      // set the interpretation of *this as an open, closed , or
      // semi-closed interval
   {
      bool leftOK=(leftOpen ?  x>left : x>=left);
      bool rightOK=(rightOpen ?  x<right : x<=right);
      return leftOK&&rightOK;
   }

   bool contains(const Iv& iv)const
      // says whether iv is a subset (true subset 'or equal') of *this
      // or not.
   {  return left<=iv.left && iv.right<=right;}

   bool isSubsetOf(const Iv& iv)const
      // says whether *this is a subset (true subset 'or equal') of iv
      // or not.
   {  return iv.left<=left && right<=iv.right;}

   const Iv& stretch(R k, Z s=0);
      // multiplies the size of the interval by |k| and leaves
      // a point p invariant, where this (fixed) point is
      // (i)   the left end if s=-1
      // (ii)  the center if s=0
      // (iii) the right end if s=1
      // Error for other values of s

   const Iv& shift(R s){left+=s;right+=s; center+=s; return *this;}
      // shifts both ends of the interval by s

// modern form of moving both ends of the interval by the same
// step
```

```
Iv& shf_(R s){left+=s; right+=s; center+=s; return *this;}
   //: shift
   // no change of the other dependent data
Iv& operator+=(R s){left+=s;right+=s; center+=s; return *this;}
Iv& operator-=(R s){left-=s;right-=s; center-=s; return *this;}
Iv& operator*=(R s)
   // assumes s>0, multiplies the size by s, leaving the center
   // invariant
{
   halfWidth*=s; absInv*=cpminv(s);
   right=center+halfWidth; left=center-halfWidth;
   return *this;
}
const Iv& shiftLeftEnd(R s){ left+=s; update(); return *this;}

const Iv& shiftRightEnd(R s){ right+=s; update(); return *this;}

Vl<R> cen(Z n)const;
   //: centers
   // The interval is divided into n subintervals of equal length
   // and the n centers of these subintervals are returned as a
   // real vector in ascending order. n>=1 needed.
   // returns a lean array

V<R> centers(Z n)const{ return V<R>(cen(n));}
   // returns a conventional array

V<R> div(Z n)const;
   // equidistant division.
   // Regular case: n>=2
   // We return res such that res.dim()==n,
   // res[1]=inf(), res[n]=sup().
   // and res[i+1]-res[i]=const
   // Special cases: negative n: n acts as |n|
   // n=0 V<R>(0) returned
   // n=1 V<R>(1,mean()) returned

V<R> divGeoProg(R dt,Z n)const;
   // division according to geometric progression.
   // We return res such that res.dim()==n,
   // res[1]=inf(), res[n]=sup().
   // res[2]-res[1]=dt
   // and res[i+1]-res[i]=q*(res[i]-res[i-1])
   // for all i=2,...n-1 with some fixed q
   // Assumption thus n>2, 0<dt<absVal(). These
   // are checked via cpmassert

V<R> divSmooth(R dti,R dtf,Z n)const;
   // We return res such that res.dim()==n,
   // res[1]=inf(), res[n]=sup().
```

```
   // res[2]-res[1]=dti
   // res[n]-res[n-1]=dtf
   // and res[i] 'as slovly varying with i as possible'.
   // Assumption thus n>3, 0<dti+dtf<absVal(). These
   // are checked via cpmassert.
   // First intention was to minimize sum of (res[i]-res[i-1])^2
   // which leads (Lagrange multipliers needed for constraints
   // of the problem) to a matrix inversion problem which
   // I used numerically and found steps si=res[i]-res[i-1]
   // smoothly going to negative values and then smoothly
   // going back to positive ones. This is no solution to the
   // present problem. The present robust method is to place the
   // n-4 not yet determined points equidistantly and then
   // smoothing many times by a simple linear filter of mask size 3.
   // See implemnentation code for details.

V<Iv> divide(Z n)const;
   // The interval is divided into n subintervals of equal length
   // which are returned as a vector in ascending order. n>=1 needed.

R periodic(R x)const;
   // returns a function which is periodic with respect to
   // a periodic repetition of the interval and takes the value 0
   // at the left end of the interval and the value 1 at the right
   // one. In between it is linear. (thus it is a saw-tooth function)
   // This function creates an error if absVal()==0.

R ramp(R x)const;
   // is a function which takes the value 0
   // for all points left of the interval and the value 1 everywhere
   // right of it. In between it is linear. (thus it is a linear ramp)
   // If the interval has length 0, we return 0.5 for x==left==right

Z qun(R x, Z n, bool restrict=false)const;
   //: quantize
   // simple quantization into n levels
   // assumes and checks n>0.
   // We divide the interval [left,right) into a disjoint sum of
   // intervals of type [ ) and of equal length. For x in [left,right)
   // we return the number of the subinterval (thus a number in
   // {1,2,...n} to which x belongs.
   // For x==right, we also return n. For x outside of
   // [left,right] the return value is determined by the formula
   // 1+(Z)(n*(x-left)/(right-left)) for restrict==false
   // and 1 for x<=left , n for x>=right for restrict==true

Z quant(R x, Z n, bool restrict=false)const{ return qun(x,n,restrict);}
   // old name, for compatibility

R qunInv(Z n, IvZ const& iv)const;
```

```
   //: quantize inverse
   // returns y which is related to *this just as n is
   // related to iv.


X2<R2,Z2> intPolInfo(R x, Z n)const;
   // For the following explanation we assume the return value
   // being written as (R2(w1,w2),Z2(i1,i2)) giving us two
   // weights w1, w2 (wi>=0, w1+w2=1) and two indexes i1,i2
   // (1 <= i1 <= i2 <= n, i2-i1 \in {0,1}).
   // Consider a real-valued positive function f defined on
   // the interval *this. We consider the natural partition of
   // *this into n subintervals of equal length and we index them
   // in increasing order which gives us a well-defined list
   // I_1, ..., I_n of intervals. With interval I_i we associate
   // the real number f_i, by definition the mean value of function f
   // in I_i (i.e. the integral of f over I_i, divided by the length
   // of I_i). An important concrete situation is that the f_i are
   // pixel exposures and f an illuminance.
   // Then w1,w2,i1,i2 are set such that
   //       w1*f_i1 + w2*f_i2
   // is the most natural hypothesis for f(x). If x is outside
   // the interval, there is no problem: Then simply i1=i2, w1=w2=0.5.
   // More specificly: for x<inf() we have i1=i2=1
   // and for x>sup we have i1=i2=n.

R linear(R x, R yL=-1, R yR=1 )const{ return
   (left==right) ? 0.5*(yL+yR) : (yL*(right-x)+yR*(x-left))*absInv;}
   // Returns a linear function which takes the value yL at the left
   // end of the interval and the value yR at the right one.
   // If the interval has length 0, we return 0.5*(yL+yR)

Z stair(R x, Z n)const{ return 1+(Z)(n*ramp(x));}
   // returns a function which takes the value 0
   // at the left end of the interval and the value n+1 at the right
   // one. In between it is a monotone stair function taking the
   // values 1,2,...,n in succession, on intervals of equal width.

V<bool> bisection(R x, Z n)const;
   // returns n most significant bits (starting with the
   // utmost significant one) of the number ramp(x).
   // This is the explanation, the implementation is
   // different and aims at efficiency

R bisectionInv(const V<bool>& vb)const;
   // inversion; as accurate as the number of bits allows it
   // 02-10-02 tested with a few examples. Works !!!

R ranVal(Z j)const{ return ran(j);}
   // returns for each i \in Z a value in [left,right)
```

```
      // such that for each i0 \in Z the sequence
      // ranVal(i0),ranVal(i0+1),ranVal(i0+2),...
      // is random (I disagree with the philosophy behind saying
      // 'pseudo-random' in this place).

   R2 toR2()const{ return R2(left,right);}

   R ran(Z j=0)const;
      // returns a random number that lies in the interval *this.
      // Here the argument j plays the same role as in
      //    R CpmRoot::randomR(Z j=0);
      // (see cpmtypes.h)

   IvZ toIvZ()const{ return IvZ(left,right);}
};

////////////////////// class DiIv ////////////////////////////////////

class DiIv{ // directed interval in R
   // tool for letting two arbitrary (but not coinciding) points of
   // R play the role of the origin (x0) and of the unity (x1)
// independent data
   R x0,x1;
// dependent data
   R range,rangeInv;
// setting the dependent data
   void update();
public:
   typedef DiIv Type;
   CPM_IO
   explicit DiIv(Iv iv=Iv(0,1,true));
   explicit DiIv(R2 const& r2);
   DiIv(R x0_, R x1_);
   R p(R x)const{ return (x-x0)*rangeInv;}
      // function p returns something like the 'propoportional part' p
      // used in table interpolation; p=0 for x=x0 and p=1 for x=x1
   R x(R p_)const{ return x0+p_*range;}
      // returns a value x for which the 'proportional part' is p_
   Iv toIv()const{ return Iv(x0,x1,false);}
   R2 toR2()const{ return R2(x0,x1);}
   R operator[](Z i)const{ if (i<=1) return x0; else return x1;}
      // since DiIv,Iv and R2 are often to be used interchangingly
      // it is convenient to get to the data by the same syntax
};

//////////////////////////// proportions ////////////////////////////////

// Functions related to pairs of points, which are not assumed to
// be ordered according to size as in an interval
```

```
R proportion(R x1, R x2, R xd);
   // Look at x1,x2,xd as three points on the a staight line.
   // The position of xd with respect to the ordered pair (x1,x2)
   // of points may be described by a proportion ('Teilverhaeltnis'
   // in German). This is given by (xd-x1)/(x2-xd) so that has the value
   // takes the following special values (|huge|>>1>>|tiny|>0)
   // xd=x1          proportion=0
   // xd=(x1+x2)/2   proportion=1
   // xd=x2+tiny     proportion=huge
   // xd=huge        proportion=-1+tiny
   // The relative sign of huge and tiny there depends on the sign of
   // x2-x1.

R pointFromProportion(R x1, R x2, R prp);
   // Reconstructing a point from its proportion relative to x1 and x2:
   // For
   //       prp=proportion(x1,x2,xd)
   // we get
   //       xd=pointFromProportion(x1,x2,prp)
   // Conversely, for
   //       xd=pointFromProportion(x1,x2,prp), prp!=-1
   // we get
   //       prp=proportion(x1,x2,xd).

} // namespace

#endif
```

104 cpminterval.cpp

```
//? cpminterval.cpp
//? Status of work 2008-10-25.
//?
#include <cpminterval.h>
#include <cpmalgorithms.h>

using CpmGeo::Iv;
using CpmGeo::DiIv;

using CpmArrays::Vl;
using CpmArrays::V;
using CpmArrays::Vo;
using CpmArrays::X2;
using CpmArrays::IvZ;
using CpmFunctions::F;
using CpmFunctions::F2;
using CpmRoot::R;
using CpmRoot::Word;
using CpmRoot::Z;
using CpmLinAlg::R2;
using CpmLinAlg::Z2;
using namespace std;


//////////////////////////// class Iv ////////////////////////////////////

#define CPM_SHC\
   center=0.5*(left+right);\
   halfWidth=0.5*(right-left);\
   absInv=(halfWidth>0 ? 0.5/halfWidth : 0.);

void Iv::update(){
   CpmRootX::order<R>(left,right);
   CPM_SHC
}

void Iv::updateLight(){
   CPM_SHC
}

#undef CPM_SHC

Iv::Iv(const V<R>& vr)
{
   std::pair<R,R> p=Vo<R>(vr).infSup();
   left=p.first;
   right=p.second;
```

```
   updateLight();
}

Iv Iv::operator|(const Iv& iv)const
{
   R amin=CpmRootX::inf<R>(left,iv.left);
   R amax=CpmRootX::sup<R>(right,iv.right);
   return Iv(amin,amax,true); // fast return
}

Iv Iv::operator&(const Iv& iv)const
{
   if (right<=iv.left) return Iv(); // open intervals disjoint
   if (left>=iv.right) return Iv(); // open intervals disjoint
   R amin=CpmRootX::sup<R>(left,iv.left);
   R amax=CpmRootX::inf<R>(right,iv.right);
   return Iv(amin,amax,true); // fast return
}

bool Iv::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("Iv",str);
   cpmp(left);
   cpmp(right);
   return true;
}

bool Iv::scanFrom(istream& str)
{
   cpms(left);
   cpms(right);
   update();
   return true;
}

Z Iv::com(Iv const& iv)const
{
   if (left<iv.left) return 1;
   if (left>iv.left) return -1;
   if (right<iv.right) return 1;
   if (right>iv.right) return -1;
   return 0;
}

Vl<R> Iv::cen(Z n)const
{
   if (n<=0) cpmerror("Iv::cen(Z): n<=0");
   Vl<R> res(n);
   R d=(right-left)/n; // pixel width
   R xi=left+0.5*d;  // left+pixel half-width
```

```
   for (Z i=0;i<n;i++){
      res.cui(i)=xi;
      xi+=d;
   }
   return res;
}

V<R> Iv::div(Z n)const
{
   if (n<0) n=-n;
   V<R> res(n);
   if (n==0) return res;
   if (n==1){
      res[1]=center;
      return res;
   }
   res[1]=left;
   res[n]=right;
   if (n>2){
      R d=(right-left)/(n-1);
      R ri=left;
      for (Z i=2;i<n;i++){
         ri+=d;
         res[i]=ri;
      }
   }
   return res;
}

V<Iv> Iv::divide(Z n)const
{
   if (n<=0) cpmerror("Iv::subIntervals(Z): n<=0");
   V<Iv> res(n);
   R d=absVal()/n; // pixel width
   R bl=left;
   R br=bl+d;
   for (Z i=1;i<=n;i++){
      res[i]=Iv(bl,br,true);
      bl=br;
      br+=d;
   }
   return res;
}

const Iv& Iv::stretch(R k, Z s)
{
   R newRad=halfWidth*cpmabs(k);
   if (s==0){
      left=center-newRad;
      right=center+newRad;
```

```
   }
   else if(s==-1){
      right=left+2*newRad;
   }
   else if (s==1){
      left=right-2*newRad;
   }
   else{
      cpmerror("Iv::stretch: bad value of second argument");
   }
   update();
   return *this;
}

R Iv::periodic(R xx)const
{
   R av=absVal();
   if (av==0.){
      cpmerror("Iv::periodic(R)const called for left==right");
   }
   R y=(xx-left)*absInv;
   return y-floor(y);
}

R Iv::ramp(R xx)const
{
   if (xx<left) return 0.;
   else if (xx>right) return 1;
   else {
      if (left==right) return 0.5;
      else return (xx-left)*absInv;
   }
}

Z Iv::qun(R xx, Z n, bool restrict)const
{
   cpmassert(n>0,"Iv::qun(R,Z,bool)");
   if (restrict==false){
      if (xx==right) return n;
      else return 1+(Z)(n*(xx-left)*absInv);
   } // this was the only functionality till 2003-02-06
   else{ // new
      if (xx<=left) return 1;
      else if (xx>=right) return n;
      else return 1+(Z)(n*(xx-left)*absInv);
   }
}

R Iv::qunInv(Z n, IvZ const& iv)const
      //: quantize inverse
```

```
      // returns y which is related to *this just as n is
      // related to iv.
{
   if (iv.isVoid()) return center;
   Z rp=iv.relPos(n);
   if (rp==-1) return left;
   else if (rp==1) return right;
   else{ // now n belongs to
      Z dn=n-iv.first(); // varies from 0 to |iv|-1
      Z ni=iv.card();
      R dx=absVal()/ni;
      return (dn+0.5)*dx;
   }
}

X2<R2,Z2> Iv::intPolInfo(R xx, Z n)const
// only rater simple computations in each control path
// So, less complicated than it looks
{
   if (xx<=left) return X2<R2,Z2>(R2(0.5,0.5),Z2(1,1));
   if (xx>=right) return X2<R2,Z2>(R2(0.5,0.5),Z2(n,n));
   R r=n*(xx-left)*absInv; // takes value in [0,n)
   R rL=floor(r); // takes values in {0,....n-1}
   Z iA=1+(Z)rL; // takes values in {1,...,n}
   R qA=r-rL; // takes values [0,1). If this has value 0.5,
      // the interval indexed iA has dominant (actually
      // determining) influence, since we then are at the
      // center of this interval. For qA>0.5 the intervals
      // under consideration are indexed iA and iA+1
      // else iA and iA-1. The weight of this additional
      // index is |qA-0.5| and that of iA is 1-|aA-0.5|
   R wAdd=cpmabs(qA-0.5);
   R wA=1-wAdd;
   Z iAdd;
   if (qA<=0.5){
      iAdd=iA>1 ? iA-1 : iA;
      return X2<R2,Z2>(R2(wAdd,wA),Z2(iAdd,iA));
   }
   else{
      iAdd=iA < n ? iA+1 : iA;
      return X2<R2,Z2>(R2(wA,wAdd),Z2(iA,iAdd));
   }
}

V<bool> Iv::bisection(R xx, Z n)const
{
   Z mL=3;
   Word loc("Iv::bisection(R,Z)");
   cpmmessage(mL,loc&" started");
   cpmassert(n>0,loc);
```

```
   if (xx<left) xx=left;
   else if (xx>right) xx=right;
   else ;
   V<bool> res(n,false);
   R separator=center;
   R d=radius();
   for (Z i=1;i<=n;i++){
      d*=0.5;
      if (xx<=separator){
         separator-=d;
      }
      else{
         separator+=d;
         res[i]=true;
      }
   }
   cpmmessage(mL,loc&" done");
   return res;
}

R Iv::bisectionInv(const V<bool>& vb)const
{
   Z mL=3;
   Word loc("Iv::bisectionInv(V<bool>)");
   cpmmessage(mL,loc&" started");
   Z n=vb.dim();
   cpmassert(n>0,loc);
   R res=left;
   R d=absVal();
   for (Z i=1;i<=n;i++){
      d*=0.5;
      if (vb[i]) res+=d;
   }
   cpmmessage(mL,loc&" done");
   return res;
}

namespace{
   R fAux(const R& q, const R& b, const Z& n)
      // always defined unless ::pow(q,n-2) overflows
   {
      return q!=1. ? b-(1.-::pow(q,n-1))/(1-q): b-n+1;
   }
}

V<R> Iv::divGeoProg(R dt,Z n)const
{
   Z mL=4;
   Word loc("V<R> Iv::divGeoProg(R dt,Z n)");
   cpmmessage(mL,loc&" started");
```

```
   cpmassert(n>2,loc);
   cpmassert(dt>0,loc);
   R av=absVal();
   cpmassert(dt<av,loc);
   V<R> res(n);
   R b=av/dt;
   F<R,R> f=F2<R,R,Z,R>(b,n)(fAux);
   R qPrev=0.;
   R fPrev=f(qPrev);
   cpmassert(fPrev>0,loc);
   // lean and fast root finding since the function is
   // known to fall with increasing speed
   R qAct=0.1;
   R fAct=f(qAct);
   R incrFactor=2;
   R acc=1e-6;
   while (fAct>0){
      qPrev=qAct;
      qAct*=incrFactor;
      fAct=f(qAct);
   }
   R q=CpmAlgorithms::rootByBisection(f,qPrev,qAct,acc);
   res[1]=inf();
   res[n]=sup();
   R step=dt;
   res[2]=res[1]+step;
   if (n==3){
      return res;
   }
   for (Z i=3;i<=n;i++){
      step*=q;
      res[i]=res[i-1]+step;
   }
   cpmmessage(mL,loc&" done");
   return res;
}

namespace{
   V<R> equalize(const V<R>& v)
   {
      static R i3=1./3.;
      Z nSup=v.dim()-2;
      V<R> res=v;
      for (Z i=3;i<=nSup;i++){
         res[i]=(v[i-1]+v[i]+v[i+1])*i3;
      }
      return res;
   }
}
```

```
V<R> Iv::divSmooth(R dti,R dtf,Z n)const
{
   Z mL=4;
   Word loc("V<R> Iv::divGeoProg(R dti,dtf,Z n)");
   cpmmessage(mL,loc&" started");
   cpmassert(n>3,loc);
   cpmassert(dti>0,loc);
   cpmassert(dtf>0,loc);
   R av=absVal();
   R gap=av-dti-dtf;
   cpmassert(gap>0,loc);
   V<R> res(n);
   res[1]=inf();
   res[2]=res[1]+dti;
   res[n]=sup();
   res[n-1]=res[n]-dtf;
   Z i;
   if (n==4){
      return res;
   }
   else{
      Z p=n-4; // these divide gap into p+1 intervals (notice that the
      // end points of gap are not among the p points!
      // of length gap/(p+1)
      R gapd=gap/(p+1);
      for (i=1;i<=p;i++) res[i+2]=res[i+1]+gapd;
   }
   Z nSteps=n;
   for (i=1;i<=nSteps;i++)  res=equalize(res);
   cpmmessage(mL,loc&" done");
   return res;
}


R Iv::ran(Z j)const
{
   R xx=CpmRoot::randomR(j);
   return left+xx*(right-left);
}


////////// functions concerning proportions ////////////////////////////

R CpmGeo::proportion(R x1, R x2, R xd)
{
   R nominator=x2-xd;
   cpmassert(nominator!=0,"CpmGeo::proportion");
  // return (xd-x2)/nominator;
   return (xd-x1)/nominator; // correction 2005-06-22
}
```

```
R CpmGeo::pointFromProportion(R x1, R x2, R prp)
{
   R nominator=1+prp;
   cpmassert(nominator!=0,"CpmGeo::pointFromProportion");
   return (x1+prp*x2)/nominator;
}

//////////////////////// class DiIv ////////////////////////////////////////

void DiIv::update()
{
   CpmRoot::Word loc("DiIv::update()");
   range=x1-x0;
   cpmassert(range!=0,loc);
   rangeInv=1./range;
}

DiIv::DiIv(Iv iv):x0(iv.inf()),x1(iv.sup())
{
   update();
}

DiIv::DiIv(R a, R b):x0(a),x1(b)
{
   update();
}

DiIv::DiIv(R2 const& r2):x0(r2[1]),x1(r2[2])
{
   update();
}

bool DiIv::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("DiIv",str);
   cpmp(x0);
   cpmp(x1);
   return true;
}

bool DiIv::scanFrom(istream& str)
{
   cpms(x0);
   cpms(x1);
   update();
   return true;
}
```

105 cpmlinalg.h

```
//? cpmlinalg.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_LINALG_H_
#define CPM_LINALG_H_
/*

   Description: classes R1, R2, R3, R4, RR1, RR2, RR3, RR4
   Basis for effective geometry in 1,2,3,4-dimensional spaces.
   Defines linear spaces as non-template classes without virtual
   functions.
   Only standard C++.   C++ types used:
      void, bool, istream&, ostream&
   Metric properties of linear spaces like scalar product, and absolute
   value are provided. Dimension dependent concepts like vector product
   are provided for R2 and R3.

   Component access via indexes is save but out-of-range indexes silently
   access the nearest component.

*/

#include <cpmtypes.h>
#include <cpmv.h>
#include <cpmc.h>

namespace CpmLinAlg{

   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmRoot::C;
   using CpmRoot::Word;
   using CpmRootX::R1;
   using CpmArrays::V;
   using CpmArrays::X2;

////////// Tuples of 2,3,4 components of type R, defining linear algebra.

////////////////////////////// class R2 ////////////////////////////////

class R2{  //  tuples of two real elements
public:
   R x1;
   R x2;
   typedef R2 Type;

// constructors
```

```
   R2():x1(0),x2(0){}
   R2(const R& a, const R& b):x1(a),x2(b){}
   R2(const R& a, const R& b, const R& c):x1(a),x2(b){c;}
      // for formal uniformity to R3, third argument is
      // passive. Introduced 2004-04-29
explicit R2(const V<R>&);
      // construction from dimension-independent lists
      // no restiction to the argument
// access
explicit R2(C z):x1(z[1]),x2(z[2]){}
   C toC()const{ return C(x1,x2);}
   R& operator[](Z i){ if (i>=2) return x2; else return x1;}
   const R& operator[](Z i)const{ if (i>=2) return x2; else return x1;}
      // 'else' should not be needed. Was intoduced in 2001-7-27 in order
      // to avoid malfunction in Dim2::Spc::toC() which, then, was based
      // on Dim2::Vec::operator[], Dim2::Vec derived from R2.
      // now also operator[] for Z2,R3,R4,C2,C3,C4 is defined by a
      // complete if, else if, ...else if,else selection statement
   virtual R operator()(Z i)const
      {if (i<1 || i>2) return 0; else return (*this)[i]; }
   void set(R t){ x1=t; x2=t;}
// mutating arithmetics (linear algebra)
   R2& operator +=(const R2& a)
   { x1+=a.x1; x2+=a.x2; return *this;}
   R2& operator -=(const R2& a)
   { x1-=a.x1; x2-=a.x2; return *this;}
   R2& operator *=(const R& r)
   { x1*=r; x2*=r; return *this;}
   R2& neg(void)
   { x1=-x1; x2=-x2; return *this;  }

   R operator |(const R2& a)const
   { return x1*a.x1+x2*a.x2;}
   R crsPrd(const R2& a)const{ return x1*a.x2-x2*a.x1;}
      // related to cross product (vector product)
   R absVal(bool careful=true)const;
   R absSqr()const{return x1*x1+x2*x2;}
   R meanValue()const{ return (x1+x2)*0.5;}
   R dist(R2 const& a)const;
      // distance
   R distSquare(R2 const& a)const;
      // square of distance
   R disVal(R2 const&)const;
   R sup()const{ return x1<x2 ? x2:x1;}
   R inf()const{ return x1<x2 ? x1:x2;}
   R mod()const{ R a=absVal(); return a==0 ? 0. : (sup()-inf())/a;}
      // modulation, measures the deviation from constancy
   R2 operator~()const{ return R2(x2,x1);}
   X2<R,R2 > polDec()const;
      // polar decomposition: First component (res.first if res is the
```

```
      // return value) is the absolute value r and the second component
      // a unit vector n such that *this = n*r. If *this is the zero
      // vector, n is arbitrarily chosen as (1,0)
   R2 carToPol()const;
      // Cartesian coordinates (x1,x2) to polar coordinates.
      // Returns R2(r,phi) with r=absVal(), phi=::atan2(x2,x1)).
      // thus the phi-range is that given by the systems atan2 function.
      // Needless to say that angles are here treated as arcs of
      // unit circles of type R. The class CpmGeo::Angle is not
      // used in this file.
   R2 polToCar()const;
      // Polar coordinates (r,phi) to  Cartesian coordinates (x1,x2).
      // Returns R2(x1*cos(x2),x1*sin(x2))
   R normalize()
      { X2<R,R2> p=polDec(); *this=p.c2();  return p.c1();}
      // turns *this into a unit vector and returns the norm of the
      // original in order not to loose information
   R normalize_(R r=1);
      // turns *this into a vector of length r and returns the norm of
      // the original in order not to loose information. If *this was
      // the zero vector, it will be turned into  a 'standard vector'
      // of length r is
   R nor_(R r=1){ return normalize_(r);}
      //: normalize
      // C+- conformant name
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 2;}
   bool isVal()const
   { return CpmRoot::isVal(x1)&&CpmRoot::isVal(x2);}

   R2 ranVal(Z j=0)const;
   virtual Word nameOf()const{ return Word("R2");}
   static V<R2> canBas(){ return V<R2>("",R2(1,0),R2(0,1));}
      // canonical basis
};

inline R2 operator +(const R2& x1, const R2& x2)
{ R2 res=x1; return res+=x2;}


inline R2 operator -(const R2& x1, const R2& x2)
{ R2 res=x1; return res-=x2;}

inline R2 operator -(const R2& x)
{ R2 res=x; return res.neg();}


inline R2 operator *(const R2& x1, const R& s)
{ R2 res=x1; return res*=s;}
```

```
inline R2 operator *(const R& s, const R2& x1)
{ R2 res=x1; return res*=s;}

inline R R2::distSquare(R2 const& a)const{ return (*this-a).absSqr();}

inline R R2::dist(R2 const& a)const{ return (*this-a).absVal();}

//////////////////////////// class Z2 ////////////////////////////////

class Z2{  //  tuples of two integer elements
public:
   Z x1;
   Z x2;
public:
   typedef Z2 Type;

// constructors
   Z2():x1(0),x2(0){}
   Z2(const Z& a, const Z& b):x1(a),x2(b){}
explicit Z2(const V<Z>&);
      // construction from dimension-independent lists
      // no restiction to the argument
// access
   Z& operator[](Z i){ if (i>=2) return x2; else return x1;}
   const Z& operator[](Z i)const{ if (i>=2) return x2; else return x1;}
   void set(Z t){ x1=t; x2=t;}
// mutating arithmetics (linear algebra)
   Z2& operator +=(const Z2& a)
   { x1+=a.x1; x2+=a.x2; return *this;}
   Z2& operator -=(const Z2& a)
   { x1-=a.x1; x2-=a.x2; return *this;}
   Z2& operator *=(const Z& r)
   { x1*=r; x2*=r; return *this;}
   Z2& neg(void)
   { x1=-x1; x2=-x2; return *this;  }
   Z operator |(const Z2& a)const
   { return x1*a.x1+x2*a.x2;}
   R absVal(bool careful=true)const;
   Z absSqr()const{return x1*x1+x2*x2;}
   Z2 operator~()const{ return Z2(x2,x1);}
   Z sup()const{ return x1<x2 ? x2:x1;}
   Z inf()const{ return x1<x2 ? x1:x2;}
   R disVal(const Z2&)const;
   X2<R,R2 > polDec()const;
      // polar decomposition: First component (res.first if res is the
      // return value) is the absolute value r and the second component
      // a unit vector n such that *this = n*r. If *this is the zero
      // vector, n is arbitrarily chosen as (1,0)
   CPM_IO
```

```
   CPM_ORDER
   CPM_WORD
   Z dim()const{ return 2;}

   bool isVal()const
   { return CpmRoot::isVal(x1)&&CpmRoot::isVal(x2);}
   Word nameOf()const{ return Word("Z2");}
};

inline Z2 operator +(const Z2& x1, const Z2& x2)
{ Z2 res=x1; return res+=x2;}

inline Z2 operator -(const Z2& x1, const Z2& x2)
{ Z2 res=x1; return res-=x2;}

inline Z2 operator -(const Z2& x)
{ Z2 res=x; return res.neg();}

inline Z2 operator *(const Z2& x1, const Z& s)
{ Z2 res=x1; return res*=s;}

inline Z2 operator *(const Z& s, const Z2& x1)
{ Z2 res=x1; return res*=s;}

//////////////////////////// class R3 ////////////////////////////////

class R3{  //  tuples of 3 real elements
public:
   typedef R3 Type;
   R x1;
   R x2;
   R x3;
public:
// constructors
   R3():x1(0.),x2(0.),x3(0.){}
   R3(const R& a, const R& b, const R& c):x1(a),x2(b),x3(c){}
   R3(const R& a, const R& b):x1(a),x2(b),x3(0.){}
      // added 02-09-07 to have common constructor for R2 and R3
      // which in some code is Rdim
explicit R3(const V<R>&); // construction from dimension-independent
// lists, no restiction to the argument.
// access
   R& operator[](Z i){ if (i>=3) return x3; else if (i==2) return x2;
    else return x1;}
   const R& operator[](Z i)const{ if (i>=3) return x3; else if (i==2)
    return x2; else return x1;}
   virtual R operator()(Z i)const
      {if (i<1 || i>3) return 0; else return (*this)[i]; }
   void set(R t){ x1=t; x2=t; x3=t;}
// mutating arithmetics (linear algebra)
```

```
R3& operator +=(const R3& a)
{ x1+=a.x1; x2+=a.x2; x3+=a.x3; return *this;}
R3& operator -=(const R3& a)
{ x1-=a.x1; x2-=a.x2; x3-=a.x3; return *this;}
R3& operator *=(const R& r)
{ x1*=r; x2*=r; x3*=r; return *this;}
R3& neg(void)
{ x1=-x1; x2=-x2; x3=-x3; return *this;  }
R operator |(const R3& a)const
{ return x1*a.x1+x2*a.x2+x3*a.x3;}

R3 crsPrd(R3 const& a)const
   // cross product (vector product)
{ return  R3(x2*a.x3-x3*a.x2,
             x3*a.x1-x1*a.x3,
             x1*a.x2-x2*a.x1); }

R absVal(bool careful=true)const;
R absSqr()const{return x1*x1+x2*x2+x3*x3;}
R meanValue()const{ return (x1+x2+x3)/3;}
R dist(R3 const& a)const;
   // distance
R distSquare(R3 const& a)const;
   // square of distance
R disVal(R3 const&)const;
R sup()const{ return CpmRootX::sup<R>(x1,x2,x3);}
R inf()const{ return CpmRootX::inf<R>(x1,x2,x3);}
R mod()const{ R a=absVal(); return a==0 ? 0. : (sup()-inf())/a;}
   // modulation, measures the deviation from constancy
X2<R,R3> polDec()const;
   // polar decomposition: First component (res.first if res is the
   // return value) is the absolute value r and the second component
   // a unit vector n such that *this = n*r. If *this is the zero
   // vector, n is arbitrarily chosen as (1,0,0)

R3 carToPol()const;
   // Cartesian coordinates (x1,x2,x3) to polar coordinates
   // (r,phi,delta) where delta is the angle relative to the 1-2-plane
   // as is declination with respect to the equator plane in
   // astronomy.
   // This is in most circumstances more convenient and symmetric
   // than a polar angle theta measuring the angle against the
   // 3-direction. The precise definition is:
   // r=absVal(),
   // phi=R2(x1,x2).carToPol()[2]
   // delta=R2(rho,x3).carToPol()[2]
   // where  rho:=R2(x1,x2).carToPol()[1]
   // It turned out that taken this as implementation causes dark
   // behavior with MS C++ 6.0. Now we express the same scheme
   // in a form that does not call R2::functions.
```

```
   R3 polToCar()const;
      // Polar coordinates to  Cartesian coordinates.
      // Returns R3(rho*cos(x2),rho*sin(x2),x1*sin(x3)),
      // where rho:=x1*cos(x3)

   R normalize()
      { X2<R,R3> p=polDec(); *this=p.c2();  return p.c1();}
      // turns *this into a unit vector and returns the norm of the
      // original in order not to loose information

   R normalize_(R r=1);
      // turns *this into a vector of length r and returns the norm of
      // the original in order not to loose information. If *this was
      // the zero vector, it will be turned into  a 'standard vector'
      // of length r is

   R nor_(R r=1){ return normalize_(r);}
      //: normalize
      // C+- conformant name

   CPM_IO
   CPM_ORDER
   CPM_WORD
   Z dim()const{ return 3;}
   bool isVal()const
   { return CpmRoot::isVal(x1)&&CpmRoot::isVal(x2)
      &&CpmRoot::isVal(x3);}

   R3 ranVal(Z j=0)const;
   virtual Word nameOf()const{ return Word("R3");}
   static V<R3> canBas()
      { return V<R3>("",R3(1,0,0),R3(0,1,0),R3(0,0,1));}
      // canonical basis

// operations as used in my publication: 'Rigidly connected spherical
// particles: a versatile grain model' Granular Matter 2006
   R3 ast(R3 const& x)const; // ast for asterisk, \ast in LaTex
      // returns (*this)*x according to equation (6)
   R3 circ(R3 const& r)const; // circ for \circ of LaTex
      // returns (*this)o r according to equation (7)
   R3 exp()const // see equation (11) of paper cited above
   {
      R3 res=*this;
      R av=absVal();
      if (av==0) return res;
      R fac=tan(av*0.5)/av;
      return res*=fac;
   }
```

```
   R3 ln()const // inverse function to exp
   {
      R3 res=*this;
      R av=absVal();
      if (av==0) return res;
      R fac=2*atan(av)/av;
      return res*=fac;
   }

};

inline R3 operator +(const R3& x1, const R3& x2)
{ R3 res=x1; return res+=x2;}

inline R3 operator -(const R3& x1, const R3& x2)
{ R3 res=x1; return res-=x2;}

inline R3 operator -(const R3& x)
{ R3 res=x; return res.neg();}

inline R3 operator *(const R3& x1, const R& s)
{ R3 res=x1; return res*=s;}

inline R3 operator *(const R& s, const R3& x1)
{ R3 res=x1; return res*=s;}

inline R R3::dist(R3 const& a)const{ return (*this-a).absVal();}

inline R R3::distSquare(R3 const& a)const{ return (*this-a).absSqr();}

//////////////////////////// class Z3 ////////////////////////////////
// placed behind R3 since type R3 is needed in X2<R,R3 > polDec()const;

class Z3{  // tuples of three integer elements
public:
   Z x1;
   Z x2;
   Z x3;
public:
   typedef Z3 Type;

// constructors
   Z3():x1(0),x2(0),x3(0){}
   Z3(const Z& a, const Z& b, const Z& c):x1(a),x2(b),x3(c){}
   Z3(const Z& a, const Z& b):x1(a),x2(b),x3(0){}
      // added 02-09-07 treat Z3 like R3
explicit Z3(const V<Z>&); // construction from dimension-independent
// lists
      // no restiction to the argument
// access
```

```
   Z& operator[](Z i){ if (i>=3) return x3; else if (i==2) return x2;
    else return x1;}
   const Z& operator[](Z i)const{ if (i>=3) return x3; else if (i==2)
    return x2; else return x1;}
   void set(Z t){ x1=t; x2=t; x3=t;}
// mutating arithmetics (linear algebra)
   Z3& operator +=(const Z3& a)
   { x1+=a.x1; x2+=a.x2; x3+=a.x3; return *this;}
   Z3& operator -=(const Z3& a)
   { x1-=a.x1; x2-=a.x2; x3-=a.x3; return *this;}
   Z3& operator *=(const Z& r)
   { x1*=r; x2*=r; x3*=r; return *this;}
   Z3& neg(void)
   { x1=-x1; x2=-x2; x3=-x3; return *this;}
   Z operator |(const Z3& a)const
   { return x1*a.x1+x2*a.x2+x3*a.x3;}
   R absVal(bool careful=true)const;
   Z absSqr()const{return x1*x1+x2*x2+x3*x3;}
   R disVal(const Z3&)const;
   X2<R,R3 > polDec()const;
      // polar decomposition: First component (res.first if res is the
      // return value)
      // is the absolute value r and the second component a unit vector n
      // such that
      // *this = n*r. If *this is the zero vector, n is arbitrarily
      // chosen
      // as (1,0)
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 3;}
   bool isVal()const
   { return CpmRoot::isVal(x1)&&CpmRoot::isVal(x2)
      &&CpmRoot::isVal(x3);}
   Word nameOf()const{ return Word("Z3");}
};

inline Z3 operator +(const Z3& x1, const Z3& x2)
{ Z3 res=x1; return res+=x2;}

inline Z3 operator -(const Z3& x1, const Z3& x2)
{ Z3 res=x1; return res-=x2;}

inline Z3 operator -(const Z3& x)
{ Z3 res=x; return res.neg();}

inline Z3 operator *(const Z3& x1, const Z& s)
{ Z3 res=x1; return res*=s;}

inline Z3 operator *(const Z& s, const Z3& x1)
{ Z3 res=x1; return res*=s;}
```

```
//////////////////////////// class R4 ////////////////////////////////

class R4{  //  tuples of 4 real elements
public:
   typedef R4 Type;
   R x1;
   R x2;
   R x3;
   R x4;
public:
// constructors
   R4():x1(0),x2(0),x3(0),x4(0){}
   R4(const R& a, const R& b, const R& c, const R& d):x1(a),x2(b),
      x3(c),x4(d){}
explicit R4(const V<R>&); // construction from dimension-independent
// lists
      // no restiction to the argument
// access
   R& operator[](Z i)
   { if (i>=4) return x4; else if (i==3) return x3;
      else if (i==2) return x2; else return x1;}
   const R& operator[](Z i)const
   { if (i>=4) return x4; else if (i==3) return x3;
      else if (i==2) return x2; else return x1;}
   virtual R operator()(Z i)const
      {if (i<1 || i>4) return 0; else return (*this)[i]; }
   void set(R t){ x1=t; x2=t; x3=t; x4=t;}
// mutating arithmetics (linear algebra)
   R4& operator +=(const R4& a)
   { x1+=a.x1; x2+=a.x2; x3+=a.x3; x4+=a.x4; return *this;}
   R4& operator -=(const R4& a)
   { x1-=a.x1; x2-=a.x2; x3-=a.x3; x4-=a.x4; return *this;}
   R4& operator *=(const R& r)
   { x1*=r; x2*=r; x3*=r; x4*=r; return *this;}
   R4& neg(void)
   { x1=-x1; x2=-x2; x3=-x3; x4=-x4; return *this; }
   R operator |(const R4& a)const
   { return x1*a.x1+x2*a.x2+x3*a.x3+x4*a.x4;}
   R absVal(bool careful=true)const;
   R absSqr()const{return x1*x1+x2*x2+x3*x3+x4*x4;}
   R meanValue()const{ return (x1+x2+x3+x4)*0.25;}
   R dist(R4 const& a)const;
      // distance
   R distSquare(R4 const& a)const;
      // square of distance
   R disVal(const R4&)const;
   R sup()const{ return CpmRootX::sup<R>(x1,x2,x3,x4);}
   R inf()const{ return CpmRootX::inf<R>(x1,x2,x3,x4);}
   R mod()const{ R a=absVal(); return a==0 ? 0. : (sup()-inf())/a;}
```

```
      // modulation, measures the deviation from constancy
   X2<R,R4 > polDec()const;
      // polar decomposition: First component (res.first if res is the
      // return value) is the absolute value r and the second component
      // a unit vector n such that *this = n*r. If *this is the zero
      // vector, n is arbitrarily chosen as (1,0,0,0)
   R normalize(){ X2<R,R4> p=polDec(); *this=p.second();
      return p.first();}
      // turns *this into a unit vector and returns the norm of the
      // original in order not to loose information

   R normalize_(R r=1);
      // turns *this into a vector of length r and returns the norm of
      // the original in order not to loose information. If *this was
      // the zero vector, it will be turned into  a 'standard vector'
      // of length r is

   CPM_IO
   CPM_ORDER
   CPM_WORD
   Z dim()const{ return 4;}
   bool isVal()const
   { return CpmRoot::isVal(x1)&&CpmRoot::isVal(x2)
      &&CpmRoot::isVal(x3)&&CpmRoot::isVal(x4);}

   R4 ranVal(Z j=0)const;
   virtual Word nameOf()const{ return Word("R4");}
   static V<R4> canBas()
   { return V<R4>("",R4(1,0,0,0),R4(0,1,0,0),R4(0,0,1,0),R4(0,0,0,1));}
      // canonical basis
};

inline R4 operator +(const R4& x1, const R4& x2)
{ R4 res=x1; return res+=x2;}

inline R4 operator -(const R4& x1, const R4& x2)
{ R4 res=x1; return res-=x2;}

inline R4 operator -(const R4& x)
{ R4 res=x; return res.neg();}

inline R4 operator *(const R4& x1, const R& s)
{ R4 res=x1; return res*=s;}

inline R4 operator *(const R& s, const R4& x1)
{ R4 res=x1; return res*=s;}

inline R R4::dist(R4 const& a)const{ return (*this-a).absVal();}

inline R R4::distSquare(R4 const& a)const{ return (*this-a).absSqr();}
```

```
//////////////////////////// class Z4 ////////////////////////////////
// placed behind R4 since type R4 is needed in X2<R,R4> polDec()const;

class Z4{  //  tuples of 4 integer elements
public:
   typedef Z4 Type;
   Z x1;
   Z x2;
   Z x3;
   Z x4;
public:
// constructors
   Z4():x1(0),x2(0),x3(0),x4(0){}
   Z4(const Z& a, const Z& b, const Z& c, const Z& d):x1(a),x2(b),
      x3(c),x4(d){}
explicit Z4(const V<Z>&); // construction from dimension-independent
// lists
      // no restiction to the argument
// access
   Z& operator[](Z i)
   { if (i>=4) return x4; else if (i==3) return x3;
      else if (i==2) return x2; else return x1;}
   const Z& operator[](Z i)const
   { if (i>=4) return x4; else if (i==3) return x3;
      else if (i==2) return x2; else return x1;}
   void set(Z t){ x1=t; x2=t; x3=t; x4=t;}
// mutating arithmetics (linear algebra)
   Z4& operator +=(const Z4& a)
   { x1+=a.x1; x2+=a.x2; x3+=a.x3; x4+=a.x4; return *this;}
   Z4& operator -=(const Z4& a)
   { x1-=a.x1; x2-=a.x2; x3-=a.x3; x4-=a.x4; return *this;}
   Z4& operator *=(const Z& r)
   { x1*=r; x2*=r; x3*=r; x4*=r; return *this;}
   Z4& neg(void)
   { x1=-x1; x2=-x2; x3=-x3; x4=-x4; return *this; }
   Z operator |(const Z4& a)const
   { return x1*a.x1+x2*a.x2+x3*a.x3+x4*a.x4;}
   R absVal(bool careful=true)const;
   Z absSqr()const{return x1*x1+x2*x2+x3*x3+x4*x4;}
   R disVal(const Z4&)const;
   X2<R,R4 > polDec()const;
      // polar decomposition: First component (res.first if res is the
      // return value) is the absolute value r and the second component
      // a unit vector n such that *this = n*r. If *this is the zero
      // vector, n is arbitrarily chosen as (1,0,0,0)
   CPM_IO
   CPM_ORDER
   CPM_WORD
   Z dim()const{ return 4;}
```

```
   bool isVal()const
   { return CpmRoot::isVal(x1)&&CpmRoot::isVal(x2)
      &&CpmRoot::isVal(x3)&&CpmRoot::isVal(x4);}
   Word nameOf()const{ return Word("Z4");}
};

inline Z4 operator +(const Z4& x1, const Z4& x2)
{ Z4 res=x1; return res+=x2;}

inline Z4 operator -(const Z4& x1, const Z4& x2)
{ Z4 res=x1; return res-=x2;}

inline Z4 operator -(const Z4& x)
{ Z4 res=x; return res.neg();}

inline Z4 operator *(const Z4& x1, const Z& s)
{ Z4 res=x1; return res*=s;}

inline Z4 operator *(const Z& s, const Z4& x1)
{ Z4 res=x1; return res*=s;}

/*************** lean square matrices (1*1) - (4*4) *******************/

//////////////////////////// class RR1 ////////////////////////////////

#define CPM_MAT_ALG(X,XX)\
friend X operator*(const XX&, const X&);\
friend XX operator*(const XX& , const XX&);\
friend XX operator+(const XX& , const XX& );\
friend XX operator~(const XX& );\
friend X operator*(const X& x, const XX& a){ return ~a*x;}

// comment to the last function where matrices are acting on vectors
// from the right side. This kind of action on a vector space gives a
// representation of the multiplicative monoid of matrices if the product
// law for endomorphisms of linear spaces is defined as contravariant.
// In our specific notation here:
// x*(a*b)=(x*a)*b
// Proof: x*(a*b)=~(a*b)*x=((~b)*(~a))*x=(~b)*((~a)*x)=((~a)*x)*b=(x*a)*b

class RR1{  //  real 1*1 matrices
public:
   typedef RR1 Type;
   R1 x1;
public:
   static RR1 zero;
   static RR1 unit;
// constructors
   RR1(){}
   explicit RR1(const R1& a1):x1(a1){} // no automatic conversion wanted
```

```
   // !
// access
   R1& operator[](Z i){i; return x1;}
   const R1& operator[](Z i)const{i; return x1;}
   CPM_IO
   CPM_WORD
   CPM_MAT_ALG(R1,RR1)
   Word nameOf()const{ return Word("RR1");}
};

///////////////////////////// class RR2 ////////////////////////////////

class RR2{  //  real 2*2 matrices
   // (*this)[i][k] is just xi.xk
   // the 2*2 matrices get special treatment in order to do
   // justice to the group SL(2,R)
public:
   typedef RR2 Type;
   R2 x1;
   R2 x2;
public:
   static RR2 zero;
   static RR2 unit;
// constructors
   RR2(){}
   RR2(const R2& a1, const R2& a2):x1(a1),x2(a2){}
   RR2(R z11, R z12, R z21, R z22):x1(z11,z12),x2(z21,z22){}
// access
   R2& operator[](Z i){ if (i>=2) return x2; return x1;}
   const R2& operator[](Z i)const{ if (i>=2) return x2; return x1;}
   R det()const{ return x1.x1*x2.x2-x1.x2*x2.x1;}
   RR2 operator !()const;
      // inverse matrix can be expressed easily in this case
   R operator *(R z)const
   { return (x1.x1*z+x1.x2)*CpmRoot::inv(x2.x1*z+x2.x2);}
      // RR2 acts on R as a 'broken linear transformation'
   CPM_IO
   CPM_WORD
   CPM_MAT_ALG(R2,RR2)
   Word nameOf()const{ return Word("RR2");}
};

///////////////////////////// class RR3 ////////////////////////////////

class RR3{  //  real 3*3 matrices
public:
   typedef RR3 Type;
   R3 x1;
   R3 x2;
   R3 x3;
```

```
public:
   static RR3 zero;
   static RR3 unit;
// constructors
   RR3(){}
   RR3(const R3& a1, const R3& a2, const R3& a3):x1(a1),x2(a2),x3(a3){}
// access
   R3& operator[](Z i)
   { if (i>=3) return x3; if (i==2) return x2; return x1;}

   const R3& operator[](Z i)const
   { if (i>=3) return x3; if (i==2) return x2; return x1;}

   CPM_IO
   CPM_WORD
   CPM_MAT_ALG(R3,RR3)
   Word nameOf()const{ return Word("RR3");}
};

//////////////////////////// class RR4 ////////////////////////////////

class RR4{  //  real 4*4 matrices
public:
   typedef RR4 Type;
   R4 x1;
   R4 x2;
   R4 x3;
   R4 x4;
public:
   static RR4 zero;
   static RR4 unit;
// constructors
   RR4(){}
   RR4(const R4& a1, const R4& a2, const R4& a3, const R4& a4):
      x1(a1),x2(a2),x3(a3),x4(a4){}
// access
   R4& operator[](Z i)
      { if (i>=4) return x4; if (i==3) return x3;
         if (i==2) return x2; return x1;}

   const R4& operator[](Z i)const
       { if (i>=4) return x4; if (i==3) return x3;
         if (i==2) return x2; return x1;}

   CPM_IO
   CPM_WORD
   CPM_MAT_ALG(R4,RR4)
   Word nameOf()const{ return Word("RR4");}
};
```

```
#undef CPM_MAT_ALG

} // namespace

#endif
```

106 cpmlinalg.cpp

```
//? cpmlinalg.cpp
//? Status of work 2008-10-25.
//?
#include <cpmlinalg.h>

using namespace CpmRoot;
using namespace CpmLinAlg;

using CpmRoot::distFunc;
using CpmArrays::X2;

//#define CPM_NOT_KILL_OPTIMIZER
   // this avoids a reasonable definition of multiplication of
   // 3*3 and 4*4 matrices in order not to overload the standard
   // full optimizer for release versions made by the Microsoft
   // Visual C++ compiler (it never comes to an end !!!!)

//////////////////////////// class R2 ////////////////////////////////////

namespace{

R pytCom(R x1, R x2, bool careful)
{
   if (careful==false) return ::sqrt(x1*x1+x2*x2);
   R y1=(x1>=0 ? x1 : -x1);
   R y2=(x2>=0 ? x2 : -x2);
   R y=y1+y2;
   if (y==0) return 0;
   R yInv=1./y;
   y1*=yInv;
   y2*=yInv;
   return y*::sqrt(y1*y1+y2*y2);
}

}

R2::R2(const V<R>& v)
{
   Z d=v.dim();
   x1= (1<=d ? v[1] : 0);
   x2= (2<=d ? v[2] : 0);
}

bool R2::prnOn(ostream& out)const
{
   if (!CpmRoot::writeTitle("R2",out)) return false;
   if (!CpmRoot::write(x1,out)) return false;
```

```
   if (!CpmRoot::write(x2,out)) return false;
   return true;
}

bool R2::scanFrom(istream& in)
{
   if (!CpmRoot::read(x1,in)) return false;
   if (!CpmRoot::read(x2,in)) return false;
   return true;
}

Z R2::com(const R2& a)const
{
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   if (x2<a.x2) return 1;
   if (x2>a.x2) return -1;
   return 0;
}

R R2::absVal(bool careful)const
{
   return pytCom(x1,x2,careful);
}

R R2::disVal(const R2& x)const
{
   R a=absVal();
   R b=x.absVal();
   R c=(*this-x).absVal();
   return distFunc(a,b,c);
}

X2<R,R2 > R2::polDec()const
{
   R r=absVal(true);
   R2 pol=*this;
   if (r!=0){
     R rInv=1./r; pol*=rInv;
     return X2<R,R2 >(r,pol);
   }
   else{
      return X2<R,R2 >(0.,R2(1.,0.));
   }
}

R R2::normalize_(R r)
{
   X2<R,R2> p=polDec(); R r0=p.first();
   *this=p.second()*r;  return r0;
```

```
   // was wrong till 2004-11-05
}

R2 R2::carToPol()const
{
   return R2(absVal(),::atan2(x2,x1));
}

R2 R2::polToCar()const
{
   return R2(x1*::cos(x2),x1*::sin(x2));
}

#define CPM_SC\
   Type res;\
   R ri;\
   Z d=res.dim(),i;\
   if (j==0){\
      for (i=1; i<=d; i++){\
         ri=(*this)[i];\
         res[i]=CpmRoot::ranVal(ri,j);\
      }\
   }\
   else{\
      Z j0=1000;\
      Z jIncr=CpmRoot::ranVal(j0,j);\
      Z jc=j;\
      for (i=1; i<=d; i++){\
         ri=(*this)[i];\
         jc+=jIncr;\
         res[i]=CpmRoot::ranVal(ri,jc);\
      }\
   }\
   return res


R2 R2::ranVal(Z j)const{ CPM_SC;}


//////////////////////////// class Z2 ////////////////////////////////////

Z2::Z2(const V<Z>& v)
{
   Z d=v.dim();
   x1= (1<=d ? v[1] : 0);
   x2= (2<=d ? v[2] : 0);
}

bool Z2::prnOn(ostream& out)const
{
```

```
   if (!CpmRoot::writeTitle("Z2",out)) return false;
   if (!CpmRoot::write(x1,out)) return false;
   if (!CpmRoot::write(x2,out)) return false;
   return true;
}

bool Z2::scanFrom(istream& in)
{
   if (!CpmRoot::read(x1,in)) return false;
   if (!CpmRoot::read(x2,in)) return false;
   return true;
}

Z Z2::com(const Z2& a)const
{
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   if (x2<a.x2) return 1;
   if (x2>a.x2) return -1;
   return 0;
}

R Z2::absVal(bool careful)const
{
   return R2(R(x1),R(x2)).absVal(careful);
}

R Z2::disVal(const Z2& x)const
{
   R a=absVal();
   R b=x.absVal();
   R c=(*this-x).absVal();
   return distFunc(a,b,c);
}

X2<R,R2 > Z2::polDec()const
{
   R r=absVal(true);
   R2 pol(x1,x2);
   if (r!=0){
     R rInv=1./r; pol*=rInv;
     return X2<R,R2 >(r,pol);
   }
   else{
      return X2<R,R2 >(0.,R2(1.,0.));
   }
}

//////////////////////////// class Z3 ////////////////////////////////////
```

```
Z3::Z3(const V<Z>& v)
{
   Z d=v.dim();
   x1= (1<=d ? v[1] : 0);
   x2= (2<=d ? v[2] : 0);
   x3= (3<=d ? v[3] : 0);
}

bool Z3::prnOn(ostream& out)const
{
   if (!CpmRoot::writeTitle("Z3",out)) return false;
   if (!CpmRoot::write(x1,out)) return false;
   if (!CpmRoot::write(x2,out)) return false;
   if (!CpmRoot::write(x3,out)) return false;
   return true;
}

bool Z3::scanFrom(istream& in)
{
   if (!CpmRoot::read(x1,in)) return false;
   if (!CpmRoot::read(x2,in)) return false;
   if (!CpmRoot::read(x3,in)) return false;
   return true;
}

Z Z3::com(const Z3& a)const
{
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   if (x2<a.x2) return 1;
   if (x2>a.x2) return -1;
   if (x3<a.x3) return 1;
   if (x3>a.x3) return -1;
   return 0;
}

R Z3::absVal(bool careful)const
{
   return R3(R(x1),R(x2),R(x3)).absVal(careful);
}

R Z3::disVal(const Z3& x)const
{
   R a=absVal();
   R b=x.absVal();
   R c=(*this-x).absVal();
   return distFunc(a,b,c);
}

X2<R,R3> Z3::polDec()const
```

```
{
   R r=absVal(true);
   R3 pol(x1,x2,x3);
   if (r!=0){
     R rInv=1./r; pol*=rInv;
     return X2<R,R3>(r,pol);
   }
   else{
      return X2<R,R3>(0.,R3(1.,0.,0.));
   }
}

//////////////////////////// class R3 ////////////////////////////////////

R3::R3(const V<R>& v)
{
   Z d=v.dim();
   x1= (1<=d ? v[1] : 0);
   x2= (2<=d ? v[2] : 0);
   x3= (3<=d ? v[3] : 0);
}

bool R3::prnOn(ostream& out)const
{
   if (!CpmRoot::writeTitle("R3",out)) return false;
   if (!CpmRoot::write(x1,out)) return false;
   if (!CpmRoot::write(x2,out)) return false;
   if (!CpmRoot::write(x3,out)) return false;
   return true;
}

bool R3::scanFrom(istream& in)
{
   if (!CpmRoot::read(x1,in)) return false;
   if (!CpmRoot::read(x2,in)) return false;
   if (!CpmRoot::read(x3,in)) return false;
   return true;
}


Z R3::com(const R3& a)const
{
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   if (x2<a.x2) return 1;
   if (x2>a.x2) return -1;
   if (x3<a.x3) return 1;
   if (x3>a.x3) return -1;
   return 0;
}
```

```
R R3::absVal(bool careful)const
{
   if (careful==false) return sqrt(x1*x1+x2*x2+x3*x3);
   R y1=(x1>=0 ? x1 : -x1);
   R y2=(x2>=0 ? x2 : -x2);
   R y3=(x3>=0 ? x3 : -x3);
   R y=y1+y2+y3;
   if (y==0) return 0;
   R yInv=1./y;
   y1*=yInv;
   y2*=yInv;
   y3*=yInv;
   return y*sqrt(y1*y1+y2*y2+y3*y3);
}

R R3::disVal(const R3& x)const
{
   R a=absVal();
   R b=x.absVal();
   R c=(*this-x).absVal();
   return distFunc(a,b,c);
}

X2<R,R3 > R3::polDec()const
{
   R r=absVal(true);
   R3 pol=*this;
   if (r!=0){
     R rInv=1./r; pol*=rInv;
     return X2<R,R3 >(r,pol);
   }
   else{
     return X2<R,R3 >(0.,R3(1.,0.,0.));
   }
}

R R3::normalize_(R r)
{
   X2<R,R3> p=polDec(); R r0=p.first();
   *this=p.second()*r;  return r0;
      // was wrong till 2004-11-05
}

R3 R3::carToPol()const
{
   R r=pytCom(x1,x2,true);
   R phi=::atan2(x2,x1);
   R rho=pytCom(r,x3,true);
   R decl=::atan2(x3,r);
```

```
   return R3(rho,phi,decl);
}

R3 R3::polToCar()const
{
   R z=x1*sin(x3); // x1=r, x2=phi, x3=declination
   R r=x1*cos(x3);
   R x=r*cos(x2);
   R y=r*sin(x2);
   return R3(x,y,z);
}

R3 R3::ast(R3 const& x)const
{
   R rr=x1*x1+x2*x2+x3*x3;
   R f1=1-rr;
   R f2=(x1*x.x1+x2*x.x2+x3*x.x3)*2;
   R f3=1./(1+rr); // nominator >= 1
   return R3((f1*x.x1+f2*x1+2*(x2*x.x3-x3*x.x2))*f3,
             (f1*x.x2+f2*x2+2*(x3*x.x1-x1*x.x3))*f3,
             (f1*x.x3+f2*x3+2*(x1*x.x2-x2*x.x1))*f3 );
}

R3 R3::circ(R3 const& x)const
{
   const R eps=1e-12;
   R sp=x1*x.x1+x2*x.x2+x3*x.x3;
   R f1=1-sp;
   if (cpmabs(f1)<eps) f1=eps;
   R f2=1./f1;
   return R3((x1+x.x1+x2*x.x3-x3*x.x2)*f2,
             (x2+x.x2+x3*x.x1-x1*x.x3)*f2,
             (x3+x.x3+x1*x.x2-x2*x.x1)*f2 );
}


R3 R3::ranVal(Z j)const{ CPM_SC;}

//////////////////////////// class Z4 ////////////////////////////////////

Z4::Z4(const V<Z>& v)
{
   Z d=v.dim();
   x1= (1<=d ? v[1] : 0);
   x2= (2<=d ? v[2] : 0);
   x3= (3<=d ? v[3] : 0);
   x4= (4<=d ? v[4] : 0);
}

bool Z4::prnOn(ostream& out)const
```

```
{
   if (!CpmRoot::writeTitle("Z4",out)) return false;
   if (!CpmRoot::write(x1,out)) return false;
   if (!CpmRoot::write(x2,out)) return false;
   if (!CpmRoot::write(x3,out)) return false;
   if (!CpmRoot::write(x4,out)) return false;
   return true;
}

bool Z4::scanFrom(istream& in)
{
   if (!CpmRoot::read(x1,in)) return false;
   if (!CpmRoot::read(x2,in)) return false;
   if (!CpmRoot::read(x3,in)) return false;
   if (!CpmRoot::read(x4,in)) return false;
   return true;
}

Z Z4::com(const Z4& a)const
{
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   if (x2<a.x2) return 1;
   if (x2>a.x2) return -1;
   if (x3<a.x3) return 1;
   if (x3>a.x3) return -1;
   if (x4<a.x4) return 1;
   if (x4>a.x4) return -1;
   return 0;
}

R Z4::absVal(bool careful)const
{
   return R4(R(x1),R(x2),R(x3),R(x4)).absVal(careful);
}

R Z4::disVal(const Z4& x)const
{
   R a=absVal();
   R b=x.absVal();
   R c=(*this-x).absVal();
   return distFunc(a,b,c);
}

X2<R,R4> Z4::polDec()const
{
   R r=absVal(true);
   R4 pol(x1,x2,x3,x4);
   if (r!=0){
     R rInv=1./r; pol*=rInv;
```

```
     return X2<R,R4>(r,pol);
   }
   else{
      return X2<R,R4>(0.,R4(1.,0.,0.,0.));
   }
}

//////////////////////////// class R4 ////////////////////////////////////

R4::R4(const V<R>& v)
{
   Z d=v.dim();
   x1= (1<=d ? v[1] : 0);
   x2= (2<=d ? v[2] : 0);
   x3= (3<=d ? v[3] : 0);
   x4= (4<=d ? v[4] : 0);
}

bool R4::prnOn(ostream& out)const
{
   if (!CpmRoot::writeTitle("R4",out)) return false;
   if (!CpmRoot::write(x1,out)) return false;
   if (!CpmRoot::write(x2,out)) return false;
   if (!CpmRoot::write(x3,out)) return false;
   if (!CpmRoot::write(x4,out)) return false;
   return true;
}

bool R4::scanFrom(istream& in)
{
   if (!CpmRoot::read(x1,in)) return false;
   if (!CpmRoot::read(x2,in)) return false;
   if (!CpmRoot::read(x3,in)) return false;
   if (!CpmRoot::read(x4,in)) return false;
   return true;
}

Z R4::com(const R4& a)const
{
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   if (x2<a.x2) return 1;
   if (x2>a.x2) return -1;
   if (x3<a.x3) return 1;
   if (x3>a.x3) return -1;
   if (x4<a.x4) return 1;
   if (x4>a.x4) return -1;
   return 0;
}
```

```
R R4::absVal(bool careful)const
{
   if (careful==false) return sqrt(x1*x1+x2*x2+x3*x3+x4*x4);
   R y1=(x1>=0 ? x1 : -x1);
   R y2=(x2>=0 ? x2 : -x2);
   R y3=(x3>=0 ? x3 : -x3);
   R y4=(x4>=0 ? x4 : -x4);
   R y=y1+y2+y3+y4;
   if (y==0) return 0;
   R yInv=1./y;
   y1*=yInv;
   y2*=yInv;
   y3*=yInv;
   y4*=yInv;
   return y*sqrt(y1*y1+y2*y2+y3*y3+y4*y4);
}

R R4::disVal(const R4& x)const
{
   R a=absVal();
   R b=x.absVal();
   R c=(*this-x).absVal();
   return distFunc(a,b,c);
}

X2<R,R4 > R4::polDec()const
{
   R r=absVal(true);
   R4 pol=*this;
   if (r!=0){
     R rInv=1./r; pol*=rInv;
     return X2<R,R4 >(r,pol);
   }
   else{
      return X2<R,R4 >(0.,R4(1.,0.,0.,0.));
   }
}

R R4::normalize_(R r)
{
   X2<R,R4> p=polDec(); R r0=p.first();
   *this=p.second()*r;  return r0;
}

R4 R4::ranVal(Z j)const{ CPM_SC;}

#undef CPM_SC
//////////////////////// RR2 ////////////////////////////////////////////

R2 CpmLinAlg::operator*(const RR2& a, const R2& x)
```

```
   {
      R2 res;
      res.x1=(a.x1|x);
      res.x2=(a.x2|x);
      return res;
   }

RR2 CpmLinAlg::operator*(const RR2& a, const RR2& b)
   {
      RR2 res;
      res.x1.x1=a.x1.x1*b.x1.x1+a.x1.x2*b.x2.x1;
      res.x1.x2=a.x1.x1*b.x1.x2+a.x1.x2*b.x2.x2;
      res.x2.x1=a.x2.x1*b.x1.x1+a.x2.x2*b.x2.x1;
      res.x2.x2=a.x2.x1*b.x1.x2+a.x2.x2*b.x2.x2;
      return res;
   }

RR2 CpmLinAlg::operator+(const RR2& a, const RR2& b)
   {
      RR2 res=a;
      res.x1+=b.x1;
      res.x2+=b.x2;
      return res;
   }

RR2 CpmLinAlg::operator~(const RR2& a)
   {
      RR2 res=a;
      res.x1.x2=a.x2.x1;
      res.x2.x1=a.x1.x2;
      return res;
   }

RR2 RR2::operator !()const
{
   R detInv=CpmRoot::inv(det());
   RR2 res;
   res.x1.x1=x2.x2*detInv;
   res.x2.x2=x1.x1*detInv;
   res.x1.x2=-x1.x2*detInv;
   res.x2.x1=-x2.x1*detInv;
      // *detInv was missing till 2005-06-22
   return res;
}

RR2 RR2::zero;

RR2 RR2::unit=RR2(R2(1,0),R2(0,1));

bool RR2::prnOn(ostream& out)const
```

```
{
     if (!x1.prnOn(out)) return false;
     if (!x2.prnOn(out)) return false;
     return true;
}

bool RR2::scanFrom(istream& in)
{
   if (!x1.scanFrom(in)) return false;
   if (!x2.scanFrom(in)) return false;
   return true;
}

R3 CpmLinAlg::operator*(const RR3& a, const R3& x)
   {
      R3 res;
      res.x1=(a.x1|x);
      res.x2=(a.x2|x);
      res.x3=(a.x3|x);
      return res;
   }


RR3 CpmLinAlg::operator*(const RR3& a, const RR3& b)
   {
      RR3 res;
#ifndef CPM_NOT_KILL_OPTIMIZER
      res.x1.x1=a.x1.x1*b.x1.x1 + a.x1.x2*b.x2.x1 + a.x1.x3*b.x3.x1;
      res.x1.x2=a.x1.x1*b.x1.x2 + a.x1.x2*b.x2.x2 + a.x1.x3*b.x3.x2;
      res.x1.x3=a.x1.x1*b.x1.x3 + a.x1.x2*b.x2.x3 + a.x1.x3*b.x3.x3;

      res.x2.x1=a.x2.x1*b.x1.x1 + a.x2.x2*b.x2.x1 + a.x2.x3*b.x3.x1;
      res.x2.x2=a.x2.x1*b.x1.x2 + a.x2.x2*b.x2.x2 + a.x2.x3*b.x3.x2;
      res.x2.x3=a.x2.x1*b.x1.x3 + a.x2.x2*b.x2.x3 + a.x2.x3*b.x3.x3;

      res.x3.x1=a.x3.x1*b.x1.x1 + a.x3.x2*b.x2.x1 + a.x3.x3*b.x3.x1;
      res.x3.x2=a.x3.x1*b.x1.x2 + a.x3.x2*b.x2.x2 + a.x3.x3*b.x3.x2;
      res.x3.x3=a.x3.x1*b.x1.x3 + a.x3.x2*b.x2.x3 + a.x3.x3*b.x3.x3;
#else
      cpmerror(
         "definition of RR3*RR3 disabled by a macro in cpmlinalg.cpp");
#endif
      return res;
   }

RR3 CpmLinAlg::operator+(const RR3& a, const RR3& b)
   {
      RR3 res=a;
      res.x1+=b.x1;
      res.x2+=b.x2;
```

```
      res.x3+=b.x3;
      return res;
   }

RR3 CpmLinAlg::operator~(const RR3& a)
   {
      RR3 res(a);

      res.x1.x2=a.x2.x1;
      res.x1.x3=a.x3.x1;

      res.x2.x3=a.x3.x2;
      res.x2.x1=a.x1.x2;

      res.x3.x1=a.x1.x3;
      res.x3.x2=a.x2.x3;

      return res;
   }

RR3 RR3::zero;

RR3 RR3::unit=RR3(R3(1,0,0),R3(0,1,0),R3(0,0,1));

bool RR3::prnOn(ostream& out)const
{
     if (!x1.prnOn(out)) return false;
     if (!x2.prnOn(out)) return false;
     if (!x3.prnOn(out)) return false;
     return true;
}

bool RR3::scanFrom(istream& in)
{
   if (!x1.scanFrom(in)) return false;
   if (!x2.scanFrom(in)) return false;
   if (!x3.scanFrom(in)) return false;
   return true;
}

R4 CpmLinAlg::operator*(const RR4& a, const R4& x)
   {
      R4 res;
      res.x1=(a.x1|x);
      res.x2=(a.x2|x);
      res.x3=(a.x3|x);
      res.x4=(a.x4|x);
      return res;
   }
```

```
RR4 CpmLinAlg::operator*(const RR4& a, const RR4& b)
   {
      RR4 res;
#ifndef CPM_NOT_KILL_OPTIMIZER
      res.x1.x1=a.x1.x1*b.x1.x1 + a.x1.x2*b.x2.x1 + a.x1.x3*b.x3.x1
         + a.x1.x4*b.x4.x1;
      res.x1.x2=a.x1.x1*b.x1.x2 + a.x1.x2*b.x2.x2 + a.x1.x3*b.x3.x2
         + a.x1.x4*b.x4.x2;
      res.x1.x3=a.x1.x1*b.x1.x3 + a.x1.x2*b.x2.x3 + a.x1.x3*b.x3.x3
         + a.x1.x4*b.x4.x3;
      res.x1.x4=a.x1.x1*b.x1.x4 + a.x1.x2*b.x2.x4 + a.x1.x3*b.x3.x4
         + a.x1.x4*b.x4.x4;

      res.x2.x1=a.x2.x1*b.x1.x1 + a.x2.x2*b.x2.x1 + a.x2.x3*b.x3.x1
         + a.x2.x4*b.x4.x1;
      res.x2.x2=a.x2.x1*b.x1.x2 + a.x2.x2*b.x2.x2 + a.x2.x3*b.x3.x2
         + a.x2.x4*b.x4.x2;
      res.x2.x3=a.x2.x1*b.x1.x3 + a.x2.x2*b.x2.x3 + a.x2.x3*b.x3.x3
         + a.x2.x4*b.x4.x3;
      res.x2.x4=a.x2.x1*b.x1.x4 + a.x2.x2*b.x2.x4 + a.x2.x3*b.x3.x4
         + a.x2.x4*b.x4.x4;

      res.x3.x1=a.x3.x1*b.x1.x1 + a.x3.x2*b.x2.x1 + a.x3.x3*b.x3.x1
         + a.x3.x4*b.x4.x1;
      res.x3.x2=a.x3.x1*b.x1.x2 + a.x3.x2*b.x2.x2 + a.x3.x3*b.x3.x2
         + a.x3.x4*b.x4.x2;
      res.x3.x3=a.x3.x1*b.x1.x3 + a.x3.x2*b.x2.x3 + a.x3.x3*b.x3.x3
         + a.x3.x4*b.x4.x3;
      res.x3.x4=a.x3.x1*b.x1.x4 + a.x3.x2*b.x2.x4 + a.x3.x3*b.x3.x4
         + a.x3.x4*b.x4.x4;

      res.x4.x1=a.x4.x1*b.x1.x1 + a.x4.x2*b.x2.x1 + a.x4.x3*b.x3.x1
         + a.x4.x4*b.x4.x1;
      res.x4.x2=a.x4.x1*b.x1.x2 + a.x4.x2*b.x2.x2 + a.x4.x3*b.x3.x2
         + a.x4.x4*b.x4.x2;
      res.x4.x3=a.x4.x1*b.x1.x3 + a.x4.x2*b.x2.x3 + a.x4.x3*b.x3.x3
         + a.x4.x4*b.x4.x3;
      res.x4.x4=a.x4.x1*b.x1.x4 + a.x4.x2*b.x2.x4 + a.x4.x3*b.x3.x4
         + a.x4.x4*b.x4.x4;
#else
      cpmerror(
         "definition of RR4*RR4 disabled by a macro in cpmlinalg.cpp");
#endif

      return res;
   }

RR4 CpmLinAlg::operator+(const RR4& a, const RR4& b)
   {
      RR4 res=a;
```

```
      res.x1+=b.x1;
      res.x2+=b.x2;
      res.x3+=b.x3;
      res.x4+=b.x4;
      return res;
   }

RR4 CpmLinAlg::operator~(const RR4& a)
   {
      RR4 res(a);

      res.x1.x2=a.x2.x1;
      res.x1.x3=a.x3.x1;
      res.x1.x4=a.x4.x1;

      res.x2.x1=a.x1.x2;
      res.x2.x3=a.x3.x2;
      res.x2.x4=a.x4.x2;

      res.x3.x1=a.x1.x3;
      res.x3.x2=a.x2.x3;
      res.x3.x4=a.x4.x3;

      res.x4.x1=a.x1.x4;
      res.x4.x2=a.x2.x4;
      res.x4.x3=a.x3.x4;

      return res;
   }

RR4 RR4::zero;

RR4 RR4::unit=
RR4(R4(1,0,0,0),R4(0,1,0,0),R4(0,0,1,0),R4(0,0,0,1));

bool RR4::prnOn(ostream& out)const
{
     if (!x1.prnOn(out)) return false;
     if (!x2.prnOn(out)) return false;
     if (!x3.prnOn(out)) return false;
     if (!x4.prnOn(out)) return false;
     return true;
}

bool RR4::scanFrom(istream& in)
{
   if (!x1.scanFrom(in)) return false;
   if (!x2.scanFrom(in)) return false;
   if (!x3.scanFrom(in)) return false;
   if (!x4.scanFrom(in)) return false;
```

```
   return true;
}
```

107 cpmlineart.h

```
//? cpmlineart.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_LINEART_H_
#define CPM_LINEART_H_

//Description: see cpmlineartx.h

#include <cpmdimdef.h>
#include <cpmdim.h>
#include <cpmcamera.h>
#if defined(CPM_DIMDEF2)

#define CPM_DIM CpmDim2
#define CPM_CAMERA CpmCamera2

namespace CPM_DIM{
   #include <cpmlineartx.h>
}

#undef CPM_DIM
#undef CPM_CAMERA

#endif

//////////////////////////////////////////////////////////////////////

#if defined(CPM_DIMDEF3)

#define CPM_DIM CpmDim3
#define CPM_CAMERA CpmCamera3

namespace CPM_DIM{
   #include <cpmlineartx.h>
}

#undef CPM_DIM
#undef CPM_CAMERA

#endif

#endif
```

108 cpmlineart.cpp

```
//? cpmlineart.cpp
//? Status of work 2008-10-25.
//?
#include <cpmlineart.h>

#if defined(CPM_DIMDEF2)

#define CPM_DIM CpmDim2
#define CPM_CAMERA CpmCamera2

namespace CPM_DIM{
   #include <cpmlineartcpp.h>
}

#undef CPM_DIM
#undef CPM_CAMERA

#endif

//////////////////////////////////////////////////////////////////////////

#if defined(CPM_DIMDEF3)

#define CPM_DIM CpmDim3
#define CPM_CAMERA CpmCamera3

namespace CPM_DIM{
   #include <cpmlineartcpp.h>
}

#undef CPM_DIM
#undef CPM_CAMERA

#endif
```

109 cpmlineartcpp.h

```
//? cpmlineartcpp.h
//? Status of work 2008-10-25.
//?
using namespace CpmRoot;
using namespace CpmImaging;
using namespace CpmArrays;

using CPM_DIM::Vec;
using CPM_DIM::AxVec;
using CPM_DIM::Spc;
using CPM_DIM::Group;
using CPM_DIM::Rdim;
using CPM_DIM::Rdim_;
using CPM_DIM::Skeleton;
using CPM_DIM::Triangle;
using CPM_DIM::Path;
using CPM_DIM::IndCon;

using CPM_CAMERA::Camera;

using CpmGeo::Angle;

//////////////////////////////// class Wrt ////////////////////////////////

V<Vec> Wrt::bas()const
{
   Vec ex=l_-f_; ex.nor_();
   Vec ey=(u_-f_)-ex*w_; ey.nor_();
   Z n=Spc::dimension();
   V<Vec> res(n);
   res[1]=ex;
   res[2]=ey;
   if (n==3) res[3]=Vec(ex.cross(ey));
      // cross directly gives an AxVec
   return res;
}

Wrt& Wrt::operator *=(Group const& g)
{
   for (Z i=v_.b(); i<=v_.e(); ++i) v_[i]*=g;
   f_*=g;
   l_*=g;
   u_*=g;
   return *this;
}

Wrt& Wrt::operator +=(Vec const& vec)
```

```
{
   for (Z i=v_.b(); i<=v_.e(); ++i) v_[i]+=vec;
   f_+=vec;
   l_+=vec;
   u_+=vec;
   return *this;
}

Wrt& Wrt::operator *=(AxVec const& av)
{
   for (Z i=v_.b(); i<=v_.e(); ++i) v_[i].rot_(f_,av);
   l_=l_.rot_(f_,av);
   u_=u_.rot_(f_,av);
   return *this;
}

Wrt& Wrt::operator *=(R scl)
{
   for (Z i=v_.b(); i<=v_.e(); ++i) v_[i]=v_[i].scl(f_,scl);
   h_*=scl;
   w_*=scl;
   d_*=scl;
   l_=l_.scl(f_,scl);
   u_=u_.scl(f_,scl);
   return *this;
}

Wrt& Wrt::scl_(Spc const& fix, R fac)
{
   for (Z i=v_.b(); i<=v_.e(); ++i) v_[i].scl_(fix,fac);
   h_*=fac;
   w_*=fac;
   d_*=fac;
   l_.scl_(fix,fac);
   u_.scl_(fix,fac);
   return *this;
}

Wrt& Wrt::thc_(R r, Z n)
{
   if (r<=0 || n<=1) return *this;
   Spc o;
   Path disk=Path::circle(o,r,n);
   V<Spc> cir=disk();
   Z nc=cir.dim(); // paranoia, should be n
   V<Vec> vcir(nc);
   Z i,j;
   for (i=1;i<=nc;++i) vcir[i]=cir[i]-o;
   V<Path> vThick=v_;
   Z np=v_.dim();
```

```
   for (j=1;j<=nc;++j){
      V<Path> vj=v_;
      Vec sj=vcir[j];
      for (i=1; i<=np; ++i) vj[i]+=sj;
      vThick&=vj;
   }
   v_=vThick;
   Vec sh(2*r,0);
   l_+=sh;
   u_+=sh;
   h_+=r; // height over base line grows only by r not 2r
   w_+=2*r;
   d_+=r;
   return *this;
}

Wrt& Wrt::to_(Spc const& pp)
{
   Vec shift=pp-f_;
   f_+=shift;
   l_+=shift;
   u_+=shift;
   for (Z i=v_.b(); i<=v_.e(); ++i) v_[i]+=shift;
   return *this;
}

bool Wrt::prnOn(ostream& str)const
{
   cpmwat;
   cpmp(v_);
   cpmp(f_);
   cpmp(l_);
   cpmp(u_);
   cpmp(h_);
   cpmp(w_);
   cpmp(d_);
   return true;
}

bool Wrt::scanFrom(istream& str)
{
   cpms(v_);
   cpms(f_);
   cpms(l_);
   cpms(u_);
   cpms(h_);
   cpms(w_);
   cpms(d_);
   return true;
}
```

```
void Wrt::mark(Camera const& ca, Color const& c)const
{
   for (Z i=v_.b();i<=v_.e();++i) ca.mark(v_[i],c);
}

Wrt& Wrt::vec_()
{
   Vec ex=e1();
   Vec ey=e2();
   R hv=1.2*h_; // 1.2 adjustable parameter
   Spc p1=f_+ey*hv;
   Spc p2=p1+ex*w_;
   V<Path> vs(2);
   Vec dir=ex*(-2)+ey; // -2 adjustable parameter
   dir.nor_();
   dir*=(w_*0.3); // 0.3 adjustable parameter
   vs[1]=Path(p1,p2);
   vs[2]=Path(p2,p2+dir);
   v_&=vs;
   R dh=(dir|ey);
   h_=hv+dh;
   u_=f_+ex*w_+ey*h_;
   return *this;
}

Wrt& Wrt::sub_(V<Wrt> const& index, IndCon const& ic)
{
   V<Vec> b=bas();
   Vec sl=spc();
   LineArt la=LineArt::text(index);
   Spc fpi=la.cw().fir().fir();
   la.scl_(fpi,ic.fac);
   R dh=h_*ic.dh, dw=w_*ic.dw;
   Spc con=last()+b[1]*dw+b[2]*dh;
   Vec shift=con-fpi;
   la+=shift;
   V<Wrt> vw=la.cw();
   for (Z i=vw.b();i<=vw.e();++i) v_&=vw[i].v();
   Spc lpi=la.cw().last().rigLowCor();
   w_=((lpi-f_)|b[1]);
   l_=f_+w_*b[1]+sl;
   // u_ unchanged, to allow for upper indexes to be appended
   // after lower ones
   return *this;
}

Wrt& Wrt::sup_(V<Wrt> const& index, IndCon const& ic)
{
   V<Vec> b=bas();
```

```
   Vec sl=spc();
   LineArt la=LineArt::text(index);
   Spc fpi=la.cw().fir().fir();
   la.scl_(fpi,ic.fac);
   R dh=h_*ic.dh, dw=w_*ic.dw;
   Spc con=rigUppCor()+b[1]*dw+b[2]*dh+sl;
   Vec shift=con-fpi;
   la+=shift;
   V<Wrt> vw=la.cw();
   for (Z i=vw.b();i<=vw.e();++i) v_&=vw[i].v();
   Spc upi=la.cw().last().rigUppCor();
   u_=upi;
   Vec uv=u_-f_;
   R w_=(uv|b[1]);
   R h_=(uv|b[2]);
   l_=f_+b[1]*w_+sl;
   return *this;
}

/// what follows is a gigantic function which defines some vector font ///

Wrt::Wrt(Word const& name)
{
   Word loc("Wrt(Word)");
   Spc o(0,0);
   Vec e1(1,0);
   Vec e2(0,1);
   R uR=1./7;
   AxVec rot180(Angle(180,DEG),3);
   AxVec rot90(Angle(90,DEG),3);
   AxVec rot45(Angle(45,DEG),3);
   AxVec rot90m(Angle(-90,DEG),3);
   R w,h;
   R wi=1,wj=2,wr=3,wa=4,wm=7;
   R wI=1,wJ=3,wL=4,wB=5,wA=6,wW=7;
   R dA=2;
   R hA=7, ha=5, dg=2;
   R wPeriod=1; // width of period and 'i-tüpfelchen'
   if (name=="A"){
      w=wA;
      h=hA;
      R hr=0.3;
      v_=V<Path>(3);
      f_=o;
      l_=o+Vec(w+dA,0);
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      Spc top(w*0.5,h);
      Spc foot(w,0);
      Spc m1=o.versus(top,hr);
```

```
      Spc m2=foot.versus(top,hr);
      v_[1]=Path(o,top);
      v_[2]=Path(m1,m2);
      v_[3]=Path(top,foot);
   }
   if (name=="Au"){
      Wrt au("A");
      w=wA;
      R w1=0.3*w;
      R w2=0.7*w;
      h=hA;
      R r=0.5*wPeriod;
      Spc pc1(w1,h+wPeriod);
      Spc pc2(w2,h+wPeriod);
      V<Path> add(2);
      add[1]=Path::circle(pc1,r);
      add[2]=Path::circle(pc2,r);
      au&=add;
      *this=au;
   }
   else if (name=="B"){
      w=wB;
      h=hA;
      R hr=0.5;
      v_=V<Path>(6);
      f_=o;
      l_=o+Vec(w+dA,0);
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      R horB=w-h*0.25;
      Spc t1(0,h);
      Spc m1=o.versus(t1,hr);
      Vec hor(horB,0);
      Spc t2=t1+hor;
      Spc m2=m1+hor;
      Spc f2=o+hor;
      v_[1]=Path(o,t1);
      v_[2]=Path(o,f2);
      v_[3]=Path(m1,m2);
      v_[4]=Path(t1,t2);
      v_[5]=Path::arc(f2,f2.cen(m2),rot180);
      v_[6]=Path::arc(m2,m2.cen(t2),rot180);
   }
   else if (name=="C"){
      w=wB;
      h=hA;
      R l1=0.3;
      R l2=0.7;
      v_=V<Path>(5);
      f_=o;
```

```
      R wl=w+dA;
      l_=o+e1*wl;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      R h1=h*l1, h2=h*l2, w1=w*0.5, w2=w;
      Spc p1(0,h1);
      Spc p2(0,h2);
      v_[1]=Path(p1,p2);
      Spc p3(w1,h);
      v_[2]=Path::quarterEllipse(p2,e2,p3);
      Spc p4(w2,h2);
      v_[3]=Path::quarterEllipse(p3,e1,p4);
      Spc q1(w1,0);
      v_[4]=Path::quarterEllipse(p1,-e2,q1);
      Spc q2(w2,h1);
      v_[5]=Path::quarterEllipse(q1,e1,q2);
   }
   else if (name=="D"){
      w=wB;
      h=hA;
      v_=V<Path>(6);
      f_=o;
      l_=Spc(w+dA,0);
      h_=hA;
      d_=0;
      Spc t0(0,h);
      v_[1]=Path(o,t0);
      R dr=0.2;
      R d=dr*h;
      R r=(h-d)*0.5;
      R dz=w-r;
      cpmassert(dz>0,"Wr(D)");
      Spc a1(dz,h);
      Spc c1(dz,h-r);
      Spc m1(w,h-r);
      v_[2]=Path::arc(a1,c1,rot90m);
      Spc a2(dz,0);
      Spc c2(dz,r);
      Spc m2(w,r);
      v_[3]=Path::arc(a2,c2,rot90);
      v_[4]=Path(m1,m2);
      v_[5]=Path(o,a2);
      v_[6]=Path(t0,a1);
   }
   else if (name=="E"){
      w=wB;
      h=hA;
      R rm=0.75;
      v_=V<Path>(4);
      f_=o;
```

```
   l_=Spc(w+dA,0);
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   Spc t(0,h);
   v_[1]=Path(o,t);
   Vec h(w,0);
   Vec hm(w*rm,0);
   Spc t1=t+h;
   Spc f1=o+h;
   Spc m=o.versus(t,0.5);
   v_[2]=Path(o,f1);
   v_[3]=Path(t,t1);
   v_[4]=Path(m,m+hm);
}
else if (name=="F"){
   w=wB;
   h=hA;
   R rm=0.75;
   v_=V<Path>(3);
   f_=o;
   l_=Spc(w+dA,0);
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   Spc t(0,h);
   v_[1]=Path(o,t);
   Vec h(w,0);
   Vec hm(w*rm,0);
   Spc t1=t+h;
   Spc m=o.versus(t,0.5);
   v_[2]=Path(t,t1);
   v_[3]=Path(m,m+hm);
}
else if (name=="G"){
   w=wB;
   h=hA;
   v_=V<Path>(8);
   f_=o;
   R tW=w+dA;
   l_=o+e1*tW;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   R l2=0.7;
   R l1=0.3;
   Spc t(0,h);
   Spc m1=o.versus(t,l1);
   Spc m2=o.versus(t,l2);
   Vec w1=e1*w;
   v_[1]=Path(m1,m2);
   Spc m3=m1+w1;
   Spc m4=m2+w1;
```

```
   Vec w2=w1*0.5;
   Spc f2=o+w2;
   Spc t2=t+w2;
   v_[3]=Path::quarterEllipse(m2,e2,t2);
   v_[4]=Path::quarterEllipse(t2,e1,m4);
   v_[5]=Path::quarterEllipse(m1,-e2,f2);
   v_[6]=Path::quarterEllipse(f2,e1,m3);
   Spc m5(w,0.5*h_);
   v_[7]=Path(m3,m5);
   Spc m6=m5-e1*(w*0.4);
   v_[8]=Path(m5,m6);
}
else if (name=="H"){
   w=wB;
   h=hA;
   v_=V<Path>(3);
   f_=o;
   R tH=w+dA;
   l_=o+e1*tH;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   Spc t(0,h);
   Vec v=e1*w;
   Spc m1=o.versus(t,0.5);
   v_[1]=Path(o,t);
   v_[2]=Path(o+v,t+v);
   v_[3]=Path(m1,m1+v);
}
else if (name=="I"){
   w=wI;
   h=hA;
   v_=V<Path>(1);
   f_=o;
   R tI=w+dA;
   l_=o+e1*tI;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   Vec v=e1*(0.5*wI);
   Spc t(0,h);
   v_[1]=Path(o+v,t+v);
}
else if (name=="J"){
   w=wJ;
   h=hA;
   R h1r=0.2;
   R wJh=w*0.5;
   v_=V<Path>(3);
   f_=o;
   R tJ=w+dA;
   l_=o+e1*tJ;
```

```
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   R h1=h*h1r;
   Spc p1(0,h1);
   Spc p2(wJh,0);
   Spc p3(w,h1);
   Spc p4(w,h);
   v_[1]=Path::quarterEllipse(p1,-e2,p2);
   v_[2]=Path::quarterEllipse(p2,e1,p3);
   v_[3]=Path(p3,p4);
}
else if (name=="K"){
   w=wB;
   h=hA;
   R h1=0.45;
   R h2=0.25;
   v_=V<Path>(3);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   Spc p1(o);
   Spc p2(0,h);
   Spc p3=p1.versus(p2,h1);
   Spc p4(w,h);
   Spc p5=p3.versus(p4,h2);
   Spc p6(w,0);
   v_[1]=Path(p1,p2);
   v_[2]=Path(p3,p4);
   v_[3]=Path(p5,p6);
}
else if (name=="L"){
   w=wL;
   h=hA;
   v_=V<Path>(2);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   Spc p1(o);
   Spc p2(0,h);
   Spc p3(wL,0);
   v_[1]=Path(p1,p2);
   v_[2]=Path(p1,p3);
}
else if (name=="M"){
   w=wA;
   h=hA;
```

```
   R hr=0.4;
   h_=h; w_=w; u_=Spc(w_,h_);
   R h1=h_*hr;
   v_=V<Path>(4);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;
   d_=0;
   Spc p1(o);
   Spc p2(0,h);
   Spc p3(0.5*w,h1);
   Spc p4(w,h);
   Spc p5(w,0);
   v_[1]=Path(p1,p2);
   v_[2]=Path(p2,p3);
   v_[3]=Path(p3,p4);
   v_[4]=Path(p4,p5);
}
else if (name=="N"){
   w=wB;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(3);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;

   Spc p1(o);
   Spc p2(0,h);
   Spc p3(w,0);
   Spc p4(w,h);

   v_[1]=Path(p1,p2);
   v_[2]=Path(p2,p3);
   v_[3]=Path(p3,p4);
}
else if (name=="O"){
   w=wB;
   h=hA;
   v_=V<Path>(6);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   R l2=0.75;
   R l1=0.25;
   Spc t(0,h);
   Spc m1=o.versus(t,l1);
```

```
   Spc m2=o.versus(t,l2);
   Vec w1=e1*w;
   v_[1]=Path(m1,m2);
   Spc m3=m1+w1;
   Spc m4=m2+w1;
   v_[2]=Path(m3,m4);
   Vec w2=w1*0.5;
   Spc f2=o+w2;
   Spc t2=t+w2;
   v_[3]=Path::quarterEllipse(m2,e2,t2);
   v_[4]=Path::quarterEllipse(t2,e1,m4);
   v_[5]=Path::quarterEllipse(m1,-e2,f2);
   v_[6]=Path::quarterEllipse(f2,e1,m3);
      // quarterEllipse works as expected
}
if (name=="Ou"){
   Wrt au("O");
   w=wB;
   R w1=0.3*w;
   R w2=0.7*w;
   h=hA;
   R r=0.5*wPeriod;
   Spc pc1(w1,h+wPeriod);
   Spc pc2(w2,h+wPeriod);
   V<Path> add(2);
   add[1]=Path::circle(pc1,r);
   add[2]=Path::circle(pc2,r);
   au&=add;
   *this=au;
}
else if (name=="P"){
   w=wB;
   h=hA;
   R hr=0.5;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   R sP=w-h*(1-hr)*0.5;
   cpmassert(sP>0,loc);
   v_=V<Path>(4);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;
   Vec v=e1*sP;

   Spc p1(o);
   Spc p2(0,h);
   Spc p3=p1.versus(p2,hr);
   Spc p4=p2+v;
   Spc p5=p3+v;
```

```
   v_[1]=Path(p1,p2);
   v_[2]=Path(p2,p4);
   v_[3]=Path(p3,p5);
   v_[4]=Path::arc(p5,p4.cen(p5),rot180);
}
else if (name=="Q"){
   w=wB;
   h=hA;
   v_=V<Path>(7);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   R l2=0.75;
   R l1=0.25;
   Spc t(0,h);
   Spc m1=o.versus(t,l1);
   Spc m2=o.versus(t,l2);
   Vec w1=e1*w;
   v_[1]=Path(m1,m2);
   Spc m3=m1+w1;
   Spc m4=m2+w1;
   v_[2]=Path(m3,m4);
   Vec w2=w1*0.5;
   Spc f2=o+w2;
   Spc t2=t+w2;
   v_[3]=Path::quarterEllipse(m2,e2,t2);
   v_[4]=Path::quarterEllipse(t2,e1,m4);
   v_[5]=Path::quarterEllipse(m1,-e2,f2);
   v_[6]=Path::quarterEllipse(f2,e1,m3);
   R hx=l1*h;
   v_[7]=Path(Spc(w,0),Spc(w-hx,hx) );
}
else if (name=="R"){
   w=wB;
   h=hA;
   R hr=0.5;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   R sR=w-h*(1-hr)*0.5;
   cpmassert(sR>0,loc);
   v_=V<Path>(5);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;
   Vec v=e1*sR;

   Spc p1(o);
   Spc p2(0,h);
```

```
      Spc p3=p1.versus(p2,hr);
      Spc p4=p2+v;
      Spc p5=p3+v;

      v_[1]=Path(p1,p2);
      v_[2]=Path(p2,p4);
      v_[3]=Path(p3,p5);
      v_[4]=Path::arc(p5,p4.cen(p5),rot180);
      v_[5]=Path(p5,Spc(w,0));
   }
   else if (name=="S"){
      w=wB;
      h=hA;
      R h1r=0.25;
      R h2r=0.5;
      R h3r=0.75;
      h_=h; w_=w; u_=Spc(w_,h_);
      R h1=h1r*h;
      R h2=h2r*h;
      R h3=h3r*h;
      R w1=w*0.5;
      d_=0;
      v_=V<Path>(6);
      f_=o;
      R wl=w+dA;
      l_=o+e1*wl;
      Vec v=e1*w;
      Vec vh=v*0.5;
      Spc p1(0,h3);
      Spc p2(w1,h);
      v_[1]=Path::quarterEllipse(p1,e2,p2);
      Spc p3(w,h3);
      v_[2]=Path::quarterEllipse(p2,e1,p3);
      Spc q1(w1,h2);
      v_[3]=Path::quarterEllipse(p1,-e2,q1);
      Spc q2(w,h1);
      v_[4]=Path::quarterEllipse(q1,e1,q2);
      Spc q3(w1,0);
      v_[5]=Path::quarterEllipse(q2,-e2,q3);
      Spc q4(0,h1);
      v_[6]=Path::quarterEllipse(q3,-e1,q4);
   }
   else if (name=="T"){
      w=wB;
      h=hA;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      v_=V<Path>(2);
      f_=o;
      R wl=w+dA;
```

```
      l_=o+e1*wl;

      R w1=w*0.5;

      Spc p1(w1,0);
      Spc p2(w1,h);
      Spc p3(0,h);
      Spc p4(w,h);

      v_[1]=Path(p1,p2);
      v_[2]=Path(p3,p4);
   }
   else if (name=="U"){
      w=wB;
      h=hA;
      R hr=0.25;
      h_=h; w_=w; u_=Spc(w_,h_);
      R h1=h*hr;
      d_=0;
      v_=V<Path>(4);
      f_=o;
      R wl=w+dA;
      l_=o+e1*wl;

      R w1=w*0.5;

      Spc p1(0,h);
      Spc p2(0,h1);
      v_[1]=Path(p1,p2);
      Spc p3(w1,0);
      v_[2]=Path::quarterEllipse(p2,-e2,p3);
      Spc p4(w,h1);
      v_[3]=Path::quarterEllipse(p3,e1,p4);
      Spc p5(w,h);
      v_[4]=Path(p4,p5);
   }
   if (name=="Uu"){
      Wrt au("U");
      w=wB;
      R w1=0.3*w;
      R w2=0.7*w;
      h=hA;
      R r=0.5*wPeriod;
      Spc pc1(w1,h+wPeriod);
      Spc pc2(w2,h+wPeriod);
      V<Path> add(2);
      add[1]=Path::circle(pc1,r);
      add[2]=Path::circle(pc2,r);
      au&=add;
      *this=au;
```

```
}
else if (name=="V"){
   w=wB;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(2);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;

   R w1=w*0.5;

   Spc p1(0,h);
   Spc p2(w1,0);
   v_[1]=Path(p1,p2);
   Spc p3(w,h);
   v_[2]=Path(p2,p3);
}
else if (name=="W"){
   w=wW;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(4);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;
   R w1=w*0.25;
   R w2=w*0.5;
   R w3=w*0.75;
   Spc p1(0,h);
   Spc p2(w1,0);
   v_[1]=Path(p1,p2);
   Spc p3(w2,h);
   v_[2]=Path(p2,p3);
   Spc p4(w3,0);
   v_[3]=Path(p3,p4);
   Spc p5(w,h);
   v_[4]=Path(p4,p5);
}
else if (name=="X"){
   w=wB;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(2);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;
```

```
      Spc p1(0,0);
      Spc p2(0,h);
      Spc p3(w,0);
      Spc p4(w,h);
      v_[1]=Path(p1,p4);
      v_[2]=Path(p2,p3);
   }
   else if (name=="Y"){
      w=wB;
      h=hA;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      v_=V<Path>(3);
      f_=o;
      R wl=w+dA;
      l_=o+e1*wl;

      R w1=w*0.5;
      R h1=h*0.5;

      Spc p1(0,h);
      Spc p2(w1,h1);
      v_[1]=Path(p1,p2);
      Spc p3(w,h);
      v_[2]=Path(p2,p3);
      Spc p4(w1,0);
      v_[3]=Path(p2,p4);
   }
   else if (name=="Z"){
      w=wB;
      h=hA;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      v_=V<Path>(3);
      f_=o;
      R wl=w+dA;
      l_=o+e1*wl;

      Spc p1(0,0);
      Spc p2(0,h);
      Spc p3(w,0);
      Spc p4(w,h);
      v_[1]=Path(p2,p4);
      v_[2]=Path(p4,p1);
      v_[3]=Path(p1,p3);
   }
   else if (name=="a"){
      w=wa;
      h=ha;
```

```
   R h1r=0.3, h2r=0.6;
   R wre=0.2;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(7);
   f_=o;
   R h1=h1r*h;
   R h2=h2r*h;
   R w1=w*0.5;
   R we=w*(1+wre);
   R wl=we+dA;
   l_=o+e1*wl;
   Spc p1(0,h1);
   Spc p2(0,h2);
   v_[1]=Path(p1,p2);
   Spc p3(w1,h);
   v_[2]=Path::quarterEllipse(p2,e2,p3);
   Spc p4(w,h2);
   v_[3]=Path::quarterEllipse(p3,e1,p4);
   Spc q1(w1,0);
   v_[4]=Path::quarterEllipse(p1,-e2,q1);
   Spc q2(w,h1);
   v_[5]=Path::quarterEllipse(q1,e1,q2);
   Spc q3(w,h);
   v_[6]=Path(q3,q2);
   Spc q4(we,0);
   v_[7]=Path::quarterEllipse(q2,-e2,q4);
}
if (name=="au"){
   Wrt au("a");
   w=wa;
   R w1=0.3*w;
   R w2=0.7*w;
   h=ha;
   R r=0.5*wPeriod;
   Spc pc1(w1,hA-r);
   Spc pc2(w2,hA-r);
   V<Path> add(2);
   add[1]=Path::circle(pc1,r);
   add[2]=Path::circle(pc2,r);
   au&=add;
   *this=au;
}
else if (name=="b"){
   w=wa;
   h=ha;
   R h1r=0.3, h2r=0.6;
   h_=hA;
   d_=0;
   v_=V<Path>(6);
```

```
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;

   R h1=h1r*h;
   R h2=h2r*h;
   R w1=w*0.5;

   Spc p1(0,h_);
   Spc p2(0,0);
   v_[1]=Path(p1,p2);
   Spc p3(0,h2);
   Spc p4(w1,h);
   v_[2]=Path::quarterEllipse(p3,e2,p4);
   Spc p5(w,h2);
   v_[3]=Path::quarterEllipse(p4,e1,p5);
   Spc p6(w,h1);
   v_[4]=Path(p5,p6);
   Spc p7(w1,0);
   v_[5]=Path::quarterEllipse(p6,-e2,p7);
   Spc p8(0,h1);
   v_[6]=Path::quarterEllipse(p7,-e1,p8);
}
else if (name=="c"){
   w=wa;
   h=ha;
   R h1r=0.3, h2r=0.6;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(5);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;

   R h1=h1r*h;
   R h2=h2r*h;
   R w1=w*0.5;

   Spc p1(0,h1);
   Spc p2(0,h2);
   v_[1]=Path(p1,p2);
   Spc p3(w1,h);
   v_[2]=Path::quarterEllipse(p2,e2,p3);
   Spc p4(w,h2);
   v_[3]=Path::quarterEllipse(p3,e1,p4);
   Spc q1(w1,0);
   v_[4]=Path::quarterEllipse(p1,-e2,q1);
   Spc q2(w,h1);
   v_[5]=Path::quarterEllipse(q1,e1,q2);
}
```

```
else if (name=="d"){
   w=wa;
   h=ha;
   R h1r=0.3, h2r=0.6;
   R wre=0.2;
   h_=hA;
   d_=0;
   v_=V<Path>(7);
   f_=o;
   R h1=h1r*h;
   R h2=h2r*h;
   R w1=w*0.5;
   R we=w*(1+wre);
   R wl=we+dA;
   l_=o+e1*wl;
   Spc p1(0,h1);
   Spc p2(0,h2);
   v_[1]=Path(p1,p2);
   Spc p3(w1,h);
   v_[2]=Path::quarterEllipse(p2,e2,p3);
   Spc p4(w,h2);
   v_[3]=Path::quarterEllipse(p3,e1,p4);
   Spc q1(w1,0);
   v_[4]=Path::quarterEllipse(p1,-e2,q1);
   Spc q2(w,h1);
   v_[5]=Path::quarterEllipse(q1,e1,q2);
   Spc q3(w,h_);
   v_[6]=Path(q3,q2);
   Spc q4(we,0);
   v_[7]=Path::quarterEllipse(q2,-e2,q4);
}
else if (name=="e"){
   w=wa;
   h=ha;
   R h1r=0.3, h2r=0.5;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(6);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;

   R h1=h1r*h;
   R h2=h2r*h;
   R w1=w*0.5;

   Spc p1(0,h1);
   Spc p2(0,h2);
   v_[1]=Path(p1,p2);
   Spc p3(w1,h);
```

```
      v_[2]=Path::quarterEllipse(p2,e2,p3);
      Spc p4(w,h2);
      v_[3]=Path::quarterEllipse(p3,e1,p4);
      Spc q1(w1,0);
      v_[4]=Path::quarterEllipse(p1,-e2,q1);
      Spc q2(w,h1);
      v_[5]=Path::quarterEllipse(q1,e1,q2);
      Spc p5(0,h2);
      v_[6]=Path(p4,p5);
   }
   else if (name=="f"){
      w=wr;
      h=hA;
      R h1=ha, h2r=0.85;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      v_=V<Path>(3);
      f_=o;
      R h2=h2r*h;
      R w1=w*0.5;
      R wl=w+dA;
      l_=o+e1*wl;
      Spc p1(0,h1);
      Spc p2(w,h1);
      v_[1]=Path(p1,p2);
      Spc p3(w1,0);
      Spc p4(w1,h2);
      v_[2]=Path(p3,p4);
      Spc p5(w,h);
      v_[3]=Path::quarterEllipse(p4,e2,p5);
   }
   else if (name=="g"){
      w=wa;
      h=ha;
      R h1r=0.3, h2r=0.6;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=dg;
      v_=V<Path>(8);
      f_=o;
      R h1=h1r*h;
      R h2=h2r*h;
      R w1=w*0.5;
      R wl=w+dA;
      l_=o+e1*wl;
      Spc p1(0,h1);
      Spc p2(0,h2);
      v_[1]=Path(p1,p2);
      Spc p3(w1,h);
      v_[2]=Path::quarterEllipse(p2,e2,p3);
      Spc p4(w,h2);
```

```
      v_[3]=Path::quarterEllipse(p3,e1,p4);
      Spc q1(w1,0);
      v_[4]=Path::quarterEllipse(p1,-e2,q1);
      Spc q2(w,h1);
      v_[5]=Path::quarterEllipse(q1,e1,q2);
      Spc q3(w,h);
      R ys=-d_*0.4;
      Spc q4(w,ys);
      v_[6]=Path(q3,q4);
      Spc q5(w1,-d_);
      v_[7]=Path::quarterEllipse(q4,-e2,q5);
      Spc q6(0,ys);
      v_[8]=Path::quarterEllipse(q5,-e1,q6);

   }
   else if (name=="h"){
      w=wa;
      h=ha;
      R h2r=0.6;
      h_=hA;
      d_=0;
      v_=V<Path>(4);
      f_=o;
      R wl=w+dA;
      l_=o+e1*wl;
      R h2=h2r*h;
      R w1=w*0.5;

      Spc p1(0,h_);
      Spc p2(0,0);
      v_[1]=Path(p1,p2);
      Spc p3(0,h2);
      Spc p4(w1,h);
      v_[2]=Path::quarterEllipse(p3,e2,p4);
      Spc p5(w,h2);
      v_[3]=Path::quarterEllipse(p4,e1,p5);
      Spc p6(w,0);
      v_[4]=Path(p5,p6);
   }
   else if (name=="i"){
      w=wi;
      h=ha;
      h_=hA;
      d_=0;
      v_=V<Path>(2);
      f_=o;
      R wl=w+dA;
      l_=o+e1*wl;

      R w1=w*0.5;
```

```
      R r=wPeriod*0.5;

      Spc p1(w1,h);
      Spc p2(w1,0);
      v_[1]=Path(p1,p2);

    //  Spc cen(w1,h_-r);
      Spc cen(w1,h_);
      v_[2]=Path::circle(cen,w1);
   }
   else if (name=="j"){
      w=wj;
      h_=hA;
      h=ha;
      d_=dg;
      v_=V<Path>(3);
      f_=o;
      R wl=w+dA;
      l_=o+e1*wl;

      R w1=w*0.5;
      R r=wPeriod*0.5;

      Spc p1(w-r,h);
      Spc p2(w-r,0);
      v_[1]=Path(p1,p2);

      //Spc cen(w-r,h_-r);
      Spc cen(w-r,h_);
      v_[2]=Path::circle(cen,r);

      Spc p3(0,-d_);
      v_[3]=Path::quarterEllipse(p2,-e2,p3);
   }
   else if (name=="k"){
      w=wa;
      R h1=0.14;
      R h2=0.5;
      v_=V<Path>(3);
      f_=o;
      R wl=w+dA;
      l_=o+e1*wl;
      h_=hA;
      h=ha;
      d_=0;
      Spc p1(o);
      Spc p2(0,h_);
      Spc p3=p1.versus(p2,h1);
      Spc p4(w,h);
      Spc p5=p3.versus(p4,h2);
```

```
   Spc p6(w,0);
   v_[1]=Path(p1,p2);
   v_[2]=Path(p3,p4);
   v_[3]=Path(p5,p6);
}
else if (name=="l"){
   w=wj;
   h_=hA;
   h=hA;
   d_=0;
   v_=V<Path>(2);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;

   R w1=w*0.5;
   R h1=h*0.14;

   Spc p1(0,h);
   Spc p2(0,h1);
   v_[1]=Path(p1,p2);
   Spc p3(w,0);
   v_[2]=Path::quarterEllipse(p2,-e2,p3);
}
else if (name=="m"){
   w=wm;
   R hr=0.75;
   h_=hA;
   h=ha;
   d_=0;
   v_=V<Path>(7);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;

   R w1=w*0.25;
   R w2=w*0.5;
   R w3=w*0.75;
   R h1=h*hr;

   Spc p1(0,0);
   Spc p2(0,h);
   v_[1]=Path(p1,p2);
   Spc q1(0,h1);
   Spc q2(w1,h);
   v_[2]=Path::quarterEllipse(q1,e2,q2);
   Spc q3(w2,h1);
   v_[3]=Path::quarterEllipse(q2,e1,q3);
   Spc q4(w3,h);
   v_[4]=Path::quarterEllipse(q3,e2,q4);
```

```
   Spc q5(w,h1);
   v_[5]=Path::quarterEllipse(q4,e1,q5);
   Spc f1(w2,0);
   v_[6]=Path(q3,f1);
   Spc f2(w,0);
   v_[7]=Path(q5,f2);
}
else if (name=="n"){
   w=wa;
   R hr=0.75;
   h_=hA;
   h=ha;
   d_=0;
   v_=V<Path>(4);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;

   R w1=w*0.5;
   R h1=h*hr;

   Spc p1(0,0);
   Spc p2(0,h);
   v_[1]=Path(p1,p2);
   Spc q1(0,h1);
   Spc q2(w1,h);
   v_[2]=Path::quarterEllipse(q1,e2,q2);
   Spc q3(w,h1);
   v_[3]=Path::quarterEllipse(q2,e1,q3);
   Spc f1(w,0);
   v_[4]=Path(q3,f1);
}
else if (name=="o"){
   w=wa;
   h_=hA;
   h=ha;
   d_=0;
   v_=V<Path>(1);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;
   R h1=h*0.5;
   R w1=w*0.5;
   Spc pc(w1,h1);
   Path el=Path::ellipse(pc,w1,h1,32);
   v_[1]=el;
}
if (name=="ou"){
   Wrt au("o");
   w=wa;
```

```
      R w1=0.3*w;
      R w2=0.7*w;
      h=ha;
      R r=0.5*wPeriod;
      Spc pc1(w1,hA-r);
      Spc pc2(w2,hA-r);
      V<Path> add(2);
      add[1]=Path::circle(pc1,r);
      add[2]=Path::circle(pc2,r);
      au&=add;
      *this=au;
   }
   else if (name=="p"){
      w=wa;
      R h1r=0.3, h2r=0.6;
      h_=ha;
      h=h_;
      d_=dg;
      v_=V<Path>(6);
      f_=o;
      R h1=h1r*h;
      R h2=h2r*h;
      R w1=w*0.5;
      R wl=w+dA;
      l_=o+e1*wl;

      Spc q1(0,h);
      Spc q2(0,-d_);
      v_[1]=Path(q1,q2);

      Spc p1(0,h2);
      Spc p2(w1,h);
      v_[2]=Path::quarterEllipse(p1,e2,p2);
      Spc p3(w,h2);
      v_[3]=Path::quarterEllipse(p2,e1,p3);
      Spc p4(w,h1);
      v_[4]=Path(p3,p4);
      Spc p5(w1,0);
      v_[5]=Path::quarterEllipse(p4,-e2,p5);
      Spc p6(0,h1);
      v_[6]=Path::quarterEllipse(p5,-e1,p6);
   }
   else if (name=="q"){
      w=wa;
      R h1r=0.3, h2r=0.6;
      h_=ha;
      h=h_;
      d_=dg;
      v_=V<Path>(6);
      f_=o;
```

```
   R h1=h1r*h;
   R h2=h2r*h;
   R w1=w*0.5;
   R wl=w+dA;
   l_=o+e1*wl;

   Spc q1(w,h);
   Spc q2(w,-d_);
   v_[1]=Path(q1,q2);
   Spc p1(0,h2);
   Spc p2(w1,h);
   v_[2]=Path::quarterEllipse(p1,e2,p2);
   Spc p3(w,h2);
   v_[3]=Path::quarterEllipse(p2,e1,p3);
   Spc p4(w,h1);
   Spc p5(w1,0);
   v_[4]=Path::quarterEllipse(p4,-e2,p5);
   Spc p6(0,h1);
   v_[5]=Path::quarterEllipse(p5,-e1,p6);
   v_[6]=Path(p1,p6);
}
else if (name=="r"){
   w=wr;
   R h1r=0.6;
   h_=ha;
   h=h_;
   d_=0;
   v_=V<Path>(2);
   f_=o;
   R h1=h1r*h;
   R wl=w+dA;
   l_=o+e1*wl;
   Spc q1(0,0);
   Spc q2(0,h);
   v_[1]=Path(q1,q2);
   Spc p1(0,h1);
   Spc p2(w,h);
   v_[2]=Path::quarterEllipse(p1,e2,p2);
}
else if (name=="s"){
   w=wa;
   R h1r=0.25;
   R h2r=0.5;
   R h3r=0.75;
   h_=ha;
   R h1=h1r*h_;
   R h2=h2r*h_;
   R h3=h3r*h_;
   R w1=w*0.5;
   d_=0;
```

```
   v_=V<Path>(6);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;
   Vec v=e1*w;
   Vec vh=v*0.5;
   Spc p1(0,h3);
   Spc p2(w1,h_);
   v_[1]=Path::quarterEllipse(p1,e2,p2);
   Spc p3(w,h3);
   v_[2]=Path::quarterEllipse(p2,e1,p3);
   Spc q1(w1,h2);
   v_[3]=Path::quarterEllipse(p1,-e2,q1);
   Spc q2(w,h1);
   v_[4]=Path::quarterEllipse(q1,e1,q2);
   Spc q3(w1,0);
   v_[5]=Path::quarterEllipse(q2,-e2,q3);
   Spc q4(0,h1);
   v_[6]=Path::quarterEllipse(q3,-e1,q4);
}
else if (name=="t"){
   w=wr;
   R h1=ha, h2r=0.15;
   h_=hA;
   h=h_;
   d_=0;
   v_=V<Path>(3);
   f_=o;
   R h2=h2r*h;
   R w1=w*0.5;
   R wl=w+dA;
   l_=o+e1*wl;
   Spc p1(0,h1);
   Spc p2(w,h1);
   v_[1]=Path(p1,p2);
   Spc p3(w1,h);
   Spc p4(w1,h2);
   v_[2]=Path(p3,p4);
   Spc p5(w,0);
   v_[3]=Path::quarterEllipse(p4,-e2,p5);
}
else if (name=="u"){
   w=wa;
   h=ha;
   h_=h; w_=w; u_=Spc(w_,h_);
   R h1r=0.3;
   R wre=0.2;
   d_=0;
   v_=V<Path>(5);
   f_=o;
```

```
   R h1=h1r*h_;
   R w1=w*0.5;
   R we=w*(1+wre);
   R wl=we+dA;
   l_=o+e1*wl;
   Spc p1(0,h1);
   Spc p2(0,h);
   v_[1]=Path(p1,p2);
   Spc q1(w1,0);
   v_[2]=Path::quarterEllipse(p1,-e2,q1);
   Spc q2(w,h1);
   v_[3]=Path::quarterEllipse(q1,e1,q2);
   Spc q3(w,h_);
   v_[4]=Path(q3,q2);
   Spc q4(we,0);
   v_[5]=Path::quarterEllipse(q2,-e2,q4);
}
if (name=="uu"){
   Wrt au("u");
   w=wa;
   R w1=0.3*w;
   R w2=0.7*w;
   h=ha;
   R r=0.5*wPeriod;
   Spc pc1(w1,hA-r);
   Spc pc2(w2,hA-r);
   V<Path> add(2);
   add[1]=Path::circle(pc1,r);
   add[2]=Path::circle(pc2,r);
   au&=add;
   *this=au;
}
else if (name=="v"){
   w=wa;
   h=ha;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(2);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;
   R w1=w*0.5;
   Spc p1(0,h_);
   Spc p2(w1,0);
   v_[1]=Path(p1,p2);
   Spc p3(w,h_);
   v_[2]=Path(p2,p3);
}
else if (name=="w"){
   w=wm;
```

```
    h=ha;
    h_=h; w_=w; u_=Spc(w_,h_);
    d_=0;
    v_=V<Path>(4);
    f_=o;
    R wl=w+dA;
    l_=o+e1*wl;
    R w1=w*0.25;
    R w2=w*0.5;
    R w3=w*0.75;
    Spc p1(0,h);
    Spc p2(w1,0);
    v_[1]=Path(p1,p2);
    Spc p3(w2,h);
    v_[2]=Path(p2,p3);
    Spc p4(w3,0);
    v_[3]=Path(p3,p4);
    Spc p5(w,h);
    v_[4]=Path(p4,p5);
 }
 else if (name=="x"){
    w=wa;
    h=ha;
    h_=h; w_=w; u_=Spc(w_,h_);
    d_=0;
    v_=V<Path>(2);
    f_=o;
    R wl=w+dA;
    l_=o+e1*wl;
    Spc p1(0,0);
    Spc p2(0,h);
    Spc p3(w,0);
    Spc p4(w,h);
    v_[1]=Path(p1,p4);
    v_[2]=Path(p2,p3);
 }
 else if (name=="y"){
    w=wa;
    h=ha;
    h_=h; w_=w; u_=Spc(w_,h_);
    d_=dg;
    v_=V<Path>(3);
    f_=o;
    R wl=w+dA;
    l_=o+e1*wl;
    R w1=w*0.5;
    Spc p1(0,h);
    Spc p2(w1,0);
    v_[1]=Path(p1,p2);
    Spc p3(w,h);
```

```
   v_[2]=Path(p2,p3);
   Spc p4(0,-d_);
   v_[3]=Path(p2,p2-p3,p4,-e1);
}
else if (name=="z"){
   w=wa;
   h=ha;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(3);
   f_=o;
   R wl=w+dA;
   l_=o+e1*wl;

   Spc p1(0,0);
   Spc p2(0,h);
   Spc p3(w,0);
   Spc p4(w,h);
   v_[1]=Path(p2,p4);
   v_[2]=Path(p4,p1);
   v_[3]=Path(p1,p3);
}
else if (name=="n1"){
   w=wj;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(2);
   R hr=0.8;
   R h1=h*hr;

   Spc p1(0,h1);
   Spc p2(w,h);
   Spc p3(w,0);
   v_[1]=Path(p1,p2);
   v_[2]=Path(p2,p3);
}
else if (name=="n2"){
   w=wa;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   R h1r=1.-uR;
   R h2r=1.-2*uR;
   R h3r=1-3*uR;
   R h1=h1r*h, h2=h2r*h, h3=h3r*h;
   R w1=w*0.5;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(4);
```

```
      Spc p1(0,h1);
      Spc p2(w1,h);
      v_[1]=Path::quarterEllipse(p1,e2,p2);
      Spc p3(w,h2);
      v_[2]=Path::quarterEllipse(p2,e1,p3);
      Spc p4(w,h3);
      Spc p5(0,0);
      v_[3]=Path::bezier(V<Spc>("",p3,p4,p5),8);
         // ideal for the present purpose
      Spc p6(w,0);
      v_[4]=Path(p5,p6);
   }
   else if (name=="n3"){
      w=wa;
      h=hA;
      h_=h; w_=w; u_=Spc(w_,h_);
      R w1=0.5*w;
      R w2=0.25*w;
      R h1=0.8*h;
      R h2=0.55*h;
      R h3=0.25*h;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(6);
      Spc p1(0,h1);
      Spc p2(w1,h);
      v_[1]=Path::quarterEllipse(p1,e2,p2);
      Spc p3(w,h1);
      v_[2]=Path::quarterEllipse(p2,e1,p3);
      Spc p4(w2,h2);
      v_[3]=Path::quarterEllipse(p3,-e2,p4);
      Spc p5(w,h3);
      v_[4]=Path::quarterEllipse(p4,-e1,p5);
      Spc p6(w1,0);
      v_[5]=Path::quarterEllipse(p5,-e2,p6);
      Spc p7(0,h3);
      v_[6]=Path::quarterEllipse(p6,-e1,p7);
   }
   else if (name=="n4"){
      w=wa;
      h=hA;
      h_=h; w_=w; u_=Spc(w_,h_);
      R h1=0.5*h, h2=0.25*h;
      R w1=w*0.75;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(3);
      Spc p1(w1,h);
      Spc p2(0,h2);
      v_[1]=Path(p1,p2);
```

```
   Spc p3(w,h2);
   v_[2]=Path(p2,p3);
   Spc q1(w1,h1);
   Spc q2(w1,0);
   v_[3]=Path(q1,q2);
}
else if (name=="n5"){
   w=wa;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   R h1=0.5*h ;
   R h2=0.15*h ;
   R w1=w*0.5;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(4);
   Spc p1(w*0.8,h);
   Spc p2(0,h);
   v_[1]=Path(p1,p2);
   Spc p3(0,h1);
   v_[2]=Path(p2,p3);
   Spc p4(w,0.25*h);
   Vec dir1=1.5*e1+e2;
   v_[3]=Path(p3,dir1,p4,-e2);
   Spc p5(0,h2);
   Vec dir2=-1.5*e1+e2;
   v_[4]=Path(p4,-e2,p5,dir2);
}
else if (name=="n6"){
   w=wa;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   R h1=0.8*h ;
   R w1=w*0.5;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(4);
   R a=w1;
   R b=ha*0.45;
   Spc pc(a,b);
   v_[1]=Path::ellipse(pc,a,b);
   Spc p1(0,b);
   Spc p2(0,h1);
   v_[2]=Path(p1,p2);
   Spc p3(w1,h);
   v_[3]=Path::quarterEllipse(p2,e2,p3);
   Spc p4(w,h1);
   v_[4]=Path::quarterEllipse(p3,e1,p4);
}
else if (name=="n7"){
```

```
      w=wa;
      h=hA;
      h_=h; w_=w; u_=Spc(w_,h_);
      R h1=0.8*h ;
      R w1=w*0.5;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(3);
      Spc p1(0,h1);
      Spc p2(0,h);
      v_[1]=Path(p1,p2);
      Spc p3(w,h);
      v_[2]=Path(p2,p3);
      v_[3]=Path(p3,o);
   }
   else if (name=="n8"){
      w=wa;
      h=hA;
      h_=h; w_=w; u_=Spc(w_,h_);
      R h1=0.75*h ;
      R h2=0.25*h ;
      R w1=w*0.5;
      f_=o;
      l_=Spc(w+dA,0);
      R a=0.5*w;
      R b=0.25*h;
      Spc c1(w1,h1);
      v_=V<Path>(2);
      v_[1]=Path::ellipse(c1,a,b);
      Spc c2(w1,h2);
      v_[2]=Path::ellipse(c2,a,b);
   }
   else if (name=="n9"){
      w=wa;
      h=hA;
      h_=h; w_=w; u_=Spc(w_,h_);
      R h1=0.2*h ;
      R w1=w*0.5;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(4);
      R a=w1;
      R b=ha*0.45;
      Spc pc(a,h-b);
      v_[1]=Path::ellipse(pc,a,b);
      Spc p1(w,h-b);
      Spc p2(w,h1);
      v_[2]=Path(p1,p2);
      Spc p3(w1,0);
      v_[3]=Path::quarterEllipse(p2,-e2,p3);
```

```
      Spc p4(0,h1);
      v_[4]=Path::quarterEllipse(p3,-e1,p4);
   }
   else if (name=="n0"){
      w=wa;
      h=hA;
      h_=h; w_=w; u_=Spc(w_,h_);
      R w1=w*0.5;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(1);
      R a=w1;
      R b=h*0.5;
      Spc pc(a,b);
      v_[1]=Path::ellipse(pc,a,b);
   }
   else if (name=="alpha"){
      w=wA;
      h=ha;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(5);
      R w1=0.33*w;
      R w2=0.66*w;
      R h1=0.5*h;
      Spc p1(w,h);
      Spc p2(w1,0);
      v_[1]=Path(p1,p2,-e1,0.33);
      Spc p3(0,h1);
      v_[2]=Path::quarterEllipse(p2,-e1,p3);
      Spc p4(w1,h);
      v_[3]=Path::quarterEllipse(p3,e2,p4);
      Spc p5(w2,h1);
      Vec dir=e1-4*e2;
      v_[4]=Path(p4,e1,p5,dir);
      Spc p6(w,0);
      v_[5]=Path(p5,dir,p6,e1);
   }
   else if (name=="beta"){
      w=wa;
      h=hA;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=dg;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(6);
      R w1=0.5*w;
      R h1=0.5*h;
```

```
   Spc p1(0,-d_);
   Spc p2(w1,h);
   v_[1]=Path(p1,p2,e1,0.8);
   Spc p3(w,0.75*h);
   v_[2]=Path::quarterEllipse(p2,e1,p3);
   Spc p4(w1,h1);
   v_[3]=Path::quarterEllipse(p3,-e2,p4);
   Spc p5(w,0.25*h);
   v_[4]=Path::quarterEllipse(p4,e1,p5);
   Spc p6(w1,0);
   v_[5]=Path::quarterEllipse(p5,-e2,p6);
   Spc p7(0.4*w1,0.1*h);
   v_[6]=Path(p6,-e1,p7,-e1+e2);
}
else if (name=="gamma"){
   w=wa;
   h=ha;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=dg;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(5);
   R w1=0.3*w;
   R w2=0.6*w;
   R h1=0.5*h;
   Spc p1(0,0.8*h);
   Spc p2(w1,h);
   v_[1]=Path(p1,e1+2*e2,p2,e1);
   Spc p3(w2,0);
   v_[2]=Path(p2,e1,p3,-e2);
   Spc p4(w1,-d_);
   v_[3]=Path(p3,-e2,p4,-e1);
   Spc p5(o);
   v_[4]=Path(p4,-e1,p5,e2);
   Spc p6(w,h);
   v_[5]=Path(p6,p5,-e2);
}
else if (name=="delta"){
   w=wa;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(6);
   R w1=0.5*w;
   R w2=0.8*w;
   R h1=ha;
   R h2=0.5*h1;
   Spc p1(w*0.8,h1*0.9); // meeting point
```

```
      Spc p2(w,h2);
      Path pa1(p1,p2,-e2,0.5);
      v_[1]=pa1;
      Vec di=pa1.firstDir();
      Spc p0(w,h);
      Path pa2(p1,-di,p0,e1);
      v_[2]=pa2;
      Spc p3(w1,0);
      v_[3]=Path::quarterEllipse(p2,-e2,p3);
      Spc p4(0,h2);
      v_[4]=Path::quarterEllipse(p3,-e1,p4);
      Spc p5(w1,h1);
      v_[5]=Path::quarterEllipse(p4,e2,p5);
      v_[6]=Path(p5,e1,p1,di);
   }
   else if (name=="epsilon"){
      w=wa;
      h=ha;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(6);
      R w1=0.5*w;
      R h1=0.5*h;
      Spc p1(w,0.75*h);
      Spc p2(w1,h);
      v_[1]=Path(p1,-e1+2*e2,p2,-e1);
      Spc p3(0,0.75*h);
      v_[2]=Path::quarterEllipse(p2,-e1,p3);
      Spc p4(w1,h1);
      v_[3]=Path::quarterEllipse(p3,-e2,p4);
      Spc p5(0,0.25*h);
      v_[4]=Path::quarterEllipse(p4,-e1,p5);
      Spc p6(w1,0);
      v_[5]=Path::quarterEllipse(p5,-e2,p6);
      Spc p7(w,0.25*h);
      v_[6]=Path(p6,e1,p7,e1+2*e2);
   }
   else if (name=="theta"){
      w=wA;
      h=hA;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(6);
      R w1=0.5*w;
      R h1=0.5*h;
      R a=8;
```

```
   Vec dir1=e1+2*e2;
   Spc p1(o);
   Spc p2(w1,0);
   R g1=4,g2=1.5;
   v_[1]=Path::cubCur(p1,dir1,g1,p2,e1,g2);
   Spc p3(w1,h);
   g1=1.5; g2=1.5;
   v_[2]=Path::cubCur(p2,e1,g1,p3,-e1,g2);
   Spc p4(w,0.8*h);
   g1=2.5; g2=4;
   v_[3]=Path::cubCur(p3,-e1,g1,p4,e1*5+e2*2,g2);
}
else if (name=="phi"){
   w=wA;
   h=ha;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=dg;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(6);
   R w1=0.5*w;
   R h1=0.5*h;
   Spc p1(w*0.4,h);
   Spc p2(0,h1);
   v_[1]=Path::quarterEllipse(p1,-e1,p2);
   Spc p3(w1,0);
   v_[2]=Path::quarterEllipse(p2,e2,p3);
   Spc p4(w,h1);
   v_[3]=Path::quarterEllipse(p3,e1,p4);
   Spc p5(w*0.7,h*0.9);
   v_[4]=Path::quarterEllipse(p4,e2,p5);
   Spc p6(w1,-d_);
   v_[5]=Path::quarterEllipse(p5,-e1,p6);

}
else if (name=="psi"){
   w=wA;
   h=ha;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=dg;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(4);
   R w1=0.6*w;
   Spc p1(0,h*0.8);
   Spc p2(w*0.075,h);
   Vec d1=e1+e2*2.5;
   v_[1]=Path(p1,d1,p2,e1);
   Spc p3(w1,0);
   v_[2]=Path::cubCur(p2,e1,2,p3,e1,2);
```

```
   Spc p4(w,h);
   Vec d2=e1+e2*2;
   v_[3]=Path::cubCur(p3,e1,1.75,p4,d2,2);
   v_[4]=Path(Spc(w1,h),Spc(w1,-d_));
}
else if (name=="omega"){
   w=wA;
   h=ha;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(6);
   R w1=0.2*w;
   R w2=0.5*w;
   R w3=0.8*w;
   R h0=0.5*h;
   R h1=0.75*h;
   Spc p1(w1,h);
   Spc p2(0,h0);
   v_[1]=Path(p1,p2,-e2);
   Spc p3(0.25*w,0);
   v_[2]=Path::quarterEllipse(p2,-e2,p3);
   Spc p4(w2,h1);
   v_[3]=Path(p3,e1,p4,e2);
   Spc p5(0.75*w,0);
   v_[4]=Path(p4,-e2,p5,e1);
   Spc p6(w,h0);
   v_[5]=Path(p5,e1,p6,e2);
   Spc p7(w3,h);
   v_[6]=Path(p7,p6,-e2);
}
else if (name=="rho"){
   w=wa;
   h=ha;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=dg;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(2);
   R w1=0.5*w;
   R h1=0.5*h;
   v_[1]=Path::ellipse(Spc(w1,h1),w1,h1);
   Spc p1(0,h1);
   Spc p2(w1,-d_);
   v_[2]=Path::cubCur(p1,-e2,2.5,p2,-e1*2.5-e2,2);
}
else if (name=="lambda"){
   w=wa;
   h=hA;
```

```
    h_=h; w_=w; u_=Spc(w_,h_);
    d_=0;
    f_=o;
    l_=Spc(w+dA,0);
    v_=V<Path>(3);
    R w1=0.5*w;
    R h1=0.5*h;
    Spc p1(0,h);
    Spc p2(w,0);
    Spc pc(w1,h1);
    v_[1]=Path(pc,p1,-e1,0.5);
    v_[2]=Path(pc,p2,e1,0.5);
    v_[3]=Path(o,pc);
 }
 else if (name=="chi"){
    w=wa;
    h=ha;
    h_=h; w_=w; u_=Spc(w_,h_);
    d_=dg;
    f_=o;
    l_=Spc(w+dA,0);
    v_=V<Path>(3);
    Spc p1(0,h);
    Spc p2(0,-d_);
    Spc p3(w,h);
    Spc p4(w,-d_);
    Spc pc=p1.cen(p4);
    v_[1]=Path(pc,p1,-e1,0.5);
    v_[2]=Path(pc,p4,e1,0.5);
    v_[3]=Path(p2,p3);
 }
 else if (name=="sigma"){
    w=wa;
    h=0.9*ha;
    R h2=1.2*ha;
    h_=h; w_=w; u_=Spc(w_,h_);
    d_=0;
    f_=o;
    l_=Spc(w+dA,0);
    v_=V<Path>(3);
    R w1=0.5*w;
    R h1=0.5*h;
    Spc p1(0,h);
    Spc p2(0,0);
    Spc p3(w,h);
    Spc p4(w,0);
    Spc pc=p2.cen(p3);
    v_[1]=Path::ellipse(pc,w1,h1);
    Spc q1(0,h1);
    Spc q2(w1,h2);
```

```
   v_[2]=Path::cubCur(q1,e2,1.5,q2,e1,1.5);
   Spc q3(w,h2);
   v_[3]=Path::cubCur(q2,e1,1.5,q3,e1+e2,1.5);
}
else if (name=="mu"){
   w=wA;
   h=ha;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=dg;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(6);
   R w1=0.5*w;
   R h1=0.5*h;
   Spc p1(0,-d_);
   R wr=0.20;
   R hr=0.25;
   R w0=wr*w;
   R h0=hr*h;
   R wm=w0+(w-w0)*0.5;
   Spc p2(w0,h0);
   v_[1]=Path(p1,p2,e2);
   Spc p3(w0,h);
   v_[2]=Path(p2,p3);
   Spc p4(wm,0);
   v_[3]=Path::quarterEllipse(p2,-e2,p4);
   Spc p5(w-w0,h0);
   v_[4]=Path::quarterEllipse(p4,e1,p5);
   Spc p6(w-w0,h);
   v_[5]=Path(p5,p6);
   Spc p7(w,0);
   v_[6]=Path::quarterEllipse(p5,-e2,p7);
}
else if (name=="nu"){
   w=wA;
   h=ha;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(4);
   R wr=0.2;
   R hr=0.85;
   R w0=wr*w;
   R h0=hr*h;
   Spc p1(0,h0);
   Spc p2(w0,h);
   v_[1]=Path(p1,p2,e1);
   R w2=2*w0;
   Spc p3(w2,h0);
```

```
    v_[2]=Path::quarterEllipse(p2,e1,p3);
    Spc p4(w2,0);
    v_[3]=Path(p3,p4);
    v_[4]=Path(p4,Spc(w,h),e1+e2*3);
 }
 else if (name=="upsilon"){
    w=wA;
    h=ha;
    h_=h; w_=w; u_=Spc(w_,h_);
    d_=0;
    f_=o;
    l_=Spc(w+dA,0);
    v_=V<Path>(5);
    R wr=0.2;
    R hr=0.85;
    R w0=wr*w;
    R h0=hr*h;
    Spc p1(0,h0);
    Spc p2(w0,h);
    v_[1]=Path(p1,p2,e1);
    R w2=2*w0;
    R w3=3.25*w0;
    Spc p3(w2,h0);
    v_[2]=Path::quarterEllipse(p2,e1,p3);
    Spc p4(w3,0);
    v_[3]=Path::quarterEllipse(p3,-e2,p4);
    Spc p5(w,0.5*h);
    v_[4]=Path::quarterEllipse(p4,e1,p5);
    Spc p6(0.9*w,h);
    v_[5]=Path(p6,p5,-e2);
 }
 else if (name=="eta"){
    w=wA;
    h=ha;
    h_=h; w_=w; u_=Spc(w_,h_);
    d_=dg;
    f_=o;
    l_=Spc(w+dA,0);
    v_=V<Path>(6);
    R wr=0.2;
    R hr=0.75;
    R w0=wr*w;
    R h0=hr*h;
    Spc p1(0,h);
    Spc p2(w0,h0);
    v_[1]=Path(p1,p2,e2);
    Spc p3(w0,0);
    v_[2]=Path(p2,p3);
    R wm=w*0.5;
    Spc p4(wm,h);
```

```
      v_[3]=Path::quarterEllipse(p2,e2,p4);
      R wl=w-w0;
      Spc p5(wl,h0);
      v_[4]=Path::quarterEllipse(p4,e1,p5);
      Spc p6(wl,0);
      v_[5]=Path(p5,p6);
      Spc p7(w,-d_);
      v_[6]=Path(p6,p7,e1);
   }
   else if (name=="tau"){
      w=wA;
      h=ha;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(4);
      R dh=0.15*h;
      Spc p1(0,h-dh);
      Spc p2(w*0.5,h);
      Spc p3(w,h+dh);
      Vec d1=e1+e2;
      Vec d2=e1;
      v_[1]=Path(p1,d1,p2,d2);
      v_[2]=Path(p2,d2,p3,d1);
      R h2=0.25*w;
      Spc q1(0.5*w,h2);
      Spc q2(w,h2);
      v_[3]=Path(p2,q1);
      v_[4]=Path::arc(q1,q1.cen(q2),AxVec(Angle(180,DEG),3));
   }
   else if (name=="pi"){
      w=wA;
      h=ha;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(7);
      R w1=0.33*w;
      R w2=0.67*w;
      R dh=0.2*h;
      R dw=0.1*w;
      R h1=0.5*h;
      Spc p1(0,h-dh);
      Spc p2(w1,h);
      Spc p3(w2,h);
      Spc p4(w,h+dh);
      v_[1]=Path::quarterEllipse(p1,e2,p2);
      v_[2]=Path(p2,p3);
```

```
   v_[3]=Path::quarterEllipse(p3,e1,p4);
   Spc q2(w1,h1);
   Spc q3(w2,h1);
   v_[4]=Path(p2,q2);
   v_[5]=Path(p3,q3);
   Spc r2(dw,0);
   Spc r3(w-dw,0);
   v_[6]=Path::quarterEllipse(r2,e1,q2);
   v_[7]=Path::quarterEllipse(r3,-e1,q3);
}
else if (name=="zeta"){
   w=wa;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=dg;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(5);
   R w1=0.5*w;
   R dh8=h/8;
   R dh4=h/4;
   Spc p1(w1,h);
   Spc p2(0,h-dh8);
   v_[1]=Path::quarterEllipse(p1,-e1,p2);
   Spc p3(w,h-dh4);
   v_[2]=Path::quarterEllipse(p2,-e2,p3);
   Spc p4(0,h*0.5);
   v_[3]=Path::quarterEllipse(p3,-e1,p4);
   Spc p5(w,0);
   v_[4]=Path::cubCur(p4,-e2,1.6,p5,-e2,1.25);
   Spc p6(0,-d_);
   v_[5]=Path::quarterEllipse(p5,-e2,p6);
}
else if (name=="xi"){
   w=wa;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=dg;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(7);
   R w1=0.5*w;
   R dh8=h/9;
   R dh4=dh8*2;
   R dh2=dh4*2;
   Spc p1(w1,h);
   Spc p2(dh8,h-dh8);
   v_[1]=Path::quarterEllipse(p1,-e1,p2);
   Spc p3(w,h-dh4);
   v_[2]=Path::quarterEllipse(p2,-e2,p3);
```

```
   Spc q1(dh8,h-dh4-dh8);
   v_[3]=Path::quarterEllipse(p3,-e1,q1);
   Spc q2(w,h-dh2);
   v_[4]=Path::quarterEllipse(q1,-e2,q2);
   Spc p4(0,h*0.5-dh8);
   v_[5]=Path::quarterEllipse(q2,-e1,p4);
   Spc p5(w,0);
   v_[6]=Path::cubCur(p4,-e2,1.6,p5,-e2,1.25);
   Spc p6(0,-d_);
   v_[7]=Path::quarterEllipse(p5,-e2,p6);
}
else if (name=="kappa"){
   w=wA;
   h=ha;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(5);
   R wr=0.2;
   R hr=0.85;
   R w0=wr*w;
   R h0=hr*h;
   Spc p1(0,h0);
   Spc p2(w0,h);
   v_[1]=Path(p1,p2,e1);
   R w2=2*w0;
   Spc p3(w2,h0);
   v_[2]=Path::quarterEllipse(p2,e1,p3);
   Spc p4(w2,0);
   v_[3]=Path(p3,p4);
   Spc p5=p4.versus(p3,0.5);
   Spc p6(w,h);
   Path pa1(p5,p6,e1);
   v_[4]=pa1;
   Vec d1=pa1.firstDir();
   Spc p7(w,0);
   v_[5]=Path::cubCur(p5,d1,2,p7,e1,2);
}
else if (name=="omicron"){
   w=wa;
   h=ha;
   h_=h; w_=w; u_=Spc(w_,h_);
   R w1=w*0.5;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(1);
   R a=w1;
   R b=h*0.5;
   Spc pc(a,b);
```

```
      v_[1]=Path::ellipse(pc,a,b);
   }
   else if (name=="iota"){
      w=wr;
      h=ha;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(2);
      R r=w*0.5;
      Spc q0(0,h);
      Spc q1(0,r);
      Spc q2(w,r);
      v_[1]=Path(q0,q1);
      v_[2]=Path::arc(q1,q1.cen(q2),AxVec(Angle(180,DEG),3));
   }
   else if (name=="Gamma"){
      w=wa;
      h=hA;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(3);
      Spc p0(o);
      Spc p1(0,h);
      Spc p2(w,h);
      Spc p3(w,0.9*h);
      v_[1]=Path(p0,p1);
      v_[2]=Path(p1,p2);
      v_[3]=Path(p2,p3);
   }
   else if (name=="Lambda"){
      w=wA;
      h=hA;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(3);
      R dw=0.1*w;
      R w1=0.45*w;
      Spc p0(o);
      Spc p1(w1,h);
      Spc p2(w-dw,0);
      Spc p3(w,0);
      v_[1]=Path(p0,p1);
      v_[2]=Path(p1,p2);
      v_[3]=Path(p2,p3);
```

```
}
else if (name=="Delta"){
   w=wA;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(3);
   R dw=0.1*w;
   R w1=0.45*w;
   Spc p0(o);
   Spc p1(w1,h);
   Spc p2(w,0);
   v_[1]=Path(p0,p1);
   v_[2]=Path(p1,p2);
   v_[3]=Path(p2,p0);
}
else if (name=="Pi"){
   w=wa;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(3);
   R dw=0.1*w;
   R w1=0.45*w;
   Spc p0(o);
   Spc p1(0,h);
   Spc p2(w,h);
   Spc p3(w,0);
   v_[1]=Path(p0,p1);
   v_[2]=Path(p1,p2);
   v_[3]=Path(p2,p3);
}
else if (name=="Sigma"){
   w=wA;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(4);
   R w1=0.5*w;
   R h1=0.5*h;
   Spc p1(w,h);
   Spc p2(0,h);
   Spc p3(w1,h1);
   Spc p4(0,0);
```

```
   Spc p5(w,0);
   v_[1]=Path(p1,p2);
   v_[2]=Path(p2,p3);
   v_[3]=Path(p3,p4);
   v_[4]=Path(p4,p5);
}
else if (name=="Theta"){
   w=wA;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   R w1=w*0.5;
   R h1=h*0.5;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(2);
   R a=w1;
   R b=h*0.5;
   Spc pc(a,b);
   v_[1]=Path::ellipse(pc,a,b);
   v_[2]=Path(Spc(0.33*w,h1),Spc(0.67*w,h1));
}
else if (name=="Xi"){
   w=wA;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   R w1=w*0.5;
   R h1=h*0.5;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(3);
   v_[1]=Path(Spc(0,h),Spc(w,h));
   v_[2]=Path(Spc(0.33*w,h1),Spc(0.67*w,h1));
   v_[3]=Path(Spc(0,0),Spc(w,0));
}
else if (name=="Omega"){
   w=wA;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   R w1=w*0.5;
   R h1=h*0.5;
   R dw=0.25*w;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(6);
   Spc p0(o);
   Spc p1(dw,0);
   v_[1]=Path(p0,p1);
```

```
   Spc p2(0,h1);
   v_[2]=Path(p1,p2,e2);
   Spc p3(w1,h);
   v_[3]=Path::quarterEllipse(p2,e2,p3);
   Spc p4(w,h1);
   v_[4]=Path::quarterEllipse(p3,e1,p4);
   Spc p5(w-dw,0);
   v_[5]=Path(p5,p4,e2);
   Spc p6(w,0);
   v_[6]=Path(p5,p6);

}
else if (name=="Phi"){
   w=wA;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   R w1=w*0.5;
   R h1=h*0.8;
   R h2=h*0.5;
   R h3=h*0.2;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(5);
   Spc p1(w1,h);
   Spc p2(w1,0);
   v_[1]=Path(p1,p2);
   Spc q1(w1,h3);
   Spc q2(0,h2);
   v_[2]=Path::quarterEllipse(q1,-e1,q2);
   Spc q3(w1,h1);
   v_[3]=Path::quarterEllipse(q2,e2,q3);
   Spc q4(w,h2);
   v_[4]=Path::quarterEllipse(q3,e1,q4);
   v_[5]=Path::quarterEllipse(q4,-e2,q1);

}
else if (name=="Psi"){
   w=wA;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   R w1=w*0.5;
   R h1=h*0.33;
   f_=o;
   l_=Spc(w+dA,0);
   v_=V<Path>(3);
   Spc p1(w1,h);
   Spc p2(w1,0);
   v_[1]=Path(p1,p2);
```

```
   Spc q1(w1,h1);
   Spc q2(0,h);
   v_[2]=Path::quarterEllipse(q1,-e1,q2);
   Spc q3(w,h);
   v_[3]=Path::quarterEllipse(q1,e1,q3);
}
else if (name=="space"){
   w=wB;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   v_=V<Path>(0);
   f_=o;
   l_=Spc(w,0);
   h_=0;
   d_=0;
}
else if (name=="hs" || name=="halfspace"){
   w=wj;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   v_=V<Path>(0);
   f_=o;
   l_=Spc(w,0);
   h_=0;
   d_=0;
}
else if (name=="backspace"){
   R wSpace=wB+dA;
   v_=V<Path>(0);
   f_=o;
   l_=Spc(-wSpace,0);
   h_=0;
   d_=0;
}
else if (name=="backspace2"){
   R wSpace=wA+dA;
   v_=V<Path>(0);
   f_=o;
   l_=Spc(-wSpace,0);
   h_=0;
   d_=0;
}
else if (name=="period"){
   w=wPeriod;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   R w1=0.5*w;
   f_=o;
   l_=Spc(w+dA,0);
   d_=0;
```

```
   Spc cen(w1,w1);
   v_=V<Path>(1);
   v_[1]=Path::circle(cen,w1);
}
else if (name=="colon"){
   w=wPeriod;
   h=ha;
   h_=h; w_=w; u_=Spc(w_,h_);
   R w1=0.5*w;
   f_=o;
   l_=Spc(w+dA,0);
   d_=0;
   Spc cen1(w1,w1);
   Spc cen2(w1,h-w1);
   v_=V<Path>(2);
   v_[1]=Path::circle(cen1,w1);
   v_[2]=Path::circle(cen2,w1);
}
else if (name=="comma"){
   w=wPeriod;
   h=ha;
   R w1=0.5*w;
   f_=o;
   l_=Spc(w+dA,0);
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0.75*dg;
   Spc cen(w1,w1);
   Spc p1(w,w1);
   Spc p2(0,-d_);
   v_=V<Path>(2);
   v_[1]=Path::circle(cen,w1);
   v_[2]=Path(p2,p1,e2);
}
else if (name=="semicolon"){
   w=wPeriod;
   h=ha;
   R w1=0.5*w;
   f_=o;
   l_=Spc(w+dA,0);
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0.75*dg;
   Spc cen0(w1,h-w1);
   Spc cen(w1,w1);
   Spc p1(w,w1);
   Spc p2(0,-d_);
   v_=V<Path>(3);
   v_[1]=Path::circle(cen0,w1);
   v_[2]=Path::circle(cen,w1);
   v_[3]=Path(p2,p1,e2);
}
```

```
else if (name=="exclamation_mark"){
   w=wPeriod;
   h=hA;
   R w1=0.5*w;
   f_=o;
   l_=Spc(w+dA,0);
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(2);
   Spc p1(w1,h);
   Spc p2(w1,3*w);
   v_[1]=Path(p1,p2);
   Spc cen(w1,w1);
   v_[2]=Path::circle(cen,w1);
}
else if (name=="question_mark"){
   w=wa;
   h=hA;
   R w1=0.5*w;
   f_=o;
   l_=Spc(w+dA,0);
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(4);
   R h1=h*0.8;

   Spc p1(0,h1);
   Spc p2(w1,h);
   v_[1]=Path::quarterEllipse(p1,e2,p2);
   Spc p3(w,h1);
   v_[2]=Path::quarterEllipse(p2,e1,p3);
   R r=0.5*wPeriod;
   Spc p4(w1,6*r);
   v_[3]=Path::cubCur(p3,-e2,1.5,p4,-e2,1.5);
   Spc cen(w1,r);
   v_[4]=Path::circle(cen,r);
}
else if (name=="dash"){
   w=wa;
   h=ha;
   f_=o;
   l_=Spc(w+dA,0);
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(1);
   R h1=0.5*h;
   R dw=0.2*w;
   Spc p1(dw,h1);
   Spc p2(w-dw,h1);
   v_[1]=Path(p1,p2);
```

```
}
else if (name=="l_bracket"){
   w=wj;
   h=hA;
   f_=o;
   l_=Spc(w+dA,0);
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(2);
   R h1=h*0.5;
   Spc p1(w,h);
   Spc p2(0,h1);
   Spc p3(w,0);
   v_[1]=Path(p1,p2,-e2,0.6);
   v_[2]=Path(p3,p2,e2,0.6);
}
else if (name=="r_bracket"){
   w=wj;
   h=hA;
   f_=o;
   l_=Spc(w+dA,0);
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(2);
   R h1=h*0.5;
   Spc p1(0,h);
   Spc p2(w,h1);
   Spc p3(0,0);
   v_[1]=Path(p1,p2,-e2,0.6);
   v_[2]=Path(p3,p2,e2,0.6);
}
else if (name=="slash"){
   w=wr;
   h=hA;
   f_=o;
   l_=Spc(w+dA,0);
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   v_=V<Path>(1);
   Spc p1(w,h);
   v_[1]=Path(o,p1);
}
else if (name=="plus"){
   w=wa;
   h=ha;
   f_=o;
   l_=Spc(w+dA,0);
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   R w1=w*0.5;
```

```
   R h1=h*0.5;
   v_=V<Path>(2);
   Spc p1(w,h);
   v_[1]=Path(Spc(0,h1),Spc(w,h1));
   v_[2]=Path(Spc(w1,0),Spc(w1,h));
}
else if (name=="quote"){
   w=wj;
   h=hA;
   h_=h; w_=w; u_=Spc(w_,h_);
   v_=V<Path>(1);
   f_=o;
   l_=Spc(dA,0);
   d_=0;
   Spc t1(0,hA);
   Spc t2(0,hA-w);
   v_[1]=Path(t1,t2);
}
else if (name=="dquote"){
   w=wj;
   h=hA;
   v_=V<Path>(2);
   f_=o;
   l_=Spc(dA+w,0);
   h_=h; w_=w; u_=Spc(w_,h_);
   d_=0;
   Spc p1(0,h);
   Spc p2(0,h-w);
   v_[1]=Path(p1,p2);
   Spc p3(w,h);
   Spc p4(w,h-w);
   v_[2]=Path(p3,p4);
}
else if (name=="tilde"){
   w=wA;
   R wh=0.5*w;
   v_=V<Path>(2);
   f_=o;
   l_=Spc(w+dA,0);
   h_=hA*1.25;
   d_=0;
   Spc t1(0,h_);
   Spc t2(wh,h_);
   Spc t3(w,h_);
   Vec e1(1,0);
   Angle a(40,DEG);
   Vec v1=e1*AxVec(a,3);
   Vec v2=e1*AxVec(-a,3);
   v_[1]=Path(t1,v1,t2,v2);
   v_[2]=Path(t2,v2,t3,v1);
```

```
   }
   else if (name=="schlipf"){
      w=wr;
      h_=hA;
      R h1=0.8*h_;
      v_=V<Path>(2);
      f_=o;
      l_=Spc(w+dA,0);
      d_=0;
      Spc p1(0,h_);
      Spc p2(w,h_);
      Spc p3(w,h1);
      v_[1]=Path(p1,p2);
      v_[2]=Path(p2,p3);
   }
   else if (name=="ast"){
      w=wa;
      h=ha;
      h_=h; w_=w; u_=Spc(w_,h_);
      d_=0;
      f_=o;
      l_=Spc(w+dA,0);
      v_=V<Path>(4);
      Spc p1(w,h);
      Spc pc=o.cen(p1);
      R r=CpmRootX::inf<R>(w,h)*0.5;
      r*=0.9;
      Vec v1=e1*r;
      Vec v2=v1*rot45;
      Vec v3=v2*rot45;
      Vec v4=v3*rot45;
      v_[1]=Path(pc+v1,pc-v1);
      v_[2]=Path(pc+v2,pc-v2);
      v_[3]=Path(pc+v3,pc-v3);
      v_[4]=Path(pc+v4,pc-v4);
   }
}

//////////////////////////// class LineArt ////////////////////////////////

LineArt::LineArt( V<Word> const&  vw, R r, Z n):s_(),t_(),p_(),w_()
{
   Z vwd=vw.dim();
   if (vwd<1) return;
   Z mL=3;
   Word loc("LineArt(V<Word>,R,Z)");
   CPM_MA
   V<Wrt> res(vwd);
   for (Z i=1;i<=vwd;++i){
      Wrt ai(vw[i]);
```

```
      ai.thc_(r,n);
      if (i>1) ai.to_(res[i-1].last());
      res[i]=ai;
   }
   w_=res;
   CPM_MZ
}

LineArt LineArt::text(V<Wrt> const& vw)
{
   Z vwd=vw.dim();
   V<Wrt> res(vwd);
   for (Z i=1;i<=vwd;++i){
      Wrt ai=vw[i];
      if (i>1) ai.to_(res[i-1].last());
      res[i]=ai;
   }
   return LineArt(res);
}

R LineArt::h()const
{
   Z n=w_.dim();
   R_Vector r(n);
   for (Z i=1;i<=n;++i) r[i]=w_[i].h();
   return r.sup();
}

LineArt& LineArt::operator*=(Group const& g)
{
   Z i;
   for (i=1;i<=dimS();i++) s_[i]*=g;
   for (i=1;i<=dimT();i++) t_[i]*=g;
   for (i=1;i<=dimP();i++) p_[i]*=g;
   for (i=1;i<=dimW();i++) w_[i]*=g;
   return *this;
}

LineArt& LineArt::operator+=(Vec const& v)
{
   Z i;
   for (i=1;i<=dimS();i++) s_[i]+=v;
   for (i=1;i<=dimT();i++) t_[i]+=v;
   for (i=1;i<=dimP();i++) p_[i]+=v;
   for (i=1;i<=dimW();i++) w_[i]+=v;
   return *this;
}

LineArt& LineArt::scl_(Spc const& fix, R fac)
{
```

```
   Z i;
   for (i=1;i<=dimS();i++) s_[i].scl_(fix,fac);
   for (i=1;i<=dimT();i++) t_[i].scl_(fix,fac);
   for (i=1;i<=dimP();i++) p_[i].scl_(fix,fac);
   for (i=1;i<=dimW();i++) w_[i].scl_(fix,fac);
   return *this;
}

LineArt& LineArt::thc_(R r, Z n)
{
   Z i;
   V<Path> coll;
   for (i=1;i<=dimP();i++) coll&=p_[i].thc(r,n);
   for (i=1;i<=dimW();i++) w_[i].thc_(r,n);
   p_=coll;
   return *this;
}

bool LineArt::prnOn(ostream& str)const
{
   cpmwat;
   cpmp(s_);
   cpmp(t_);
   cpmp(p_);
   cpmp(w_);
   return true;
}

bool LineArt::scanFrom(istream& str)
{
   cpms(s_);
   cpms(t_);
   cpms(p_);
   cpms(w_);
   return true;
}

LineArt LineArt::disk(Spc const& p, Vec const& h, R r, Z n)
{
   Z mL=3;
   Word loc("LineArt::disk(Spc,Vec,R,Z)");
   CPM_MA
   cpmassert(n>2,loc);
   cpmassert(r>0,loc);
   Angle phi(360./n,DEG);
   AxVec rot(phi);
   Vec rho(r,0);
   if (Spc::dimension()>2){
      // only then h enters the computation
      R ha=h.absVal();
```

```
      cpmassert(ha>0,loc);
      V<Vec> trip=h.base();
      Vec ax=trip[1];
      rho=trip[2]*r; // transversal vector
      rot=AxVec(phi,ax);
   }
   AxVec ri=rot;
   V<Vec> vv(n);
   vv[1]=rho;
   Z i;
   for (i=2;i<=n;++i){
      vv[i]=vv[i-1]*rot;
   }
   V<Triangle> bas(n);
   for (i=1;i<=n;++i){
      Spc pi=p+vv[i];
      Spc qi=p+vv(i+1,CpmArrays::CYCLIC);
      bas[i]=Triangle(pi,qi,p);
   }
   CPM_MZ
   return LineArt(bas);
}

LineArt LineArt::sph(Spc const& p, R r, Z n)
{
   const Z points=16;
   const Z steps=40;
   Z mL=3;
   Word loc("LineArt::sph(Spc,R,Z)");
   CPM_MA
   Vec e1(1,0,0);
   LineArt res=disk(p,e1,r,n);
   if (e1.dim()==2){
      CPM_MZ
      return res;
   }
   V<Vec> vv=Vec::homUnitVec(points,steps);
   for (Z i=1;i<=points;++i){
      res&=disk(p,vv[i],r,n);
   }
   CPM_MZ
   return res;
}

LineArt LineArt::cone(Spc const& p, Vec const& h, R r, Z n)
{
   Z mL=3;
   Word loc("LineArt::cone(Spc,Vec,R,Z)");
   CPM_MA
   cpmassert(n>2,loc);
```

```
   cpmassert(r>0,loc);
   if (Spc::dimension()==2){
      Spc c=p+h;
      Vec e=h;
      e.nor_();
      Vec n=e.rot90();
      Vec lat=n*r;
      Spc a=p-lat;
      Spc b=p+lat;
      Triangle tri(a,b,c);
      CPM_MZ
      return LineArt(tri);
   }
   R ha=h.absVal();
   cpmassert(ha>0,loc);
   Spc top=p+h;
   V<Vec> trip=h.base();
   Vec ax=trip[1];
   Vec rho=trip[2]*r; // transversal vector
   Angle phi(360./n,DEG);
   AxVec rot(phi,ax);
   AxVec ri=rot;
   V<Vec> vv(n);
   vv[1]=rho;
   Z i;
   for (i=2;i<=n;++i){
      vv[i]=vv[i-1]*rot;
   }
   V<Triangle> sur(n);
   V<Triangle> bas(n);
   for (i=1;i<=n;++i){
      Spc pi=p+vv[i];
      Spc qi=p+vv(i+1,CpmArrays::CYCLIC);
      sur[i]=Triangle(pi,qi,top);
      bas[i]=Triangle(pi,qi,p);
   }
   V<Triangle> all=sur&bas;
   CPM_MZ
   return LineArt(all);
}

LineArt LineArt::arr(Spc const& p1, Spc const& p2, R h, R r, Z n)
{
   Vec e=p2-p1;
   e.nor_();
   Spc p2Graph=p2-e*(h*0.5);
      // to avoid dismatches on pixel basis of
      // the end of the line and the top of the cone
   Vec hh=e*h;
   Spc pb=p2-hh;
```

```
   LineArt res=cone(pb,hh,r,n);
   return res&=Path(p1,p2Graph);
}

void LineArt::mark(Camera const& ca)const
{
   Z i;
   bool stereo=ca.is3D()&&!ca.isMono();
   Color cc=ca.is2D() ? WHITE : BLACK;
   Color c=stereo ? INVISIBLE : cc;
   for (i=1;i<=dimS();i++) ca.mark(s_[i],c);
   for (i=1;i<=dimT();i++) ca.mark(t_[i],c);
   for (i=1;i<=dimP();i++) ca.mark(p_[i],c);
   for (i=1;i<=dimW();i++) w_[i].mark(ca,c);
}
```

110 cpmlineartx.h

```
//? cpmlineartx.h
//? Status of work 2008-10-25.
//?
/*
Description: Defining classes Wrt (Writing) and LineArt
   which both represent spatial graphical objects which use a
   Camera-object for visualization. Intended to allow us to
   create figures to be placed in scientific articles.
   Includes the definition of a vector font (latin and greek).
   The size of code needed to define the letters and numbers was the
   reason to remove the classes from the files of the dim family.
*/

   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmRoot::Word;

   using CpmArrays::V;

   using CPM_DIM::Group;
   using CPM_DIM::Spc;
   using CPM_DIM::Vec;
   using CPM_DIM::AxVec;
   using CPM_DIM::Skeleton;
   using CPM_DIM::Path;

   using CPM_CAMERA::Camera;

//doc namespace CpmDimx{

//////////////////////// struct IndCon ////////////////////////////////////

struct IndCon{ // indexing control
   // Allows to shift indexes relative to their default position. No
   // limitation in range, so right indexes may be changed to left ones.
   R fac;
   R dh; // dh>0 shifts vertically upwards
   R dw; // dw>0 shifts horizontally to the right
   IndCon():fac(1),dh(0),dw(0){}
   IndCon(Word const& w){
      if (w=="sub"){ fac=0.5; dh=-0.4; dw=0;}
      else if (w=="sup"){ fac=0.5; dh=-0.2; dw=0;}
      else{ fac=1; dh=0; dw=0;}
   }
   // A constructor IndCon(R,R,R) is considered error prone:
   // It is easy to forget whether dh is the second or the third
   // argument. By contrast, saying ic.dh=-0.3; ic.dw=-0.1 is
```

```
   // perfectly clear.
};

//////////////////////// class Wrt ////////////////////////////////////

class Wrt{
   // Wrt=writing, actually printable symbols
   V<Path> v_;
   Spc f_; // first
   Spc l_; // last,
      // the intention is that the next letter should be placed
      // with its first point just at the last point of its
      // predecessor. Also connection point for lower
      // indexes.
   Spc u_;
      // upper right corner;
      // connection point for upper indexes
   R h_; // maximum hight over base line
      // last_ + Vec(0,h_) is the position to which
      // one may place a
   R w_; // maximum width of the visible object
      // not including the reserved space to the next
      // Wrt-object.
   R d_; // descender, underlength

   void operator&=(Path const& p){ v_&=p;}
      // adding to the path list without considering
      // influence on f_,l_,h_,d_. Thus to be used with care,
      // thus private
   void operator&=(V<Path> const& vp){ v_&=vp;}
      // previous function extended to arrays

   typedef Wrt Type;
public:
   CPM_IO
   explicit Wrt(Word const& w);
      // creating the letters of Latin and Greek alphabets together
      // with punctuation and some special characters.
      // examples Wrt("gamma"), Wrt("Gamma"), Wrt("ou") (o as 'Umlaut'),
      // Wrt("hs") (half-space), Wrt(n5) (5 , 'number 5')
      // All implemented cases can be infered from the implementation in
      // cpmlineartcpp.h
   Word nameOf()const{ return "Wrt";}
   Wrt():v_(),f_(),l_(),h_(0),w_(0),d_(0){}
   Spc fir()const{ return f_;}
      //: first
   Spc last()const{ return l_;}
      //: last
   Spc upp()const{ return u_;}
      //: upper
```

```
Spc rigLowCor()const{ return f_+e1()*w_;}
   //: right lower corner
Spc rigUppCor()const{ return u_;}
   //: right upper corner
Spc leftLowCor()const{ return f_;}
   //: left lower corner
Spc leftUppCor()const{ return u_-e1()*w_;}
   //: left upper corner
Vec spc()const{ return l_-rigLowCor();}
   //: space
   // space between the right lower corner of the symbol box
   // and the next symbol box
Vec e1()const{ Vec e=l_-f_; e.nor_(); return e;}
   //: e1
   // unit vector in writing direction
Vec e2()const{ Vec t=(u_-f_)-e1()*w_; t.nor_(); return t;}
   //: e2
   // unit vector from bottom to top of the symbols, perpendicular
   // to e1.
V<Vec> bas()const;
   //: basis
CpmArrays::V<Path> const& v()const{ return v_;}
   //: v
   // read access to main data v_
R wTot()const{ return (f_-l_).absVal();}
   //: w total
   // with of the object together with space to the
   // next Wrt-object
R h()const{ return h_;}
   //: h (height)
R w()const{ return w_;}
   //: w (width)
Wrt& operator *=(Group const& g);
   // general Euclidian transformation
Wrt& operator +=(Vec const& v);
   // translation
Wrt& operator *=(AxVec const& av);
   // rotation arround point fir_
Wrt& operator *=(R scl);
   // scaling
Wrt& scl_(Spc const& fix, R fac);
   //: scale
Wrt& thc_(R r, Z n);
   //: thick
   // This function is active only if r>0 and n >=2. If this is
   // satisfied, v_ gets replaced by the union of n shifted
   // copies, where the n shift vectors have length r and
   // result from equispaced points on a circle (not disk!).
   // This is a brut force method to simulate a line thickness
   // of 2*r. Works fine. But the suitable n has to be found by
```

```
      // experimentation for given r. If n is too small, there remain
      // voids, if it is too large, the function takes too much time.
   Wrt& vec_();
      //: vector
      // endowes *this with a vector arrow above it
   Wrt& sub_(V<Wrt> const& vw, IndCon const& ic=IndCon("sub"));
      //: subscript
      // Changes *this into a combined symbol which carries a
      // subscript (placed to the right of the main symbol).
      // The connection point for an upper (right) index of
      // this combined symbol is the same as for the original
      // symbol. Thus by applying sup_ after sub_ it is possible
      // to endow *this with and upper and a lower index such
      // that both indexes start at the same horizontal position.
   Wrt& sup_(V<Wrt> const& vw, IndCon const& ic=IndCon("sup"));
      //: superscript
      // See ecplanation to function sub_. See preceeding
      // struct IndCom for an explanation of the argument ic.
      //
   Wrt& to_(Spc const& p);
      //: to
      // Moves the point f_ of *this to point p by translation.
      // This is the method for designing Words. The word "good"
      // will be created like this:
      // Wrt a1("g");
      // Wrt a2("o"); a2.to_(a1);
      // Wrt a3(a2); a3.to_(a2);
      // Wrt a4("d"); a4.to_(a3);
      // Now the letters a2,a3,a4 are placed relative to a1 as to be
      // expected from the word "good". To make a graphical object
      // out of this word, we define
      // LineArt x; x&=a1; x&=a2; x&=a3; x&=a4;
      // Making use of the member functions of class LineArt,
      // we can shift, rotate, and scale the "graphical word" x
      // as a whole. Also creation of x from the Word list
      // V<Word> vw("","g","o","o","d") can be done by the
      // constructor LinList(V<Word>,R,Z).
      // The function CpmCamerax::Camera::mark(LineArt)
      // (Camerax stands for Camera2 or Camera3) allows to map the word
      // into the C+- frame buffer from which it can be displayed on
      // screen and written into a ppm-image-file.
      // Single Wrt-objects can be treated alike by the function
      // CpmCamerax::Camera::mark(Wrt,Color).
   void mark(Camera const& ca, Color const& color)const;
};

//////////////////////// class LineArt ////////////////////////////////
class LineArt{ // a variety of graphical objects in combination
   // similiar to WireFrame, but containing additional
   // objects of graphical relevance. See topic combining LineArts
```

```
   // for typical usage.

   V<Skeleton> s_;
   V<Triangle> t_;
   V<Path> p_;
   V<Wrt> w_; // writing

   typedef LineArt Type;
public:
// infrastructure
   CPM_IO
   Word nameOf()const{ return "LineArt";}

// constructing LineArts
LineArt(V<Skeleton> const& s, V<Triangle> const& t,
   V<Path> const& p, V<Wrt> const& w ):
   s_(s),t_(t),p_(p),w_(w){}

LineArt():s_(),t_(),p_(),w_(){}

explicit LineArt( V<Skeleton> const& s):
   s_(s),t_(),p_(),w_(){}

// construction from arrays
explicit LineArt( V<Triangle> const& t):
   s_(),t_(t),p_(),w_(){}

explicit LineArt( V<Path> const& p):
   s_(),t_(),p_(p),w_(){}

explicit LineArt( V<Wrt> const& w):
   s_(),t_(),p_(),w_(w){}

// construction from individuals
explicit LineArt( Triangle const& t):
   s_(),t_(V<Triangle>("",t)),p_(),w_(){}

explicit LineArt( Skeleton const& s):
   s_(V<Skeleton>("",s)),t_(),p_(),w_(){}

explicit LineArt( Path const& p):
   s_(),t_(),p_(V<Path>("",p)),w_(){}

explicit LineArt( Wrt const& w):
   s_(),t_(),p_(),w_(V<Wrt>("",w)){}

explicit LineArt( V<Word> const&  vw, R r=0, Z n=32);
// a list of names of Wrt-object will form a graphical word by
// proper concatenation, and optional thickening of the lines.
```

```
// transforming LineArts
LineArt& operator *=(Group const& g);

LineArt& operator +=(Vec const& a);

LineArt& operator *=(AxVec const& A)
{ return operator*=(Group(A));}

Spc fir()const{ return w_.dim()>0 ? w_.fir().fir() : Spc();}
//: first
// position of the first among the letters, Spc() if
// there is none

LineArt& to_(Spc const& p){ return (*this)+=(p-fir());}
//: to
// functionality just as that of the corresponding
// function Wrt::to_(Spc)

LineArt& scl_(Spc const& fix, R fac);
//: scale
// It is important to fix here a fixed point since an instance
// of LinArt does not have a distinguished point as instances
// of Wrt do.

LineArt& thc_(R r, Z n);
//: thick
// This function is active only if r>0 and n >=2. If this is
// satisfied, v_ gets replaced by the union of n shifted
// copies, where the n shift vectors have length r and
// result from equispaced points on a circle (not disk!).
// This is a brute force method to simulate a line thickness
// of 2*r. Works fine. But the suitable n has to be found by
// experimentation for given r. If n is too small, there remain
// voids, if it is too large, the function takes too much time.
// Small void will be filled when the capuring camera is asked
// to perform line art processing (see function Camera::dis).

// combining LineArts
//
// This allows to create parts in isolation, move them around,
// transform them at will, and then chain them together
// to a final piece that then can be displayed or/and filed
// e.g. as an illustration for a scientific article.
//    LineArt la1=...;
//    LineArt la2=...;
//       ...
//    LineArt la;
//    la&=la1; la&=la2; ...
//    Camera ca=...;
//    la.mark(ca);
```

```
//    ca.setWriteOnShow(true);
//    bool lineArtProcessing=true;
//    ca.dis(lineArtProcessing);
//    cpmwait(30,2);

LineArt& operator &=(LineArt const& la)
   { s_&=la.s_; t_&=la.t_; p_&=la.p_; w_&=la.w_; return *this;}

LineArt& operator &=(Skeleton const& s)
   { s_&=s; return *this;}

LineArt& operator &=(Triangle const& t)
   { t_&=t; return *this;}

LineArt& operator &=(Path const& p)
   { p_&=p; return *this;}

LineArt& operator &=(Wrt const& w)
   { w_&=w; return *this;}

LineArt& operator &=(V<Skeleton> const& s)
   { s_&=s; return *this;}

LineArt& operator &=(V<Triangle> const& t)
   { t_&=t; return *this;}

LineArt& operator &=(V<Path> const& p)
   { p_&=p; return *this;}

LineArt& operator &=(V<Wrt> const& w)
   { w_&=w; return *this;}

// access
const V<Skeleton>& cs()const{ return s_;}
V<Skeleton>& cs_(){ return s_;}

const V<Triangle>& ct()const{ return t_;}
V<Triangle>& ct_(){ return t_;}

const V<Path>& cp()const{ return p_;}
V<Path>& cp_(){ return p_;}

const V<Wrt>& cw()const{ return w_;}
V<Wrt>& cw_(){ return w_;}

R h()const;
//: h
// Text height. Maximum height of a Wrt in w_, i.e.
// maximum of w_[i].h(), 1 <= i <= w_.dim().
// Notice thus that h refers only to the height of the letters involved
```

```
// and not to the height of other geometrical structures.

Z dimS()const{ return s_.dim();}
Z dimT()const{ return t_.dim();}
Z dimP()const{ return p_.dim();}
Z dimW()const{ return w_.dim();}

void mark(Camera const& cam)const;
//: mark
// writing *this to the frame the store. The Camera object cam controls
// the conversion to 2D, or a 2D transformation if we are in 2D
// already. Display to screen is under the responsibility of
// the Camera ca and gets invoked by ca.dis();
// See also comment to topic header 'combining LineArts'.

static LineArt disk(Spc const& p, Vec const& h, R r, Z n=16);
//: disk
// p is the center of the disk area which is a regular n-polygon
// of radius r. h is a normal vector of this area (length
// does not matter).

static LineArt sph(Spc const& p, R r, Z n=16);
//: sphere
// made of only three disks, which should be sufficient

static LineArt cone(Spc const& p,  Vec const& h, R r, Z n=16);
//: cone
// similar as Skeleton::cone but with more geometric parameters.
// p is the center of the base area and p+h is the top of the
// cone. In 3D the base plane is a regular n-polygon of radius r.
// In 2D the object actually is a triangle, so that n has no influence.

static LineArt arr(Spc const& p1, Spc const& p2, R h, R r, Z n=16);
//: arrow
// The arrow extends from p1 to p2. The arrowhead is a cone, the top of
// which is at p2. The height of the conic arrowhead is h (no vector in
// this case) and the radius of its base plane is r.

static LineArt text(V<Wrt> const& vw);
//: text
// Takes the position and orientation from the first Wrt in the argument
// list and forms the whole chain by shifting following symbols to the
// fitting positions. No rotation of symbols will be done.
};
```

111 cpmlogic1.h

```
//? cpmlogic1.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_LOGIC1_H_
#define CPM_LOGIC1_H_
/*
   Description: Class Version of Propositional Logic.
      First example of a class based on expression trees.
      Coding method for expressions taken from Chapter 8 of
      A. Koenig, B. Moo: Ruminations on C++, Addison-Wesley 1996
*/
#include <cpms.h>

namespace CpmMathFound{ // foundation of mathematics

   using namespace CpmStd;

   using CpmRoot::Word;
   using CpmRoot::Z;
   using CpmArrays::S;
   using CpmArrays::V;
   using CpmArrays::VV;
   using CpmRootX::B;
   using CpmRootX::Z1;

class Prop{ // atomic propositions
   // 'atomic' propositions characterized by a number
   // and a truth value. Suited as template argument for
   // PropLog<>
   B b_;
   Z1 i_;
   static Z count_;

public:
   typedef Prop Type;
   CPM_ORDER;
   CPM_IO;

   Prop():i_(1){} // no count activities
   explicit Prop(Z i):i_(i){if (i_>count_) count_=i_; if (i_<1) i_=1;}
      // creates a Prop x such that
      // x.toWord()="pk" where k is the character string denoting
      // the number i_, and x.eval()=false
   Prop(Z i, bool b):i_(i),b_(b)
      {if (i_>count_) count_=i_; if (i_<1) i_=1;}
      // as before but x.eval()=b_
   explicit Prop(Word w): i_(++count_){}
```

```
      // creates (independent of the value of w) a Prop which is
      // different from any Prop created before
   explicit Prop(Word w, bool b):i_(++count_),b_(b){}
      // as before but the truth value

   void setValue(bool b){ b_=b;}

   bool eval()const{ return b_;}
   Z index()const{ return i_;}
   Word toWord()const{ return "p"&i_.toWord();}
   Word nameOf()const{ return "Prop";}
   static Z count(){ return count_;}
};

//////////// class Expr_node<> ////////////////////////////////////////

template <class X>
class PropLog;

template <class X>
class Expr_node{ // expression nodes
   friend class PropLog<X>;
   Z use; // direct implementation of reference counting
protected:
   Expr_node():use(1){}
   virtual ~Expr_node(){}
   virtual void setValue(const V<B>&)=0;
   virtual Word toWord()const =0;
   virtual Word nameOf()const =0;
   virtual bool eval()const=0;
};

//////////// class PropLog ////////////////////////////////////////////
/* The mathematical structure of PropLog<X> is beautiful. I never found
it spelled out in full elegance.
We consider the class B as enriched by the operations 'and', 'or' and
'not'. We have the diagram:

             f1
     X ------------> B
  g1 |               | g2
     v       f2      v
  PropLog<X> ------> B

where f1 is arbitrary, g1 is given by the constructor
PropLog<X>(const X&) which inserts X into PropLog<X>, and f2,g2 is a
closure to a commutative diagram such that f2 and g2 are morphisms with
respect to the 'and','or', and 'not' operations.
f1 and f2 are named eval in our application.
Since PropLog<X> consists of expressions in X, eval as a function on X
```

```
gets recursively extended to whole PropLog<X>. This logic seems to
work always if only we consider the same operations on B and on
PropLog<X>.
Actually, it is the raison d'etre of expressions that their evaluation
is a well defined operation if the evaluation of the nodes is defined.

It looks to me that the interpretation of this structure as Logic
and, as such, as a foundation of mathematics is misleading. Here
the structure is coded in a formal language (which C++ is).

See cpmtestr.h, Test_logic<> for a verification of this structure
for selected points.

*/

template <class X> class Leaf;
template <class X> class Unary_node;
template <class X> class Binary_node;

template <class X>
class PropLog{ // propositional logic
   // creates propositional logic for the case that the
   // atomic formulae are of type X
   Expr_node<X> *p;

   PropLog(Expr_node<X>* p_):p(p_){}
      // not safe, thus private

   S<X> atoms;
      // set of different atomic formulas on which
      // the expression depends. This object is to be known
      // for generating all truth-value assignments
public:
// Koenigs stuff arround reference counting
   ~PropLog(){if (--p->use==0) delete p;}
      // destructor
   PropLog<X>& operator=(const PropLog<X>& rhs)
   { atoms=rhs.atoms; rhs.p->use++; if (--p->use==0) delete p; p=rhs.p;
    return *this;}
      // assignement
   PropLog(const PropLog<X>& t){ atoms=t.atoms;p=t.p; ++p->use;}
      // copy constructor

   PropLog(){ p=new Leaf<X>(X());}
      // default constructor, together with the previous 3 functions, the
      // value interface is implemented

   explicit PropLog(const X& x){ p=new Leaf<X>(x); atoms.add(x);}
      // constructor for a atomic proposition
```

```
static V< PropLog<X> > list( Z n);
   // creates a list of instances each of which is independent
   // af all instances created before

static VV< PropLog<X> > matrix(Z m, Z n);
   // creates a matrix of instances each of which is independent
   // af all instances created before

PropLog<X> non()const;
   // returns the negation of *this
PropLog<X> et(const PropLog<X>& rhs )const;
   // returns *this 'and' rhs
PropLog<X> vel(const PropLog<X>& rhs )const;
   // returns *this 'or' rhs
PropLog<X> aut(const PropLog<X>& rhs )const
{ return (non().et(rhs)).vel((*this).et(rhs.non()));}
   // returns *this 'exclusive or' rhs
PropLog<X> implies(const PropLog<X>& rhs )const{ return non().vel(rhs)\
;}
   // returns *this --> rhs
PropLog<X> iff(const PropLog<X>& rhs )const
{ return implies(rhs).et(rhs.implies(*this)); }

static PropLog<X> xorV(const V<PropLog<X> >& vp);
   // returns a proposition which
   // true iff exactly one of the vp[i] is true. This is not
   // simply vp[1].xor(vp[2]).xor(vp[3]),...... See
   // implementation code for the formal definition

static PropLog<X> orV(const V<PropLog<X> >& vp);
   // returns a proposition which is vp[1].vel(vp[2]).vel(vp[3]),...

static PropLog<X> andV(const V<PropLog<X> >& vp);
   // returns a proposition which is vp[1].et(vp[2]).et(vp[3]),...

typedef PropLog<X> Type;
CPM_IO
CPM_ORDER
   // IO and ORDER not yet implemented; not trivial but the lines seem
   // clear
Word toWord()const{ return p->toWord();}
   // textual representation of the instance *this
Word nameOf()const;
   // textual representation of the type or class
bool eval()const{ return p->eval();}
   // returns the truth value which results from the imported
   // truth-value assignment to the atoms
void setValue(const V<B>& vb){ p->setValue(vb);}
   // the atom at with at.index()=i on which *this depends
   // will get the truth value vb[i]. It was hard work to discover
```

```
      // how to access the leafs of expression *this if it is
      // finished. The solution is given by virtual functions
      // in the node classes.
   bool taut();
      // returns true iff *this is tautological

   bool implied(const V< PropLog<X> >& vp);
      // returns true if vp implies *this, or, expressed
      // differently, *this is a consequence (is implied,
      // thus the name) by vp.

   S<X> getAtoms()const{ return atoms;}
      // returns the set of atoms of *this
};

template <class X>
V< PropLog<X> > PropLog<X>::list( Z n)
{
   V< PropLog<X> > res(n);
   res[1]=PropLog<X>(X(""));
   for (Z i=2;i<=n;i++) res[i]=PropLog<X>(X(""));
   return res;
}

template <class X>
VV< PropLog<X> > PropLog<X>::matrix(Z m, Z n)
{
   static Z mL=4;
   static Word loc("PropLog<X>::matrix(Z m, Z n)");
   cpmmessage(mL,loc&" started");
   cpmassert(m>0,loc);
   cpmassert(n>0,loc);
   Z i,j,mn=m*n;
   Z k=0;
   V< PropLog<X> > resLin=list(mn);
   VV< PropLog<X> > res(m,n);
   for (i=1;i<=m;i++) for (j=1;j<=n;j++) res[i][j]=resLin[++k];
   cpmmessage(mL,loc&" done");
   return res;
}

template <class X>
Word PropLog<X>::nameOf()const
{
   X x;
   Word w1="Expression of Propositional Logic with propositions of type ";
   Word w2=x.nameOf();
   return w1&w2;
}
```

```
template <class X>
PropLog<X> PropLog<X>::non()const
{
   PropLog<X> res(new Unary_node<X>("non",*this));
   res.atoms=atoms;
   return res;
}

template <class X>
PropLog<X> PropLog<X>::et(const PropLog<X>& rhs)const
{
   PropLog<X> res(new Binary_node<X>("et",*this,rhs));
   res.atoms=atoms|rhs.atoms;
      // was & till 2006-06-25
   return res;
}

template <class X>
PropLog<X> PropLog<X>::vel(const PropLog<X>& rhs)const
{
   PropLog<X> res(new Binary_node<X>("vel",*this,rhs));
   res.atoms=atoms|rhs.atoms;
   return res;
}

class V_bool: public V<B>{ // array of bool-values, helper class
public:
   V_bool(const V<B>& vb):V<B>(vb){}
   V_bool(Z n):V<B>(n){}
   void next();
      // changes *this into the next instance
      // (same dim, according to lexicographic
      // order derived from false < true
   bool end()const; // returns true for the last vb
      // which is (true,true,...,true)
};

template <class X>
bool PropLog<X>::taut()
{
   Z mL=2;
   Word loc("PropLog<X>::taut()");
   cpmmessage(mL,loc&" started");
   Z counter=0;
   Z cStep=10000;
   Z count=0;
   Z dat=atoms.dim();
   Z d=atoms[dat].index();
   V_bool vb(d);
   Z cnt=X::count(); // static function
```

```
   cpmmessage(mL,"Number of Bits="&cpm(d)&", count="&cpm(cnt));
   for (;!vb.end();vb.next()){
      counter++;
      if (counter==cStep){
         count++;
         cpmmessage(mL+1,"count="&cpmwrite(count));
         counter-=cStep;
      }
      setValue(vb);
      if (!eval()) return false;
   }
   cpmmessage(mL,loc&" done");
   return true;
}

/*
template <class X>
bool PropLog<X>::implied(const V< PropLog<X> >& vpx)
{
   PropLog<X> known=andV(vpx);
   PropLog<X> toShow=known.implies(*this);
   return toShow.taut();
}
*/

template <class X>
bool PropLog<X>::implied(const V< PropLog<X> >& vpx)
// considerably faster than the previous trivial implementation
{
   Z mL=2;
   Word loc("PropLog<X>::implied(V< PropLog<X> >)");
   cpmmessage(mL,loc&" started");
   Z counter=0;
   Z cStep=10;
   Z count=0;
   S<X> s=atoms;
   V< PropLog<X> > vp=vpx;
   Z i,np=vp.dim();
   for (i=1;i<=np;i++){
     // s=(s|(vp[i].getAtoms())); // successive union
     s.add(vp[i].getAtoms()); // successive union
   }
   Z n=s.dim();
   Z ns=s[n].index();
   V_bool vb(ns);
   cpmmessage(mL,"Number of atoms="&cpmwrite(n)
   &", nBits="&cpmwrite(ns));
   bool b,bVal;
   bool first=true;
   bool last=false;
```

```
Label:
   counter++;
   if (counter==cStep){
      count++;
      cpmmessage(mL+1,"count="&cpmwrite(count));
      counter-=cStep;
   }
   if (first) first=false; else { vb.next(); last=vb.end();}
      // the initial truth-value assignment vb=(false,false,....false)
      // has to be taken as it is and not just replaced by vb.next()
   for (i=1;i<=np;i++){
      vp[i].setValue(vb);
      bVal=vp[i].eval();
      if (!bVal) goto Label;
         // then the truth-value assignment gives not true for all
         // premisses; so we can't learn from this truth-value assignment
         // and turn to the next one
   }
   // now the premisses are all true for the present values of vb
   setValue(vb);
   b=eval();
   if (!b){
      return false;
      // for present vb the premisses are true and *this is false
   }
   else if (!last){
      goto Label;
   }
   else{
      return true;
   }
}

template <class X>
PropLog<X> PropLog<X>::xorV(const V< PropLog<X> >& vp)
{
   Z i,j,k,d=vp.dim();
   if (d==0){
      return PropLog<X>();
   }
   else if (d==1){
      return vp[1];
   }
   else if (d==2) {
      PropLog<X> res=vp[1].aut(vp[2]);
      return res;
   }
   else if (d==3) {
      V<PropLog<X> > resP(d);
      resP[1]=vp[1].et(vp[2].non()).et(vp[3].non());
```

```
      resP[2]=vp[2].et(vp[3].non()).et(vp[1].non());
      resP[3]=vp[3].et(vp[1].non()).et(vp[2].non());
      return orV(resP);
   }
   else{ // the previous case in automated form
      V<PropLog<X> > resP(d);
      for (i=1;i<=d;i++){
         resP[i]=vp[i];
         k=i+1;
         for (j=1;j<d;j++){
            if (k>d) k=1;
            resP[i]=resP[i].et(vp[k].non());
            k++;
         }
      }
      return orV(resP);
   }
}

template <class X>
PropLog<X> PropLog<X>::orV(const V< PropLog<X> >& vp)
{
   Z i,d=vp.dim();
   if (d==0){
      return PropLog<X>();
   }
   else if (d==1){
      return vp[1];
   }
   else{
      PropLog<X> res=vp[1];
      for (i=2;i<=d;i++) res=res.vel(vp[i]);
      return res;
   }
}

template <class X>
PropLog<X> PropLog<X>::andV(const V< PropLog<X> >& vp)
{
   Z i,d=vp.dim();
   if (d==0){
      return PropLog<X>();
   }
   else if (d==1){
      return vp[1];
   }
   else{
      PropLog<X> res=vp[1];
      for (i=2;i<=d;i++) res=res.et(vp[i]);
      return res;
```

```
   }
}

////////////////////// class Leaf<> ////////////////////////

template <class X>
class Leaf: public Expr_node<X>{ // leafs in expression trees
   friend class PropLog<X>;
   X atom;
   Leaf(const X& x):atom(x){}
   void setValue(bool b_){ atom.setValue(b_);}
   Word toWord()const{ return "("&atom.toWord()&")";}
   Word nameOf()const{ return "node0";}
   bool eval()const{ return atom.eval();}
   void setValue(V<B> const&);
};

template <class X>
void Leaf<X>::setValue(const V<B>& vb)
{
   Z i=atom.index();
   Z vbd=vb.dim();
   if (i<1) cpmerror("Leaf<X>::setValue(const V<B>& vb): i<1, i="
      &cpm(i));
   if (i>vbd){
      cpmerror("Leaf<X>::setValue(const V<B>& vb): i>vb.dim(), i="
      &cpm(i)&" vb.dim()="&cpm(vbd) );
   }
   atom.setValue(vb[i]);
}

////////////////////// class Unary_node<> ////////////////////////////

template <class X>
class Unary_node: public Expr_node<X>{//unary nodes in expression trees
   friend class PropLog<X>;
   Word op;
   PropLog<X> opnd;
   Unary_node(const Word& op_,const PropLog<X>& opnd_):
      op(op_),opnd(opnd_){}
   Word toWord()const;
   Word nameOf()const{ return "node1";}
   bool eval()const;
   void setValue(const V<B>&);
};

template <class X>
Word Unary_node<X>::toWord()const
{
   return "("&op&opnd.toWord()&")";
```

```
}

template <class X>
bool Unary_node<X>::eval()const
{
   if (op=="non"){
      return !opnd.eval();
   }
   else{
      cpmerror("Unary_node<X>::eval() un-valid node type");
      return false;
   }
}

template <class X>
void Unary_node<X>::setValue(const V<B>& vb)
{
   opnd.setValue(vb);
}

////////////////////// class Binary_node<> ////////////////////////////

template <class X>
class Binary_node: public Expr_node<X>{// binary nodes in expression trees
   friend class PropLog<X>;
   Word op;
   PropLog<X> left;
   PropLog<X> right;
   Binary_node(const Word& op_, const PropLog<X>& le,
      const PropLog<X>& ri): op(op_),left(le),right(ri){}
   Word toWord()const;
   Word nameOf()const{ return "node2";}
   bool eval()const;
   void setValue(const V<B>&);
};

template <class X>
Word Binary_node<X>::toWord()const
{
   return "("&left.toWord()&op&right.toWord()&")";
}

template <class X>
bool Binary_node<X>::eval()const
{
   if (op=="et") return left.eval()&&right.eval();
   else if (op=="vel") return left.eval()||right.eval();
   else {
      cpmerror("Binary_node<X>::eval() un-valid node type");
      return false;
```

```
   }
}

template <class X>
void Binary_node<X>::setValue(const V<B>& vb)
{
    left.setValue(vb);
   right.setValue(vb);
}

//////////////////// class PropLogic ////////////////////////////

typedef PropLog<Prop> PropLogic;

} // namespace

#endif
```

112 cpmlogic1.cpp

```
//? cpmlogic1.cpp
//? Status of work 2008-10-25.
//?
#include <cpmlogic1.h>

using namespace CpmMathFound;

Z Prop::count_=1;

Z Prop::com(const Type& p)const
{
   if (i_<p.i_) return 1;
   else if (i_>p.i_) return -1;
   else return 0;
}

bool Prop::prnOn(ostream& str)const
   { return b_.prnOn(str)&&i_.prnOn(str); }

bool Prop::scanFrom(istream& str)
   { return b_.scanFrom(str)&&i_.scanFrom(str);}

void V_bool::next()
{
   Z mesLim=4;
   if (cpmverbose>=mesLim) cpmmessage("V_bool::next() started");
   Z act=dim()-1;
// notice that rep is an instance of Vl<bool>
Label:
   if (vl_[act]==false){
      vl_[act]=true;
      return;
   }
   else{
      vl_[act]=false;
      act--;
      if (act>=0) goto Label; else return;
   }
}

bool V_bool::end()const
{
   Z n=vl_.dim();
   for (Z i=0;i<n;i++){
      if (vl_[i]==false) return false;
   }
   // then no rep[i] was found to be false
```

```
   return true;
}
```

113 cpmm.h

```
//? cpmm.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_M_H_
#define CPM_M_H_
/*

   Description: Defines a template class M<X,Y> of mappings
      X-->Y comparable to std::map<X,Y>
*/

#include <cpms.h>

//////////////////////// class M<X,Y> ////////////////////////////////////

namespace CpmArrays{

   using namespace CpmStd;
   using CpmFunctions::F;
   using CpmArrays::S;

namespace aux{ // a technical construct

template <class X, class Y>
class  PairO1{  // Pair with order according to first element only.
   // Especially all pairs differing in only the second component
   // are equal by ==. This allows an elegant access to the x-values
   // of (x,y) pairs held in M<X,Y>::rep_. Since this is a 'heavy-duty
   // technical class' it is no longer implemented as derived from
   // X2<X,Y>.
   X x_;
   Y y_;
   typedef PairO1<X,Y> Type;
public:
   Word nameOf()const
   {
      return Word("PairO1< "&
          CpmRoot::Name<X>()(X())&
          ","&
          CpmRoot::Name<Y>()(Y())&
          " >");
   }
   CPM_ORDER
   CPM_IO
   PairO1(X const& x, Y const& y):x_(x),y_(y){}
   explicit PairO1(X const& x):x_(x),y_(){}
   PairO1():x_(),y_(){}
```

```
   X& c1(void){ return x_;}
   Y& c2(void){ return y_;}
   X const& c1(void)const{ return x_;}
   Y const& c2(void)const{ return y_;}
};

template <class X, class Y>
Z PairO1<X,Y>::com(PairO1<X,Y> const& s)const
{
   if (x_<s.x_) return 1;
   else if (x_>s.x_) return -1;
   else return 0;
}

template <class X, class Y>
bool PairO1<X,Y>::prnOn(ostream& str)const
{
   cpmp(x_);
   cpmp(y_);
   return true;
}

template <class X, class Y>
bool PairO1<X,Y>::scanFrom(istream& str)
{
   cpms(x_);
   cpms(y_);
   return true;
}

} // aux

template <class X, class Y>
   class M{ // map, associative array, dictionary, hash
   // it is assumed that X<X and X>X is defined
   Y y0_;
      // default Y, allows to define
      // const Y const& operator[](const X const& x)const;
      // which STL is omitting
   S< aux::PairO1<X,Y> > rep_;
   Z locate(X const& x)const{ return rep_.locate(aux::PairO1<X,Y>(x));}

public:
   typedef M<X,Y> Type;
//   virtual M<X,Y>* clone()const{ return new M(*this);}
   CPM_IO
   CPM_ORDER
   M(void):y0_(),rep_(){}
      // constructor for the void set of X,Y pairs.
      // Non-trivial instances can be obtained by applying
```

```
   // mutating methods, such as function set(...) on the
   // default intance

M(S<X> const& sx, F<X,Y> const& f);
   // Setting the values by an algorithm.

M(S<X> const& sx, Y const& y);
   // Setting a single value y for all elements of sx.

void merge(V<Type> const& mapList, bool reverse=false);
   // combining *this and a list of maps into a single one.
   // This constructs a map that is formed out of all
   // (x,y)-pairs of *this and the components of mapList.
   // If for a x, there are more pairs (x,y), (x,y'), ...
   // The pair coming from mapList[i] with highest i
   // will actually be taken (if we would take more than one
   // we would not get a map). Thus we work through the
   // mapList in ascending order of their indexes and use it
   // in an overwriting mode. If the second argument differs
   // from the default value, we use reversed order so that
   // mapList[i] has highest priority for smallest i.

Y operator()(X const& x)const;
   // in analogy to F<X,Y>; returns the 'value of *this at x'

S<Y> operator()(S<X> const& sx)const;
   // The common extension of functions from single argument values
   // to sets of argument values

Y& operator[](X const& x);
   // setting values in array-style (which also is the STL-style)
   // for M<X,Y> f, X x, Y y the statement
   // f[x]=y has the same effect as f.set(x,y)

Y const& operator[](X const& x)const;
   // getting to the values as references.
   // Let Z g(const& Y), M<X,Y> f, X x, Y y . Then
   // g(f[x]) does the same as
   // Y y1; f.get(x,y1); g(y1);
   // So, operator[] provides slightly more convenient grammar than
   // get(...) in this case.
   // This is the most efficient non-mutating access function

void set(X const& x, Y const& y){(*this)[x]=y;}
   //  M<X,Y> f, X x, Y y ==>
   // ( f.set(x,y) ==> (y==f(x) evaluates to true)). Interesting:
   // it is possible to express this mixture of logic notation
   // and C++ much clearer in pure C++:
   // template <class X, class Y>
   // bool setTest(X x, Y y, M<X,Y> f){ f.set(x,y); return y==f(x);}
```

```
      // returns true for all arguments.

   bool get(X const& x, Y& y)const;
      // template <class X, class Y>
      // bool getTest(X x, M<X,Y> f){ Y y; f.get(x,y); return y==f(x);}
      // returns true for all arguments.

// cardinality access: number of (x,y) pairs stored in *this
   Z card()const{ return rep_.dim();}
   Z dim()const{ return rep_.dim();} // for uniformity with other array
   // types
   Z size()const{ return rep_.dim();} // for uniformity with STL code
   bool isVoid(void)const{ return rep_.isVoid();}
   Z b()const{ return rep_.b();}
      // begin
   Z e()const{ return rep_.e();}
      // end

// accessing arguments, values, and (x,y)-pairs by an index ranging from
// 1 to card()
   X x(Z i)const{ return rep_(i).c1();}
   Y y(Z i)const{ return rep_(i).c2();}
   X2<X,Y> xy(Z i)const{ return X2<X,Y>(x(i),y(i));}
      // template <class X, class Y>
      // bool xyTest(Z i, M<X,Y> f){ return y(i)==f(x(i));}
      // returns true for all arguments.
   bool defined(X const& x)const{ return locate(x)!=0;}
      // f.defined(x) iff before a statement f[x]=... or f.set(x,...)
      // happened.
   void clear_(X const& x){ Z i=locate(x); if (i>0) rep_.eliminate(i);}
      // after f.clear_(x), we have f.defined(x)==false

   S<X> dom()const;
      // domain of the function X-->Y associated with object *this
      // returns {x \in X | defined(x)==true}

   S<Y> ran()const;
      // range of the function X-->Y associated with object *this
      // returns {y \in Y | y==operator[](x) for some x \in X}

   M<X,Y> select(F<X,bool> const& sel)const;
      // returns the M<X,Y> objects consisting of all those
      // (x,y)-pairs of *this for which sel(x)==true

   template <class T>
   M<X,T> operator()(F<Y,T> const& g)const;

   template <class T>
   M<X,T> operator&(M<Y,T> const& m)const;
```

```
   template< class T>
   M<T,Y> circ(const M<T,X>& m)const{ return m&(*this);}

   Word toWord()const;

   Word nameOf()const;
};

//////////////////////////// Implementation ////////////////////////////////

template <class X, class Y>
template<class T>
M<X,T> M<X,Y>::operator()(F<Y,T> const& g)const
{
   M<X,T> res();
   for (Z i=b();i<=e();++i){
      X2<X,Y> xyi=xy(i);
      res.set(xyi.c1(),g(xyi.c2()));
   }
   return res;
}

template <class X, class Y>
template <class T>
M<X,T> M<X,Y>::operator&(M<Y,T> const& m)const
{
   M<X,T> res;
   for (Z i=b();i<=e();++i){
      X2<X,Y> xyi=xy(i);
      bool bi=m.defined(xyi.c2());
      if (bi) res.set(xyi.c1(),m[xyi.c2()]);
   }
   return res;
}

template <class X, class Y>
Word M<X,Y>::nameOf()const
{
   Word wi="M<";
   Word wx=CpmRoot::Name<X>()(X());
   Word wy=CpmRoot::Name<Y>()(Y());
   return wi&wx&","&wy&">";
}

template <class X, class Y>
Word M<X,Y>::toWord()const
{
   Word res="{ ";
   for (Z i=b();i<=e();++i){
      X2<X,Y> xyi=xy(i);
```

```
      Word wxi=CpmRoot::toWord<X>(xyi.c1());
      Word wyi=CpmRoot::toWord<Y>(xyi.c2());
      res&=" ( "&wxi&" , "&wyi&" ) ";
      if (i!=e()) res&=" , ";
   }
   res&=" }";
   return res;
}

template <class X, class Y>
M<X,Y>::M(S<X> const& sx, F<X,Y> const& f):y0_(),rep_()
{
   for (Z i=sx.b();i<=sx.e();++i){
      X xi=sx[i];
      set(xi,f(xi));
   }
}

template <class X, class Y>
M<X,Y>::M(S<X> const& sx, Y const& y):y0_(),rep_()
{
   for (Z i=sx.b();i<=sx.e();++i){
      X xi=sx[i];
      set(xi,y);
   }
}

template <class X, class Y>
Y M<X,Y>::operator()(X const& x)const
{
   Z i=locate(x);
   if (i==0) return Y();
   else return rep_[i].c2();
}

template <class X, class Y>
Y& M<X,Y>::operator[](X const& x)
{
   aux::PairO1<X,Y> fx(x);
   Z i=rep_.locate(fx);
   if (i==0){
      rep_.add(fx);
   }
   i=rep_.locate(fx);
   if (i<=0) cpmerror("Y& M<X,Y>::operator[](X const& x): i<=0");
   return rep_.ref(i).c2();
      // since the ordering of rep_ only relies on rep_.ref(i).c1()
      // there is no danger in exposing rep_.ref(i).c2() to
      // changes from outside
}
```

```
template <class X, class Y>
Y const& M<X,Y>::operator[](X const& x)const
{
   aux::PairO1<X,Y> fx(x);
   Z i=rep_.locate(fx);
   if (i<=0){
      return y0_;
   }
   else{
      return rep_[i].c2();
   }
}

template <class X, class Y>
bool M<X,Y>::get(X const& x, Y& y)const
{
   Z i=locate(x);
   if (i==0){
      return false;
   }
   else{
      y=rep_[i].c2();
      return true;
   }
}

template <class X, class Y>
S<Y> M<X,Y>::operator()(S<X> const& sx)const
{
   Z n=sx.card();
   V<Y> val(n);
   for (Z i=1;i<=n;i++){
      val[i]=y((sx[i]));
   }
   return S<Y>(val);
}

template <class X, class Y>
void M<X,Y>::merge(V<Type> const& mapList, bool reversed)
{
   Z ni,i,j,d=mapList.dim();
   if (!reversed){
      for (i=1;i<=d;i++){
         ni=mapList[i].card();
         for (j=1;j<=ni;j++){
            X x=mapList[i].x(j);
            Y y=mapList[i].y(j);
            set(x,y);
         }
```

```
      }
   }
   else{
      for (i=d;i>=1;i--){
         ni=mapList[i].card();
         for (j=1;j<=ni;j++){
            X x=mapList[i].x(j);
            Y y=mapList[i].y(j);
            set(x,y);
         }
      }
   }
}

template <class X, class Y>
S<X> M<X,Y>::dom()const
{
   S<X> res;
   for (Z i=1;i<=dim();++i) res.add(x(i));
   return res;
}

template <class X, class Y>
S<Y> M<X,Y>::ran()const
{
   S<Y> res;
   for (Z i=1;i<=dim();++i) res.add(y(i));
   return res;
}

template <class X, class Y>
M<X,Y>  M<X,Y>::select(F<X,bool> const& sel)const
{
   M<X,Y> res;
   for (Z i=1;i<=dim();++i){
      X2<X,Y> xyi=xy(i);
      X xi=xyi.c1();
      if (sel(xi)) res[xi]=xyi.c2();
   }
   return res;
}

template <class X, class Y>
bool M<X,Y>::prnOn(ostream& str)const
{
   Z mL=3;
   static Word loc("Mio<>::prnOn(ostream&)");
   CPM_MA
   Z n=card();
   cpmwat;
```

```
   cpmp(n);
   for (Z i=1;i<=n;i++){
      if (CpmRoot::wrtTit){
         Word wi("// i=");
         wi&=cpm(i);
         bool bi=wi.prnOn(str);
         cpmassert(bi==true,loc);
      }
      X2<X,Y> val(x(i),y(i));
      if (!val.prnOn(str)){
         CPM_MZ
         return false;
      }
   }
   cpmwet;
   CPM_MZ
   return true;
}

template <class X, class Y>
bool M<X,Y>::scanFrom(istream& str)
{
   Z mL=3;
   Z mL2=5;
   static Word loc("M<>::scanFrom(istream&)");
   CPM_MA
   Z n;
   if (!CpmRoot::scanFrom<Z>(n,str)){
      cpmwarning(loc&": can’t read dimension");
      CPM_MZ
      return false;
   }
   else if (n<0){
      cpmwarning(loc&": negative dimension: n="&cpm(n));
      CPM_MZ
      return false;
   }
   else if (n>dimMax)
      cpmwarning(loc&": dimension read as "&cpm(n)&
      ": probably too large");
   else if (n==0){
      cpmwarning(loc&" dimension read as 0");
      if (cpmverbose>mL2){ // copying the stream beyond the dubious point
         // can be useful
         char c;
         while (str.get(c)) cpmdata.put(c);
      }
   }
   else{
      cpmmessage(mL,loc&": dimension read as "&cpm(n));
```

```
   }
   M<X,Y> res;
   X2<X,Y> val;
   for (Z i=1;i<=n;i++){
      if (!val.scanFrom(str)){
         cpmwarning(loc&": reading failed for i="&cpm(i)&" of "&cpm(n));
         CPM_MZ
         return false;
      }
      res.set(val.c1(),val.c2());
   }
   *this=res;
   CPM_MZ
   return true;
}

template <class X, class Y >
Z M<X,Y>::com(M<X,Y> const& s)const
{
   Z d1=dim(), d2=s.dim();
   if (d1<d2) return 1;
   else if (d1>d2) return -1;
   else{ // then d1==d2
      for (Z i=1;i<=d1;i++){
         if (xy(i)<s.xy(i))   return 1;
         if (xy(i)>s.xy(i))   return -1;
      }
      return 0;
   }
}

} // CpmArrays

#endif
```

114 cpmmacros.h

```
//? cpmmacros.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_MACROS_H_
#define CPM_MACROS_H_
/*

   Description: macros for uniform implementation of
      functions. Too often I wrote the starting cpmmessage
      and the closing one in function blocks. The abbreviation by macros
      is not impressive in this case. For the R-macro the savings are
      more convincing.

*/

//////////////////////// CPM_M ////////////////////////////////////////

// M stands for messaging
//              -
// A for beginning, Z for end (from the
// role of A and Z in the alphabet)

// needs
// Z mL=...;
// Word loc(".....");
// see the example following CPM_R

#define CPM_MA\
   cpmmessage(mL,loc&" started");

#define CPM_MZ\
   cpmmessage(mL,loc&" done");

//////////////////////// CPM_R ////////////////////////////////////////

// R stands for report
//              -
// reports the computation time for the code between CPM_RA and
// CPM_RZ. If loadMes (load measure) is assigned a positive value
// also the computation time divided by that quantity is returned
// under the name of 'normalized computation time'

// needs
// Word loc(".....");

#define CPM_RA\
   R tcStart=cpmtime();\
```

```
   R loadMes=0;

#define CPM_RZ\
   R tcFinal=cpmtime();\
   R tcTotal=tcFinal-tcStart;\
   std::ostringstream ost;\
   ost<<std::endl<<"Performance data for "<<loc.toStr()<<std::endl;\
   ost<<"computing time ="<<tcTotal;\
   if (loadMes>0){\
      R tcNorm=tcTotal/loadMes;\
      ost<<std::endl<<"normalized computing time="<<tcNorm;\
   }\
   cpmmessage(1,Word(ost.str()),-1);
      // not writing to the status bar due to '-1'

//////////////////// usage of CPM_M and CPM_R ////////////////////
/*
   a typical usage is

  void aFunction()
  {
      Z mL=2;
      Word loc("aFunction()");
      CPM_MA
      CPM_RA
      ...
      loadMes=...;
      ...
      CPM_RZ
      CPM_MZ
   }

   for a function that notifies start, end, computation time,
   and an efficiency measure on cpmcerr.txt. Notice that A and Z
   resemble opening and closing brackets.
*/
#endif
```

115 cpmmathfound.h

```
//? cpmmathfound.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_MATHFOUND_H_
#define CPM_MATHFOUND_H_
/*
   Description: Some basic classes useful for foundation of
      discrete math.
      NatNum is the class of natural numbers. Implementation
      is based solely on the typographics of character strings.
*/

#include <cpminterfaces.h>
#include <vector>

namespace CpmMathFound{ // foundation of mathematics

const char conChr='l'; // counting character
   // natural numbers are represented by strings made out of
   // this character only (according to what sometimes is called
   // the 'stone age representation' of numbers).

using std::vector;

using namespace CpmStd;

///////////////// class NatNum ////////////////////////////////////////////

class NatNum{
// natural numbers {1,2,3, ...} in 'stone age representation'

   string rep;
   NatNum(const string& x, bool b):rep(x){}
      // private since it assumes that x is a string of conChr's and
      // of no other characters
public:
   typedef NatNum Type;
   CPM_IO
   CPM_ORDER
   CPM_SUM_C
      // summation is based on concatenation of strings
   CPM_PRODUCT_C
   static NatNum unit;
      // number 1
   bool isUnit()const{ return rep.size()==1;}
      // returns true iff *this==unit
   explicit NatNum(const string& x):rep(x.size(),conChr){}
```

```
      // construction from the basic typographical representation
      // of a number as a string of n conChr's. Actually arbitrary
      // characteres are allowed, but will be translated to conChr
      // for storage in string rep.
   NatNum():rep(1,conChr){}
      // constructs number one

   explicit NatNum(int j):rep((j>0 ? j : 1),conChr){}
      // Creation from system provided numbers.
      // if j is not a natural number we create unit
   int toInt()const{ return (int)rep.size();}
      // conversion to system provided numbers

   NatNum next()const{
      string repNew=rep;
      return NatNum(repNew.append(1,conChr),true);
   }
      // next number in natural order; obtained by appending
      // the character conChr
   NatNum prev()const;
      // previous number in natural order; if *this==unit, the
      // result is unit too in order to have the function defined
      // everywhere and with homogeneous type (as the declaration says).
};

} // namespace

#endif
```

116 cpmmathfound.cpp

```
//? cpmmathfound.cpp
//? Status of work 2008-10-25.
//?
#include <cpmmathfound.h>

using CpmMathFound::NatNum;
using CpmRoot::Z;
using namespace std;

NatNum unit();

Z NatNum::com(const NatNum& n)const
{
   if (rep.size()<n.rep.size())   return 1;
   if (rep.size()>n.rep.size())   return -1;
   return 0;
}

bool NatNum::prnOn(ostream& out)const
{
   CpmRoot::writeTitle("NatNum",out);
   CpmRoot::write(rep,out);
   return (out!=0);
}

bool NatNum::scanFrom(istream& in)
{
   CpmRoot::read(rep,in);
   return (in!=0);
}

NatNum NatNum::prev()const
{
   if (isUnit()) return *this;
   else{
      string repNew=rep;
      repNew.erase(rep.size()-2,1);
      return NatNum(repNew,true);
   }
}

NatNum NatNum::operator +(const NatNum& n)const
// concatenation '+' in class string is the basic 'arithmetic
// operation.
{
   return NatNum(rep+n.rep,true);
}
```

```
NatNum NatNum::operator *(const NatNum& n)const
// no direct usage of string features; only
// NatNum::isUnit(), NatNum::prev(), NatNum::+=()
{
   NatNum res=n;
   NatNum counter=*this;
   while (!counter.isUnit()){
      counter=counter.prev();
      res+=n;
   }
   return res;
}
```

117 cpmmaxw1d.h

```
//? cpmmaxw1d.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_MAXW1D_H_
#define CPM_MAXW1D_H_
/*
   Description: Development and test of a class
      that represent a Maxwell field in a 1-dimensional cavity.
      The system under consideration is that introduced in
      Hans De Raedt et al. Computer Physics Communications 156 (2003)
         43-61 in section Implementation
*/

#include <valarray>

#include <cpmgraph.h>
#include <cpmrfunc.h>
#include <cpmrvector.h>

using CpmRoot::Z; // int by typedef
using CpmRoot::R; // double by typedef
using CpmRoot::Word; // character strings
using CpmRootX::R1;
using CpmRootX::B;
using CpmLinAlg::R2; // pairs of R
using CpmGraphics::Graph; // graphics
using CpmFunctions::R_Func; // functions R --> R
using CpmFunctions::F; // functions R --> R
using CpmArrays::V; // array template
using CpmArrays::X2; // Cartesian product of two factors
using CpmArrays::R_Vector;// R-valued array
using CpmArrays::Outside;//enumeration controling
   // out of range continuation of arrays
using CpmArrays::DEFAULT;
using CpmArrays::CONSTANT;
using CpmGeo::Iv; // intervals

using namespace std;

namespace CpmWaves{

class TEM { // electromagnetic field in TEM mode
// named TEM according to
// Jackson: Classical Electrodynamics, Wiley 1963, Sect. 8.2
// Actually there are two TEM modes: one with E and H perpendicular,
// and one with E and H parallel. The first one is treated in De Raedt et
// al., and is the one to be treated here.
```

```
// Names of data members and of non-constant member functions end in '_'.

   Z n_;
      // any positive integer
   Z d_;
      // the odd number 2*n_+1
   R L_;
      // distance from wall to wall
   R s_;
      // spacing between neigbors
      // s_=L_/(2*n_)=L_/(d_-1)
   R sInv_;
      // 1/*s_
   R s2Inv_;
      // 1./(s_*2)
// all fields depend only on x (0<=x<=L_), E is E_z, H is H_y
   valarray<R> x_;
      // equi-distant positions: x_[0]=0, x_[d_-1]=L_
   valarray<R> X_;
      // mixed magnetic and electric components according to the Yee
      // scheme.
      // See the 2003 Computer Physics Communications article by Hans De
      // Raedt et al. Fig.2. I found it convenient to include in X_ the
      // vanishing E-component at the boundaries.
   valarray<R> dX_;
      // time derivative of the field components
   void ini_(){ s_=L_/(2*n_); sInv_=1./s_; s2Inv_=0.5*sInv_; }
      // setting the quantities depending on L_ and n_
   valarray<R> der(valarray<R> const& v)const
      // Time derivative of the field values v in Yee order.
      // Expressed by Maxwell's equations as a spatial derivative.
      // Vanishes at the two boundary points.
   {
      valarray<R> res(0.,d_);
      for (Z i=1;i<d_-1;++i) res[i]=(v[i+1]-v[i-1])*s2Inv_;
      return res;
   }

   valarray<R> acc(valarray<R> const& v)const{ return der(der(v));}
      // acceleration as the second time derivative.

public:

   TEM(){}

   TEM(Z n, R L, R_Func const& fE, R_Func const& fH);
      // fE, and fH are defined on the interval [0,1].
      // n is the number of E-values and of H-values
      // The number of X_-values is 2*n+1. (n H,E-pairs and a H value at
      // the end.
```

```
      // This is a very clever way to suggest a uniform definition of the
      // time derivative of the field values in a manner that the
      // vanishing of the electric field on the boundary is guaranteed.

   TEM(Z n, R L, Iv const& iv);
      // E-field has stochastic values uniformly distributed in iv and H
      // is zero.

   void step_(R dt)
      // time step according to the direct midpoint method
   {
      R tau=0.5*dt;
      X_+=dX_*tau;
      dX_+=acc(X_)*dt;
      X_+=dX_*tau;
   }

   R energy()const{ return (X_*X_).sum();}
      // Total energy (some re-interpretation is involved since verbatim
      // the field energy of homogeneous fields in an infinite
      // volume is infinite. Turns out to be constant to good
      // approximation.

   R dissipation()const{ return (X_*dX_).sum()*2;}
      // Time derivative of total energy. Is small since the
      // total energy is close to being constant

// Connection to arrays of type R_Vector and V<R2> since all my
// graphical representation functions work with them. Their
// index range starts with 1 instead of the C-native 0.

   R_Vector getXr()const
   {
      R_Vector res(d_);
      for (Z i=0;i<d_;++i) res[i+1]=X_[i];
      return res;
   }

   R_Vector getxr()const
   {
      R_Vector res(d_);
      for (Z i=0;i<d_;++i) res[i+1]=x_[i];
      return res;
   }

   V<R2> getX()const
   {
      V<R2> res(d_);
      for (Z i=0;i<d_;++i) res[i+1]=R2(x_[i],X_[i]);
      return res;
```

```
   }

   V< V<R2> > getEH()const
      // extracting E and H-fields in a way which is suitable for
      // graphical representation
   {
      V<R2> vE(n_+1); // boundary values included
      V<R2> vH(n_); // d_=n_+(n_+1)
      Z iE=1,iH=1;
      bool putE=true;
      for (Z i=0;i<d_;++i){
         if (putE){
            vE[iE++]=R2(x_[i],X_[i]);
            putE=!putE;
         }
         else{
            vH[iH++]=R2(x_[i],X_[i]);
            putE=!putE;
         }
      }
      return V< V<R2> >("",vE,vH);
   }

   void showAsCurves(Graph& g)const;
      // auto-scaled polygon representation of the fields.
      // E as a red curve, B as a green one. Based on
      // the previous function.

   R getL()const{ return L_;}
};

namespace{

    class RV2{ // E and H field combined, tool class
      R_Vector E_;
      R_Vector H_;
public:
      RV2(R_Vector const& E, R_Vector const& H):E_(E),H_(H)
      {
         if (E_.dim()!=H_.dim())
         cpmerror("RV2(R_Vector,R_Vector): dimension mismatch");
      }
      RV2(){}
      explicit RV2(Z n):E_(n),H_(n){}
      RV2& operator+=(RV2 const& a){ E_+=a.E_; H_+=a.H_; return *this;}
      RV2& operator*=(R r){ E_*=r; H_*=r;return *this;}
      RV2 operator*(R r)const{ return RV2(E_*r,H_*r);}

      RV2 dot(R fac)const{ return  RV2(derH(H_)*fac, derE(E_)*fac);}
```

```
      RV2 acc(R fac)const
      { return dot(fac).dot(fac);}

      RV2 acc(R fac, R_Vector const& J, R_Vector const& dJ)const
      { return RV2(derE(derE(E_))-dJ, derH(derH(H_))-derE(J));}

      R_Vector c1()const{ return E_;}
      R_Vector c2()const{ return H_;}
      void setE_(Z i, R val){E_[i]=val;}
      void setH_(Z i, R val){H_[i]=val;}

      static R_Vector derE(R_Vector const& v)
         // spatial derivative (up to a factor) of a field v which
         // vanishes at the ends (as the E-field does)
      {
         Z n=v.dim();
         R_Vector res(n);
         Outside mode=DEFAULT;
         for (Z i=1;i<=n;++i) res[i]=(v.read(i+1,mode)-v.read(i-1,mode));
       //  for (Z i=1;i<=n;++i) res[i]=(v.read(i+1,1)-v.read(i-1,1));
         return res;
      }

      static R_Vector derH(R_Vector const& v)
        // spatial derivative (up to a factor) of a field v which needs
        // not vanish at the ends (as the E-field does)
      {
         Z n=v.dim();
         R_Vector res(n);
         Outside mode=CONSTANT;
         for (Z i=1;i<=n;++i) res[i]=(v.read(i+1,mode)-v.read(i-1,mode));
       //  for (Z i=1;i<=n;++i) res[i]=(v.read(i+1,1)-v.read(i-1,1));
         return res;
      }

   };
} // anonymous

class TEM2 { // TEM in C+- style
// class TEM in cpm style
// discretization does not make use of Yee scheme
// works fine

   Z n_;
      // any positive integer
   R L_;
      // distance from wall to wall
   R s_;
      // spacing between neigbors
   R sInv_,hsInv_;
```

```
      // 1/*s_

// all fields depend only on x (0<=x<=L_), E_ is E_z, H_ is H_y
   R_Vector x_;
      // equi-distant positions: x_[0]=0, x_[d_-1]=L_

   RV2 X_;

   RV2 dX_;

   R t_;

   R tci_, tcf_;

   B withCurrent_;
      // initialized as false, OK
   B reInitialized_;
      //

   void ini_(){
      t_=0; s_=L_/(n_-1); sInv_=1./s_;
      hsInv_=sInv_*0.5;
   }
      // setting the quantities depending on L_ and n_

   F<R,R_Vector> fJ_;
public:

   TEM2(){}

   TEM2(Z n, R L, R_Func const& fE, R_Func const& fH);
      // fE, and fH are defined on the interval [0,1].
      // n is the number of E-values and of H-values
      // The number of X_-values is 2*n+1. (n H,E-pairs and a H value at
      // the end.
      // this is a very clever way to suggest a uniform definition of the
      // time  derivative of the field values in a manner that the
      // vanishing of the electric field on the boundary is guaranteed

   TEM2(Z n, R L, Iv const& iv);
      // E-field has stochastic values uniformly distributed in iv and H
      // is zero.

   TEM2(Z n, R L, R xJ, Iv tJ, R omegaJ, Z nSmoothJ=8);
      // n,L as above, xJ: position of the source, tJ: time span
      // in which the AC-current J is in action
      // omegaJ: circular frequency of the AC current

   void step_(R dt)
      // time step according to the direct midpoint method
```

```
   {
      R tau=0.5*dt;
      t_+=tau;
      X_+=dX_*tau;
      if (withCurrent_){
         if (t_<=tci_ || t_>=tcf_) dX_+=X_.acc(hsInv_)*dt;
         else{
            R_Vector J=fJ_(t_);
            R_Vector dJ=(fJ_(t_+tau)-fJ_(t_-tau))/dt;
            dX_+=X_.acc(hsInv_,J,dJ)*dt;
         }
      }
      else{
         dX_+=X_.acc(hsInv_)*dt;
      }
      X_+=dX_*tau;
      t_+=tau;
   }

   R energy()const{ return (X_.c1()|X_.c1())+(X_.c2()|X_.c2());}
      // Total energy (some re-interpretation is involved since verbatim
      // the field energy of homogeneous fields in an infinite
      // volume is infinite. Turns out to be constant to good
      // approximation.

   R dissipation()const{ return ((X_.c1()|dX_.c1())+(X_.c2()|dX_.c2()))*2\
   ;}
      // Time derivative of total energy. Is small since the
      // total energy is close to being constant

// Connection to arrays of type R_Vector and V<R2> since all my
// graphical representation functions work with them. Their
// index range starts with 1 instead of the C-native 0.

   R_Vector getE()const
   {
      return X_.c1();
   }

   R_Vector getH()const
   {
      return X_.c2();
   }

   R_Vector getxr()const{ return x_;}


   V< V<R2> > getEH()const
      // extracting E and H-fields in a way which is suitable for
      // graphical representation
```

```
   {
      V<R2> vE(n_);
      V<R2> vH(n_);
      R_Vector E=getE();
      R_Vector H=getH();
      for (Z i=1;i<=n_;++i){
         R xi=x_[i];
         R Ei=E[i];
         R Hi=H[i];
         vE[i]=R2(xi,Ei);
         vH[i]=R2(xi,Hi);
      }
      return V< V<R2> >("",vE,vH);
   }

   void showAsCurves(Graph& g)const;
      // auto-scaled polygon representation of the fields.
      // E as a red curve, B as a green one. Based on
      // the previous function.

   R_Vector getXr()const
   {
      return getE()&getH();
   }

   R getL()const{ return L_;}
};

} // namespace

#endif
```

118 cpmmaxw1d.cpp

```
//? cpmmaxw1d.cpp
//? Status of work 2008-10-25.
//?
#include <cpmmaxw1d.h>
#include <cpmvectormore.h>
using namespace CpmFunctions;

CpmWaves::TEM::TEM(Z n, R L, R_Func const& fE, R_Func const& fH):
   n_(n),d_(2*n_+1),L_(L),X_(d_),x_(d_)
{
   ini_();
   R dxi=1./(d_-1);
   bool putE=true;
   R x=0,xi=0;
   for (Z i=0;i<d_;++i){
      x_[i]=x;
      x+=s_;
      if (putE){
         X_[i]=fE(xi);
         putE=false;
      }
      else{
         X_[i]=fH(xi);
         putE=true;
      }
      xi+=dxi;
   }
   dX_=der(X_);
}

CpmWaves::TEM::TEM(Z n, R L, Iv const& iv):
   n_(n),d_(2*n_+1),L_(L),X_(0.,d_),x_(d_)
{
   ini_();
   bool putE=true;
   R x=0;
   for (Z i=0;i<d_;++i){
      x_[i]=x;
      x+=s_;
      if (putE){
         X_[i]=((i==0 || i==d_-1) ? 0. : iv.ran());
         putE=false;
      }
      else putE=true;
   }
   dX_=der(X_);
}
```

```
void CpmWaves::TEM::showAsCurves(Graph& g)const
{
   using namespace CpmImaging;
   V<Color> col(2);
   col[1]=RED;
   col[2]=GREEN;
   g.addText("TEM-1D: E-field red, H-field green");
   g.mark(getEH(),col);
   g.display();
}

//////////////////////////////////////////////////////////
CpmWaves::TEM2::TEM2(Z n, R L, R_Func const& fE, R_Func const& fH):
   n_(n),L_(L),X_(n_),x_(n_)
{
   ini_();
   R x=0;
   R xi=0;
   R dxi=1./(n_-1);
   for (Z i=1;i<=n_;++i){
      x_[i]=x;
      x+=s_;
      X_.setE_(i,fE(xi));
      X_.setH_(i,fH(xi));
      xi+=dxi;
   }
   dX_=X_.dot(hsInv_);
}

CpmWaves::TEM2::TEM2(Z n, R L, Iv const& iv):
   n_(n),L_(L),X_(n_),x_(n_)
{
   ini_();
   R x=0;
   for (Z i=1;i<=n_;++i){
      x_[i]=x;
      x+=s_;
      R val=((i==1 || i==n_) ? 0. : iv.ran());
      X_.setE_(i,val);
   }
   dX_=X_.dot(hsInv_);
}


R_Vector fAux(R const& t, Iv const& iv, R_Vector const& v, R const& omega\
)
{
   R ft=sin(omega*(t-iv[1]));
   return v*ft;
```

```
}

CpmWaves::TEM2::TEM2(Z n, R L, R xJ, Iv tJ, R omegaJ,Z nSmoothJ):
n_(n),L_(L),X_(n_),x_(n_),withCurrent_(true),tci_(tJ[1]),tcf_(tJ[2])
{
   ini_();
   R x=0;
   R vE=1e-10;
   R vH=-0.5*vE;
   R xi=0;
   R dxi=1./(n_-1);
   for (Z i=1;i<=n_;++i){
      x_[i]=x;
      x+=s_;
      R y=xi*(1-xi);
      X_.setE_(i,y*vE);
      X_.setH_(i,y*vH);
      xi+=dxi;
   }
   Z iJ=cpmround(1+xJ*sInv_);
   R_Vector vJ(n_);
   vJ[iJ]=1;
   vJ=CpmArrays::smoothen(vJ,nSmoothJ);
   fJ_=F3<R,Iv,R_Vector,R,R_Vector>(tJ,vJ,omegaJ)(fAux);
   dX_=X_.dot(hsInv_);
}

void CpmWaves::TEM2::showAsCurves(Graph& g)const
{
   using namespace CpmImaging;
   V<Color> col(2);
   col[1]=RED;
   col[2]=BLUE;
   g.addText("TEM2-1D: E-field red, H-field blue");
   g.mark(getEH(),col);
   g.display();
}
```

119 cpmmini.h

```
//? cpmmini.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_MINI_H_
#define CPM_MINI_H_

/*
Purpose: Minimum search for a real-valued function of a real vector
   argument in 'modern' form, where the function is
   represented as an instance of F<R_Vector,R> and is a data member of
   the class Mini. The class thus is no longer a template.
*/

#include <cpmrmatrix.h>
#include <cpmfa.h>

//////////////////////// class Mini ////////////////////////////////////

namespace CpmAlgorithms{

   using namespace CpmStd;

   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmRootX::Z1;
   using CpmRootX::B;
   using CpmFunctions::F;
   using CpmFunctions::Fa;
   using CpmArrays::R_Vector;
   using CpmArrays::X2;
   using CpmArrays::R_Matrix;

class Mini{   // finds minimum of a function
   // ff based on Powell's method, modified from Press et al.

   typedef Mini Type;
public:
   virtual Mini* clone()const{ return new Mini(*this);}

   Mini(R fracTol=1e-3);
      // fracTol=ftol is the the fractional tolerance in the function
      // value such that failure to decrease by more than this ammount on
      // one iteration signals doneness.

   explicit Mini(F<R_Vector,R> const& f, R fracTol=1e-3);
      // sets the function to be minimized and the fractional tolerance
      // (see previous function)
```

```
void setAcc(R fracTol);
   // see first function

virtual void setF(const F<R_Vector,R>& f);
   // setting the 'function' ff to be minimized.

R_Vector getPoint(const R_Vector& xStartInput);
   // computing the minimum and returning the resulting x-value.
   // The starting point has to be provided.
   // The vectors x which are taken into account for minimization
   // are those for which dim(x)=dim(xStartInput)
   // An assert condition assures that the program gets interupted
   // if this function is called and data member ff was not yet
   // initialized by either a appropriate constructor call or by
   // calling setF

virtual  R getValue(void)const;
   // getting the value of the minimized function at the minimum
   // position. Returns reasonable values only after getPoint was
   // called before. An assert condition assures that the program
   // gets interupted if this is not the case.

X2<R_Vector,R> operator()(const R_Vector& x)
   // 'modern' interface to the main functionality of the class
{  R_Vector xm=getPoint(x); R y=getValue();
   return X2<R_Vector,R>(xm,y);}

Z getIter(void)const;
   // getting the number of iterations needed. Returns reasonable
   // values only after getMin was called before. An assert condition
   // assures that the program gets interupted if this is not the
   // case.

Z nEval()const{ return nEval_;}

Z getIterBrent(void)const;
   // getting the maximum number of iterations needed in brent().
   // Returns reasonable values only after getMin was called before.
   // An assert condition assures that the program gets interupted if
   // this is not the case.

bool getSuc()const{ return suc_;}

Z1 verbose;
   // for controlling messages in a flexible way without using
   // cpmverbose

static Z iterMaxPowell;
   // 30 seems to be a good value; should be open to changes, thus
```

```
      // public

   static Z iterMaxBrent;
      // 30 seems to be a good value; should be open to changes, thus
      // public

   void show()const;
      // writes all relevant method parameters on file for diagnostic
      // purposes

protected:
   Fa<R_Vector,R> ff_;
       //    function object to be minimized

private:
   B suc_;
      // true if minimum sucessfully found, initialized as
      // false which is OK
   R minimize(R_Vector& x);
   R ftol, yMin;
   R_Vector xStart, xMin;
   Z done;
      // zero if no minimization was yet done

   Z ff_Defined;
      // zero if ff_ not yet explicitely initializes
   Z iter;
   Z iterBrent;
   Z callsBrent;
   R_Vector pPowell;
   R_Vector xiPowell;
   R f1Powell(R x){ nEval_++; return ff_(pPowell+x*xiPowell);}

// auxiliar functions modified from Press et al.
   bool brak(R& ax, R& bx, R& cx, R& fa, R& fb, R& fc);
   bool testBrak(R& ax, R& bx, R& cx, R& fa, R& fb, R& fc);
   R brent(R ax, R bx, R cx, R& xmin);
   bool linMinPowell(R_Vector& p, R_Vector& xi, R& fret);
       // sets pPowell and xiPowell
   bool powell(R_Vector& x, R_Matrix xi, R& fret);
      // core of the method
   Z nEval_;
      // number of evaluations of ff_ during the
      // whole minimization process

// method constants from Press et al.
   static R GOLD;
   static R GLIMIT;
   static R TINY;
   static R EPS;
```

```
   static R TOL;
   static R CGOLD;
   static R ZEPS;
};

class Maxi:public Mini{ // finds maximum of a function
   // which gets entered either via constructor or via
   // setF
   typedef Mini Base;
   typedef Maxi Type;
public:
   virtual Mini* clone()const{ return new Maxi(*this);}
      // this allows instances of Maxi and of Mini to be
      // components of the polymorphic array class Vp<Mini>
   Maxi(R fracTol=1e-3):Base(fracTol){}
   Maxi(const F<R_Vector,R>& f, R fracTol):Base(f,fracTol){ff_*=-1;}
      // f is a function for which the maximum will be searched
      // Internally this is done by searching the minimum of -f,
      // as is clear from the code
   R getValue(void)const{ return -Base::getValue();}
      // getting the value of function f at point of maximum
   void setF(const F<R_Vector,R>& f){ Base::setF(f); ff_*=-1;}
      // setting the function to be maximized.
};

} // namespace

#endif
```

120 cpmmini.cpp

```
//? cpmmini.cpp
//? Status of work 2008-10-25.
//?
#include <cpmmini.h>
#include <cpmperm.h>

using CpmAlgorithms::Mini;
using CpmRoot::R;
using CpmRoot::Z;
using CpmRoot::Word;
using CpmFunctions::F;
using CpmArrays::R_Vector;
using CpmArrays::R_Matrix;
using CpmArrays::Perm;
using CpmRoot::absVal;

// initialization of static members

Z Mini::iterMaxBrent=200;
Z Mini::iterMaxPowell=200; // was 50

R Mini::GOLD=1.618034;
R Mini::GLIMIT=100; // is 100 in Press et al.
R Mini::TINY=1.0e-20;
R Mini::EPS=1.0e-15;  // machine floating point precision was 1e-16
// before
         // 97-2-20
R Mini::TOL=5.*sqrt(EPS);  // factor was 2 before 97-2-20
R Mini::CGOLD=0.3819660;
R Mini::ZEPS=1.0e-10;


namespace{

   // functions helping the implementation

R MAX(R a, R b){return a > b ? a : b;}
R SIGN(R a, R b){return b > 0. ? absVal(a):-absVal(a);}
void SHFT(R& a, R& b, R& c, R& d){a=b;b=c;c=d;}
R SQR(R a){return a*a;}

} // namespace

bool Mini::testBrak(R& ax, R& bx, R& cx, R& fa, R& fb, R& fc)
{
   static const Z countmax=3;
   Z counter=0;
```

```
Start:
   Z good=((ax<=bx && bx<=cx)||(cx<=bx && bx<=ax)) && (fb<=fa && fb<=fc);
   if (good){
      return true;
   }
   else{
      // trying to fix the problem by reordering
     // if (verbose>0){
      if (verbose>0){
         cpmwarning("unexpected reordering needed in Mini testBrak()");
      }
      counter++;
      R_Vector xVal(3);
      xVal[1]=ax;
      xVal[2]=bx;
      xVal[3]=cx;
      Perm p(xVal);
      xVal=xVal*p;
      R_Vector fVal(3);
      fVal[1]=fa;
      fVal[2]=fb;
      fVal[3]=fc;
      fVal=fVal*p;
      if (fVal[2]<=fVal[1] && fVal[2]<=fVal[3]){
         ax=xVal[1];
         bx=xVal[2];
         cx=xVal[3];
         fa=fVal[1];
         fb=fVal[2];
         fc=fVal[3];
         if (counter<=countmax){
            goto Start;   // testing wether were sucessfull
         }
         else{
            goto BadEnd;
         }
      }
      else{
         goto BadEnd;
      }
   }

BadEnd:
   {
      cpmcerr<<endl<<"bad return in testBrak"<<endl<<
      "ax="<<ax<<" bx="<<bx<<" cx="<<cx<<endl<<
      "fa="<<fa<<" fb="<<fb<<" fc="<<fc<<" counter="<<counter<<endl;
      cpmwarning("bad return in testBrak");
      return false;
   }
```

```
}


Mini::Mini(R fracTol):nEval_(0)
{
   done=0;
   iter=0;
   iterBrent=0;
   callsBrent=0;
   ftol=fracTol;
   ff_Defined=0;
}

Mini::Mini(const F<R_Vector,R>& f, R fracTol):ff_(f),ftol(fracTol),
   nEval_(0)
{
   done=0;
   iter=0;
   iterBrent=0;
   callsBrent=0;
   ff_Defined=1;
}

void Mini::setAcc(R fracTol)
{
   ftol=fracTol;
}

void Mini::setF(const F<R_Vector,R>& f)
{
   ff_=f;
   ff_Defined=1;
   nEval_=0;
}

R_Vector Mini::getPoint(const R_Vector& xStartInput)
{
   if (verbose) cpmmessage("Mini::getMin() started");
   Word loc("Mini::getMin()");
   cpmassert(ff_Defined==1,loc);
   xStart=xStartInput;
   xMin=xStart;
   iterBrent=0;
   iter=0;  // not needed due to the unconventional usage of iter in
            // Mini::powell
   yMin=minimize(xMin);
   done=1;
   if (verbose>=4) show();
   if (verbose) cpmmessage("Mini::getMin() done");
   return xMin;
```

```
}

R Mini::getValue(void)const
   { cpmassert(done==1,"Mini::getValue()"); return yMin;}

Z Mini::getIter(void)const
   { cpmassert(done==1,"getIter()"); return iter;}

Z Mini::getIterBrent(void)const
   { cpmassert(done==1,"getIterBrent()"); return iterBrent;}

bool Mini::brak(R& ax, R& bx, R& cx, R& fa, R& fb, R& fc)
/*
See Press et al. mnbrak(), p. 400
Given distinct initial points
ax and bx, this routine searches in the downhill direction
(defined by the function as evaluated at the initial points)
and returns new points ax,bx,cx which bracket a minimum of the
function. Also returned are the function values at the three
points , fa, fb, fc.
*/
{
   R ulim,u,r,q,fu,dum;

   fa=f1Powell(ax);
   fb=f1Powell(bx);
   if (fb > fa) {
      SHFT(dum,ax,bx,dum);
      SHFT(dum,fb,fa,dum);
   }
   cx=bx+GOLD*(bx-ax);
   fc=f1Powell(cx);
   while (fb > fc) {
      if (verbose>2){
         cpmvalue("fb",fb,2);
         cpmvalue("fc",fc,3);
      }
      r=(bx-ax)*(fb-fc);
      q=(bx-cx)*(fb-fa);
      u=(bx)-((bx-cx)*q-(bx-ax)*r)/
         (2.0*SIGN(MAX(absVal(q-r),TINY),q-r));
      ulim=(bx)+GLIMIT*(cx-bx);
      if ((bx-u)*(u-cx) > 0.0) {
         fu=f1Powell(u);
         if (fu < fc) {
            ax=(bx);
            bx=u;
            fa=(fb);
            fb=fu;
            return true;
```

```
         } else if (fu > fb) {
            cx=u;
            fc=fu;
            return true;
         }
         u=cx+GOLD*(cx-bx);
         fu=f1Powell(u);
      } else if ((cx-u)*(u-ulim) > 0.0) {
         fu=f1Powell(u);
         R fu1=fu, cxx=cx+GOLD*(cx-bx);
         if (fu < fc) {
            SHFT(bx,cx,u,cxx);
            SHFT(fb,fc,fu,fu1);
         }
      } else if ((u-ulim)*(ulim-cx) >= 0.0) {
         u=ulim;
         fu=f1Powell(u);
      } else {
         u=(cx)+GOLD*(cx-bx);
         fu=f1Powell(u);
      }
      SHFT(ax,bx,cx,u);
      SHFT(fa,fb,fc,fu);
   }
   return testBrak(ax, bx, cx, fa, fb, fc);
}


R Mini::brent(R ax, R bx, R cx, R& xmin)
/*
Given a bracketing triplet of abscissas ax,bx,cx
(such that bx is between ax and cx, and f(bx) is less than both f(ax) and
f(cx), this routine isolates the minimum to a fractional precision of
about TOL using Brent's method. The abscissa of the minimum is returned
as xmin, and the minimum function value is returned as brent, the
returned function value.
*/
{
   Word loc("Mini::brent(R,R,R)");
   Z iterB;
   R a,b,d=0,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm;
   R term;
   callsBrent++;
   R e=0.0;
   a=((ax < cx) ? ax : cx);
   b=((ax > cx) ? ax : cx);
   x=w=v=bx;
   fw=fv=fx=f1Powell(x);
   for (iterB=1;iterB<=iterMaxBrent;iterB++) {
      if (verbose>2) cpmvalue("iterB",iterB,2);
```

```
xm=0.5*(a+b);
tol1=TOL*absVal(x)+ZEPS;
tol2=2.0*tol1;
term=tol2-0.5*(b-a)-absVal(x-xm);
if (verbose>2){
   if ( absVal(term+1.)<=1.e-4 ){
      cpmcerr<<endl<<"term=-1"<<endl<<"tol1="<<tol1<<" tol2="<<tol2
      <<" b="<<b<<" a="<<a<<" x="<<x<<" xm="<<xm;
   }
}
if (verbose>20){ cpmcerr<<endl<<"Mini:brent(), term="<<term;}
if ( term>=0.) {
   xmin=x;
   iterBrent=CpmRootX::sup(iterBrent,iterB);
   return fx;
}
if (absVal(e) > tol1) {
   r=(x-w)*(fx-fv);
   q=(x-v)*(fx-fw);
   p=(x-v)*q-(x-w)*r;
   q=2.0*(q-r);
   if (q > 0.0) p = -p;
   q=absVal(q);
   etemp=e;
   e=d;
   if (absVal(p) >= absVal(R(0.5)*q*etemp) || p <= q*(a-x) ||
      p>=q*(b-x))
      d=CGOLD*(e=(x >= xm ? a-x : b-x));
   else {
      cpmassert(q!=0.,loc);
      d=p/q;
      u=x+d;
      if (u-a < tol2 || b-u < tol2)
         d=SIGN(tol1,xm-x);
   }
} else {
   d=CGOLD*(e=(x >= xm ? a-x : b-x));
}
u=(absVal(d) >= tol1 ? x+d : x+SIGN(tol1,d));
fu=f1Powell(u);
if (fu <= fx) {
   if (u >= x) a=x; else b=x;
   SHFT(v,w,x,u);
   SHFT(fv,fw,fx,fu);
} else {
   if (u < x) a=u; else b=u;
   if (fu <= fw || w == x) {
      v=w;
      w=u;
      fv=fw;
```

```
            fw=fu;
         } else if (fu <= fv || v == x || v == w) {
            v=u;
            fv=fu;
         }
      }
   }
   if (verbose>1){
      cpmwarning("Mini::brent() stopped after"&cpmwrite(iterMaxBrent)&
      "iterations");
   }
   xmin=x;
   iterBrent=CpmRootX::sup(iterBrent,iterB);
   return fx;
}

/**************** function powell ************************************/

bool Mini::linMinPowell(R_Vector& p, R_Vector& xi, R& fret)
{
   R ax=0.1, xx=1., bx=2.;
   R xmin,fx,fb,fa;
   pPowell=p;
   xiPowell=xi;
   bool b=brak(ax,xx,bx,fa,fx,fb);
   if (b==false) return false;
   fret=brent(ax,xx,bx,xmin);
   xi *= xmin;
   p += xi;
   return true;
}

bool Mini::powell(R_Vector& x, R_Matrix xi, R& fret)
/* Powell's direction set method in multidimensions.
Minimization of a function func of n variables. Input consists of an
initial starting point p[1,...,n]; an initial matrix xi[1,...n][1,...n]
whose columns contain the initial set of directions (usually the n unit
vectors); and ftol, the fractional tolerance in the function value such
that failure to decrease by more than this ammount on one iteration
signals doneness. On output, p is set to the best point found, xi is
the then-current direction set, fret is the returned function value at p,
and iter is the number of iterations taken.
*/
{
   static Word loc("Mini::powell(...)");
   cpmassert(x.isVal(),loc);
   Z i,ibig,n=x.dim();
   R t,fptt,fp,del;
   R_Vector pt=x;
   R_Vector ptt, xit;
```

```
   fret=ff_(x);
   nEval_++;
   for (iter=1;;iter++) {
      if (verbose>2) cpmvalue("iter",iter,2);
      fp=fret;
      ibig=0;
      del=0.0;
      for (i=1;i<=n;i++) {
         xit=xi[i];
         fptt=fret;
         bool lmpRes=linMinPowell(x,xit,fret);
         if (lmpRes==false) return false;
         if (absVal(fptt-fret) > del) {
            del=absVal(fptt-fret);
            ibig=i;
         }
      }
      if (2.0*absVal(fp-fret) <= ftol*(absVal(fp)+absVal(fret))) {
         return true;
      }

      if (iter == iterMaxPowell){
         Word messText="More than "&cpmwrite(iterMaxPowell)
         &" iterations in function Mini::powell()";
         cpmwarning(messText);
         return false;
      }
      ptt=2.0*x-pt;
      xit=x-pt;
      pt=x;
      fptt=ff_(ptt);
      nEval_++;
      if (fptt < fp) {
         t=2.0*(fp-2.0*fret+fptt)*SQR(fp-fret-del)-del*SQR(fp-fptt);
         if (t < 0.0) {
            linMinPowell(x,xit,fret);
            xi[ibig]=xit;
         }
      }
   }
}

R Mini::minimize(R_Vector& x)
/*
x is a vector which the function ff_ accepts as argument. After calling
function minimize it will be transformed to a vector for which f attains
a minimum value. If f has several minimum values, the one which is thus
obtained is the one which is got by Powells method.
*/
{
```

```
   Word loc("Mini::minimize(R_Vector& x)");
   Z mL=3;
   CPM_MA
   Z dx=x.dim();
   R_Matrix xi(dx); // dx*dx zero-matrix (was unit matrix
      // in the KAGUM version)
   xi=xi.net(1);  // unit matrix
   R res=0;
   bool b=powell(x,xi,yMin);
   if (b){
      res=yMin;
      suc_=B(true);
   }
   else{
      cpmwarning(loc&" no useful value found, 0 returned");
      // suc_=false by initialization
   }
   CPM_MZ
   return res;
}

void Mini::show()const
{
   cpmcerr<<endl<<"diagnosis of an instance of Mini";
   cpmcerr<<endl<<"verbose= "<<verbose;
   cpmcerr<<endl<<"iterMaxBrent= "<<iterMaxBrent;
   cpmcerr<<endl<<"iterMaxPowell= "<<iterMaxPowell;
   cpmcerr<<endl<<"GOLD= "<<GOLD;
   cpmcerr<<endl<<"GLIMIT= "<<GLIMIT;
   cpmcerr<<endl<<"TINY= "<<TINY;
   cpmcerr<<endl<<"EPS= "<<EPS;
   cpmcerr<<endl<<"TOL= "<<TOL;
   cpmcerr<<endl<<"CGOLD= "<<CGOLD;
   cpmcerr<<endl<<"ZEPS= "<<ZEPS;
   cpmcerr<<endl<<"ftol= "<<ftol;
   cpmcerr<<endl<<"yMin= "<<yMin;
   cpmcerr<<endl<<"nEval_= "<<nEval_;
   cpmcerr<<endl<<"iter= "<<iter;
   cpmcerr<<endl<<"iterBrent= "<<iterBrent;
   cpmcerr<<endl<<"callsBrent= "<<callsBrent;
   cpmcerr<<endl<<"done= "<<done;
   cpmcerr<<endl<<"ff_Defined= "<<ff_Defined;
   cpmcerr<<endl<<"xStart= "<<xStart;
   cpmcerr<<endl<<"xMin= "<<xMin;
}
```

121 cpmmpi.h

```
//? cpmmpi.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_MPI_H_
#define CPM_MPI_H_

/*
   Description: Functions to send and receive objects of C++
   type string via MPI functions. The length of the strings
   is handled automatically. This header file will be included in
   cpminterfaces.h and, therfore, should not include files like
   cpmsystem.h, cpmtypes.h, cpmword.h

*/

#include <cpmdefinitions.h>
   // only to make known CPM_USE_MPI
   // no additional C+- stuff needed here

#include <string>

#if defined(CPM_USE_MPI)
   #include <mpi.h>
      // MPIPro
      // including within the namespace caused trouble!
#endif

namespace CpmMPI{

using std::string;

extern int mpiverbose;

#if defined(CPM_USE_MPI)

////////////////////////////// class Com ////////////////////////////////

class Com{
// class version of MPI's communicator concept for parallel computing
   // for which rank varies from 1 to size in order
   // to cooperate with CpmArrays

protected:
   MPI_Comm mc;
   int size;
   int rank; // starts with 1
public:
```

```
   static int tagFromToStatic(int f, int t,int size_)
   // returned simply f*t till 2004-07-07. The footnote to p. 254 of GLS
   // (Gropp,Lusk,Skjellum: using MPI) indicates that using the same tag
   // several times in succession might be not so desastrous as one could
   // expect it to be. Nevertheless, one should ensure that the function
   // is injective (appart from tagFromToStatic(f, t, size_)==
   // tagFromToStatic(t, f, size_) )
   {
      int i,j;
      if (f<=t){ i=f-1; j=t;}
      else{ i=t-1; j=f;}
      return size_*i+j;
   }
   int tagFromTo(int f, int t)const{ return tagFromToStatic(f, t, size);}
   Com(MPI_Comm comm=MPI_COMM_WORLD);
   int getSize()const{ return size;}
      // number of processes between one can communicate via sendStr
      // and recStr
   int getRank()const{ return rank;}
   bool sendStr(const string& s, int dest, int tag)const;
      // sending C++-strings without having to input their length
      // return value true means success.
      // I use string as a universal data type singe the C++ class
      // stringstream offers efficient read and write.
      // 1<=dest<=size
   bool bcastStr(string& s)const;
      // broadcasts a string from the process with rank 1 to
      // all other processes
   bool recStr(string& s, int source, int tag)const;
      // receiving strings
      // 1<=source<=size
   int cyc(int i)const;
      // the return value k is equal i modulo size
      // also 1<=k<=size. Useful tool although one could argue that this
      // function belongs to the topic of 'communicator topologies'
      // and not to the communicator itself

// communication patterns:

// The aim of the following functions is to ensure a pairing
// of send and receive activities that avoids deadlock. Although
// it is not difficult to gain an intuitive insight saying that
// the combinations as implemented in these functions are safe against
// deadlock, I have no formal proof for this so far.
// Supporting points are
// 1. empirical: no exception found so far
// 2. philosophical: no simpler expression conceivable except
//    of obviously wrong ones.

// template member functions are very helpful here.
```

```
   template <class T, class V>
      // T has to define send() and rec(), V has to have
      // the basic properties of V<T> (see cpmv.h)
   void sendAndRec(const T& t, V& v)const
/*
   Typical application in a SPMD (single program, multiple data )
   situation: (using C+- classes)
   Com com;
   Z size=com.getSize();
   T t(...); // the contribution of com.getRank()
   V<T> v(size); // container for the contributions of others
   com.sendAndRec(t,v); // communication
   R sum=0; // starting to use the communicated stuff
   for (Z i=1;i<=size;i++) sum+=g(v[i]); // R g(const T&)
       // A cummulative effect of all processes is thus available for
       // each process.
*/
   {
      for (int i=1;i<=size;i++){
         if (i==rank){ // OK for size==1, rank==1
            v[i]=t;
         }
         else if (rank<i){
            t.send(i,tagFromTo(rank,i));
            v[i].rec(i,tagFromTo(i,rank));
         }
         else{
            v[i].rec(i,tagFromTo(i,rank));
            t.send(i,tagFromTo(rank,i));
         }
      }
   }

   template <class V>
      // V has to have the basic properties of V<T>, where
      // T defines send() and rec().
   void exchange(V const& s, V& v)const
/*
   Typical application in a SPMD (single program, multiple data )
   situation: (using C+- classes)
   Com com;
   int size=com.getSize();
   int rank=com.getRank();
// the contribution of com.getRank()
   V<T> s(size);
   for (int i=1;i<=size;i++){
      for (i==rank) continue;
         // there is nothing to exchange from rank to rank
      s[i]=....; // this is what rank has to say to i
   }
```

```
   V<T> v(size); // container for the messages from others
   com.exchange(s,v); // communication
   R sum=0; // starting to use the communicated stuff
   for (int i=1;i<=size;i++) sum+=g(v[i]); // R g(const T&)
       // Process rank has available the messages from its
       // partners all at once
*/
   {
      for (int i=1;i<=size;i++){
         if (i==rank){
            v[i]=s[i]; // was continue till 2003-06-10
         }
         else if (rank<i){
            s[i].send(i,tagFromTo(rank,i));
            v[i].rec(i,tagFromTo(i,rank));
         }
         else{
            v[i].rec(i,tagFromTo(i,rank));
            s[i].send(i,tagFromTo(rank,i));
         }
      }
   }

//   export // not working with MS cl
   template <class T>
      // see sendAndRec() for the assumed properties of T
   void bcast(T& t)const
      // Call to bcast has to be preceeded by initialization of t in rank
      // 1.
      // After call for each rank t is assured to have the value which
      // rank 1 had provided. Execution of this function involves each
      // rank with at least one send or receive activity. Since rank 1
      // sends to all processes, and send is not considered done before
      // data are received, rank 1 can only continue after all receives
      // are done in all other processes. Any other process also can
      // only continue after it received. Who received first may hurry
      // away not waiting till others are also ready with receiving. So
      // this is not a strict synchronization mechanism.
      // However, each process that hurries away must have completed its
      // receive and this can only be achieved if all messages sent
      // earlier have been received to. So the function should force
      // completion of all communication tasks that  were pending at the
      // moment of calling bcast. Similar considerations apply to the
      // other collective communication functions.

      // A definition of this template providing diagnostics would be
      // desirable. In order to have cpmmessage available one could in
      // principle include  cpmsystems.h into the present file and
      // implement messages much like those of sendStr and recStr. In our
      // special situation this is not working since cpmmpi.h has to be
```

```
   // included in cpminterfaces.h where cpmsystem.h should not yet be
   // known. So one had to move the implementation to cpmmpi.cpp which,
   // however, asks for the additional qualification 'export'
   // (see Vandevoorde, Josuttis: C++ Templates, p. 68) which MS cl
   // does not know

{
   if (size==1) return;
   if (rank==1){
      for (int i=2; i<=size;i++) t.send(i,tagFromTo(1,i));
   }
   else  t.rec(1,tagFromTo(1,rank));
}

template <class T, class V>
void gather(const T& t, V& v)const
// The out-commented messages made me aware of the actual timing of
// processes in my emulating MPICH runs: Writing of moviefiles for some
// processes may be in progress long before function step() is done for
// all tasks.
{
   if (size==1){ v[1]=t; return;}
   if (rank!=1){
      t.send(1,tagFromTo(rank,1));
   }
   else{
      v[1]=t;
      for (int i=2;i<=size;i++) v[i].rec(i,tagFromTo(i,1));
   }
}

template <class T, class V>
   // see sendAndRec() for the assumed properties of T and V
void gatherXL(const T& t, V& v, int n)const
   // clear from code
{
   if (size==1){ v[1]=t; return;}
   if (n<2) {
      gather(t,v);
      return;
   }
   if (rank!=1){
      t.sendXL(1,tagFromTo(rank,1),n);
   }
   else{
      v[1]=t;
      for (int i=2;i<=size;i++) v[i].rec(i,tagFromTo(i,1));
   }
}
```

```
   template <class T, class V>
      // see sendAndRec() for the assumed properties of T and V
   void scatter(T& t, const V& v)const
      // clear from code, not yet used or tested
   {
      if (rank==1){ // directly OK for size==1
         t=v[1];
         for (int i=2;i<=size;i++) v[i].send(i,tagFromTo(1,i));
      }
      else{
         t.rec(1,tagFromTo(1,rank));
      }
   }

   template <class T, class V>
   void scatterXL(T& t, const V& v, int n)const
      // clear from code, not yet used or tested
   {
      if (rank==1){
         t=v[1];
         for (int i=2;i<=size;i++) v[i].sendXL(i,tagFromTo(1,i),n);
      }
      else{
         t.rec(1,tagFromTo(1,rank));
      }
   }
};

#else // not defined(CPM_USE_MPI)

class Com{ //trivial implementation of Com

public:
   static int tagFromToStatic(int f, int t,int size_)
   {f;t;size_;return 0;}
   static int tagFromTo(int f, int t){ f;t;return 0;}
   Com(){}
   int getSize()const{ return 1;}
   int getRank()const{ return 1;}
   bool sendStr(const string& s, int dest, int tag)const
   {s;dest;tag; return true;}
   bool bcastStr(string& s)const{s; return true;}
   bool recStr(string& s, int source, int tag)const
      {s; source; tag; return true;}
   int cyc(int i)const{ i;return 1;}

   template <class T, class V>
      void sendAndRec(const T& t, V& v)const{t;v;}

   template <class V>
```

```
      void exchange(const V& s, V& v)const
      { v[1]=s[1];}

   template <class T>
      void bcast(T& t)const{t;}

   template <class T, class V>
      void gather(const T& t, V& v)const{t;v;}

   template <class T, class V>
      void gatherXL(const T& t, V& v, int n)const{t;v;n;}

   template <class T, class V>
      void scatter(T& t, const V& v)const{t;v;}

   template <class T, class V>
      void scatterXL(T& t, const V& v, int n)const{t;v;n;}
};

#endif // #if defined(CPM_USE_MPI)

extern CpmMPI::Com Cpm_com;
void initialize(int* a, char*** b);
void finalize();

} // namespace

#endif // #if defined(CPM_MPI_H)
```

122 cpmmpi.cpp

```
//? cpmmpi.cpp
//? Status of work 2008-10-25.
//?
/*
   cpmmpi.cpp

   Description: see cpmmpi.h
      can be made part of the source file set even if
      CPM_USE_MPI is not defined
*/
#include <cpmmpi.h>
#include <cpmtypes.h>
#include <cpmvo.h>

int CpmMPI::mpiverbose=1;

namespace{
   const int mL1=10;
   const int mL2=20;
}

   // limits against which mpiverbose will be tested
CpmMPI::Com CpmMPI::Cpm_com;
   // initialized by default constructor

#if defined(CPM_USE_MPI)

#include <cpmsystem.h>

using namespace CpmMPI;

using CpmRoot::Word;
using CpmRoot::Z;
using CpmRoot::R;
using CpmRootX::Z1;
using CpmArrays::Vio;
using CpmArrays::Vo;

void CpmMPI::initialize(int* a, char*** b){ MPI_Init(a,b);}

void CpmMPI::finalize(){ MPI_Finalize();}

CpmMPI::Com::Com(MPI_Comm comm):mc(comm)
{
   int mpiflag;
   MPI_Initialized(&mpiflag);
   if(mpiflag) { // 01-09-25: mpiflag introduced by John Zollweg,
```

```
      // Cornell university, to avoids warnings
      MPI_Comm_size(comm,&size);
      MPI_Comm_rank(comm,&rank);
      ++rank;
   }
}

Z CpmMPI::Com::cyc(Z i)const
{
   while(i>size) i-=size;
   while(i<1) i+=size;
   return i;
}

// not working code for non-blocking sends and receives is commentarized
bool CpmMPI::Com::sendStr(const string& s, Z dest, Z tag)const
{
   static Z shL=100; // show length
   static Z counter=0;
   counter++;
   Z n=1+(Z)s.size();
   if (mpiverbose>mL1){
      ostringstream mes;
      mes<<"sendStr( dest="<<dest<<" tag="<<tag<<") started n="
         <<n<<"; send counter="<<counter;
  //    if (counter==21) mes<<" the string to send is "<<s; // experiment
  // ad
  // hoc
      cpmmessage(Word(mes.str()));
   }
   Z tag1=tag;
   Z tag2=tag+1;
   Z* np=new Z(n);
 //  MPI_Request request;
 //  MPI_Isend(np,1, MPI_INT, dest-1, tag1, mc,&request);
   MPI_Ssend(np,1, MPI_INT, dest-1, tag1, mc);
   if (mpiverbose>mL2){
      ostringstream mes;
      mes<<"sending the number done, tag="<<tag1;
      mes<<" next send will use tag="<<tag2;
      cpmmessage(Word(mes.str()));
   }
   delete np;
   if (mpiverbose>mL2){
      ostringstream mes;
      if (n<=shL)
         mes<<endl<<"string to send is: "<<s;
      else
         mes<<endl<<"string to send not shown since longer than "<<shL;
      mes<<endl<<"end of string to send";
```

```
      cpmmessage(Word(mes.str()));
   }

  Z res= MPI_Ssend((void*)s.c_str(), n, MPI_CHAR, dest-1, tag2, mc);
  // Z res= MPI_Isend((void*)s.c_str(), n, MPI_CHAR, dest-1, tag2,
  // mc,&request);

   if (mpiverbose>mL2){
      ostringstream mes;
      mes<<"sending the string done";
      cpmmessage(Word(mes.str()));
   }

   bool b=(res==MPI_SUCCESS);
   if (mpiverbose>mL1){
      ostringstream mes;
      mes<<"sendStr( dest="<<dest<<" tag="<<tag<<"), n="<<n<<
         " done; send counter="<<counter;
      cpmmessage(Word(mes.str()));
   }
   if (!b) cpmerror("error in CpmMPI::sendStr");
   return b;
}

bool CpmMPI::Com::recStr(string& s, Z source, Z tag)const
{
   static Z counter=0;
   counter++;
   if (mpiverbose>mL1){
      ostringstream mes;
      mes<<"recStr( source="<<source<<" tag="<<tag<<") started"
         <<"; receive counter="<<counter;
      cpmmessage(Word(mes.str()));
   }
   Z tag1=tag;
   Z tag2=tag+1;
   MPI_Status status;
  // MPI_Request request;
   Z* np=new Z;
   MPI_Recv(np,1,MPI_INT,source-1,tag1,mc,&status);
 //  MPI_Irecv(np,1,MPI_INT,source-1,tag1,mc,&request);
   // The non-blocking version does not work in my program, not clear why
   Z n=*np;
   if (mpiverbose>=mL1){ // >= intentionally to have a control mode
      // in which only the most important events get reported
      // for all other tests we have mpiverbose>mL
      cpmmessage("string received of length n="&cpmwrite(n));
   }
   char* data=new char[n];
 //  Z out=MPI_Irecv(data,n,MPI_CHAR,source-1,tag2,mc,&request);
```

```
   Z out=MPI_Recv(data,n,MPI_CHAR,source-1,tag2,mc,&status);
   ostringstream ost;
   ost<<data;
   s=ost.str();
   if (mpiverbose>mL1){
      ostringstream mes;
      mes<<"recStr( source="<<source<<" tag="<<tag<<") done; n="
         <<n<<"; receive counter="<<counter;
      if (mpiverbose>mL2){
         mes<<endl<<"begin of string received "<<endl;
         mes<<s;
         mes<<endl<<"end of string received";
      }
      cpmmessage(Word(mes.str()));
   }
   bool b=(out==MPI_SUCCESS);
   delete[] data;
   delete np;
   if (!b) cpmerror("error in CpmMPI::recStr");
   return b;
}

//////// some collective communication ////////////////////////////////
bool CpmMPI::Com::bcastStr(string& s)const
{
   if (mpiverbose>mL1){
      ostringstream mes;
      mes<<"bcastStr() started";
      cpmmessage(Word(mes.str()));
   }
   bool b=true;
   if (rank==1){
      for (Z i=2; i<=size;i++){
         b=b&&sendStr(s,i,size+i);
      }
   }
   else{
      b=b&&recStr(s,1,size+rank);
   }
   if (!b) cpmerror("error in CpmMPI::bcastStr");
   if (mpiverbose>mL1){
      cpmmessage("bcastStr() done");
   }
   return b;
}

#else // !defined(CPM_USE_MPI)

void CpmMPI::initialize(int* a, char*** b){a;b;}
```

```
void CpmMPI::finalize(){}

#endif  // #if defined(CPM_USE_MPI)
```

123 cpmmultiimage.h

```
//? cpmmultiimage.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_MULTIIMAGE_H_
#define CPM_MULTIIMAGE_H_
/*
   Description: Defines template class MultiImage<T>
*/
#include <cpmimage.h>
#include <cpmdiscretespace.h>
#include <cpmframes.h>

namespace CpmImaging{

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::L;
   using CpmRoot::Pi;

   using CpmArrays::V;
   using CpmArrays::VV;
   using CpmArrays::VVV;
   using CpmArrays::X2;
   using CpmArrays::R_Vector;
   using CpmArrays::R_Matrix;

   using CpmFunctions::F;

   using CpmLinAlg::R2;
   using CpmLinAlg::R3;

   using CpmGraphics::Img24;
   using CpmGraphics::rgb;
   using CpmGraphics::Frame;
   using CpmGraphics::Frames;

//////////////////////// class MultiImage<> ////////////////////////////////

template<class T>
class MultiImage: public V<Image<T> >, ImagingTools {
// real-valued multi-plane image,  typedef MultiImage<R> ImgR
// images with potentially more than one plane as vectors of images.
// T is the type of pixel values. No positivity of the pixel values
// is assumed. Main application for T = R, for details see
// class Image<T> in file cpmimage.h
// Wide range of methods for processing color images. Convenient
// conversion back and forth with C+- basic color image type Img24.
```

```
// Notice: typedef MultiImage<R> ImgR;
// at the end of the class.

   typedef V< Image<T> > Base;

public:

   MultiImage( const V< Image<T> >& vimg):V< Image<T> >(vimg){}
      // upcast, no new member introduced in deriving MultiImage<T>
      // from V< Image<T> >

// constructors which do not provide pixel values

   MultiImage(Z p, Z m, Z n ):
      // p: number of image planes (e.g. 3 for RGB images)
      // m: number of lines
      // n: number of colums
   V<Image<T> >(p,Image<T>(m,n)){}

   explicit MultiImage( Z p ):
      // p: number of image planes (e.g. 3 for RGB images)
      V<Image<T> >(p,Image<T>()){}

   MultiImage(void):
      // default constructor
   V<Image<T> >(1,Image<T>()){}

// constructors which import pixel values from other data structures

   MultiImage(Z p, const Image<T>& im):
      // constructs a p-fold repetition of im
      V<Image<T> >(p,Image<T>(im)){}

//template <class Typ2>
   // does not compile, so type Typ2 is made semi-flexible via
   // pre-processor (we could make Typ2 a second template parameter
   // of MultiImage - this would be very unnatural and awkward)
#define Typ2 L
   MultiImage(Z p, const Image<Typ2>& im, T (*f)(Z,Typ2));
      // constructs a p-fold repetition of the geometry of im
      // and coloring the pixels using im together with a look-up
      // function f. f(i,val) is the pixel value to which the pixel
      // value val of im is translated in the i-th image plane of
      // the MultiImage to be constructed

   MultiImage(Z p, const VV<T>& y):
      // constructs a p-fold repetition of a image constructed from
      // a Matrix<T,Z>
   V<Image<T> >(p,Image<T>(y)){}
```

```
   MultiImage( const VVV<T>& y);
      // constructs a MultiImage<T> from a VVV<T> (which is the
      // natural triple-indexed data structure

   MultiImage( const VV<T>& y);
      // constructs a MultiImage<T> from a VVx<T> (which is the
      // natural double-indexed data structure

   MultiImage(Frame const& fr);
      // constructor from a Frame (graphics window)
      // and its pictorial content. This is a convenient method
      // if one wants to apply more sophisticated image processing
      // to an image created on screen. Works only for T=R

   MultiImage(Img24 const&);
      // construction native basic color image format.
      // Only for T=R.

   MultiImage(VV<Img24> const& vv, Z m, Z n);
      // construct a MultiImage of m lines an n columns
      // by composing a matrix of Img24-images into
      // it by suitable scaling.
      // Only for T=R

// trivial access functions

   Z getLin(void)const{ return ((*this)[1]).getLin();}
      // get number of lines. This refers to the first image plane.
      // Presently all constructors generate multi-images
      // for which pixel count is independent of the image plane
      // Notice that ((*this)[1]).getLin() would be terribly
      // inefficient since calculating (*this)[1] implies a return
      // and thus a copy constuctor for images!.
      // 96-1-29: this is no longer true, since now (*this)[1] would
      // be a reference

   Z getCol(void)const{ return ((*this)[1]).getCol();}
      // get number of columns

   Z h()const{ return getLin();}
   Z w()const{ return getCol();}

   Z dim1()const{ return getLin();}
   Z dim2()const{ return getCol();}

   Z getPla(void)const{ return Base::dim();}
      // get number of 'planes'

// pixel-value access
```

```
   T getPix(Z h, Z i, Z j )const{ return ((*this)[h]).getPix(i,j);}
      // acessing pixel values outside the image possible.
      // continuation as constant

   T getPix(Z i, Z j )const;
      // same as previous function but mean value over all channels
      // returned

   void putPix(Z h, Z i, Z j, T const& t){((*this)[h]).putPix(i,j,t);}

   void putPix(Z i, Z j, Color const& c){
      ((*this)[1]).putPix(i,j,c.red());
      ((*this)[2]).putPix(i,j,c.green());
      ((*this)[3]).putPix(i,j,c.blue());
   }

   T getLocalMean(Z h, Z i, Z j, const V<Mask>& a)const
      // forming a local mean value by masks which are allowed to
      // vary from image plane to image plane
   { return ((*this)[h]).getLocalMean(i,j,a[h]);}

   T operator()(Z h, R x, R y, Z method=1)const
      // returns the value at a point in the h-th plane with respect to a
      // normal coordinate system (origin lower-left, x-axis
      // to the right, y-axis vertically ascending. The upper-right
      // image corner has coordinates x=1, y=1. This system
      // will be referred to as base system. Pixelation, or
      // bilinear interpolation according to 'method' as in function
      // interpolate()
   { return ((*this)[h])(x,y,method);}

   T operator()(R x, R y, Z method=1)const;
      // same as previous function but mean value over all channels
      // returned

   R2 minMax(R xL=0, R xU=1, R yL=0, R yU=1)const;
      // Let R2 lim = minMax(); Then lim.x is the lowest of all pixel
      // values (after conversion to R, all image components)
      // taken into account) and lim.y is the highest.

// image transformations

   void setRange(T valMin=0, T valMax=255);
      // changes the range of the image to the specified range by
      // an affine transformation

   void gamma(R gammaI);
      // Uses a common gamma curve for all planes such that the total
      // range of gray-values remains unchanged. Gamma has the usual
      // meaning:
```

```
   // gamma=1 for linear curve, gamma <1 for enhancing shadow
   // contrast,
   // gamma > 1 for enhancing contrast in full light.

MultiImage<T> interpolate(Z m, Z n, Z method)const;
   // returns a m*n MultiImage created from *this by a interpolation
   // method selected by the parameter method.
   // method =1 means bilinear interpolation and method=0 means
   // pixelation. The latter is useful if the number of pixels is to
   // be compared to expectations.

Image<T> combine( R (*f)(const R_Vector& x) )const;
   // Returns a single-component image, obtained by using function
   // f to combine the pixel values in the various planes into one
   // pixel value

VVV<T> toVVV()const;
   // conversion to the natural mathematical type

Img24 toImg24()const{ return Img24(toVVV());}
   // conversion to Img24, the image type of namespace CpmGraphics
   // This works for MultiImage<R> only.
   // If all pixel values are in the range [-0.5,255.5] the negative
   // data are set to 0, thode above 255 to 255; and all other data are
   // converted to L in the natural manner. If some the pixel values
   // don't fit the range [-0.5,255.5], a linear transformation of the
   // pixel data (same transformation for all image planes) will fit
   // the pixel range into [0,255].

void insertPattern(R X, R Y, R dX,
   const  F<C, Va<T> >& f, const F<C,B>& g,
   R x=0, R y=0, R dx=1, R dy=1, B average=B(false));
   // modifying a part of an image by values derived from function
   // values on the part
   //   X <= Re(z) <= X+dX
   //   Y <= Im(z) <= Y+dY
   // of the complex plane. dY is calculated from dX by aspect ratio
   // g(z)=trueUM indicates that f(z) will be calculated
   // and the values f(z)[1],... f(z)[p] will be stored to the pixel
   // corresponding to z in image planes 1,...p.
   // If average=trueUM, not only z belonging to the current pixel is
   // used but four positions and an average is formed.
   // This was designed to give alias-reduced gray-tone images for
   // binary patterns. In this case T has to such that automatic
   // type conversion from and to R is defined. L is a valid choice
   // for T

void insert(MultiImage<T> const& img, Rect const& rect, Z scale=0);
   // changes *this by inserting the image img 'into the subarea rect'
   // Since the format of img might not fit to this prescription
```

```
      // verbally, there are options, controlled by parameter scale which
      // give a precise meaning to this:
      // scale==0:
      // The image is inserted on a pixel by pixel basis, starting at the
      // left upper corner of rect. If img is larger (pixel-wise) than
      // rect, then  only the part in rect is changed.
      // scale==1:
      // The image img is scaled (respecting the aspect ratio) such that
      // it just can be held completely by rect. Scaling is done by
      // bilinear interpolation

// image display on screen

   void mark(Graph& gr, R2 lim, Z method=0, Z faithful=1)const;
      // Writes the image *this on the Graph gr.
      // This does not display the image. gr.display() needs to
      // be called for showing the image on screen.
      // The function Frame::response transforms the pixel values
      // into integers which are interpreted as display colors.
      // This function is controlled by the parameter Frame::gamma,
      // which has the meaning usually associated with gamma.
      // The color-range is adjusted such that the lowest color value
      // corresponds to limits.x and the highest color value to limits.y.
      // All sub-images are treated via the same transfer function.
      // The frame gr also determines the number of pixels to
      // be displayed. Since this may not coincide with the number of
      // pixels of the image, a suitable transformation has to be done.
      // This is controlled via parameters method and faithful.
      // The parameter 'method' controls pixel rendition: 0 normal
      // pixelation 1 bilinear interpolation. If the parameter 'faithful'
      // is 1, the image will be placed in such a manner to the upper
      // left position that its aspect ratio is correctly reproduced on
      // the screen. Otherwise it is made filling the whole frame.
      // Addition 2004-10-17: For method=2, the averaging interpolator
      // is used which however for images with many pixels is very very
      // slow. The method could be of value for images with very small
      // number of pixels that nevertheless have to be shown on a
      // large screen area without annoyingly obvious pixel structure.
      // This function may also be used as a testbed for the
      // averaging interpolator (see cpmalgorithms.h)

   R2 mark(Graph& gr,Z method=0, Z faithful=1)const
      // The actual range of the image will be mapped to
      // display range. All sub-images are treated via the same
      // transfer function
    {
      R2 lim=minMax();
      mark(gr,lim,method,faithful);
      return lim;
    }
```

```
    void markDirect(Graph& gr, Z method=0, Z faithful=1)const
      // The pixel value range [0,255] will be mapped to
      // display range. This is the is the adequate functionality
      // for images got by reading TARGA files or other file formats
      // which adopt 1 BYTE per pixel and image plane.
    {
      R2 lim(0,255);
      mark(gr,lim,method,faithful);
    }

   void print(void);
      // outputs pixel count to screen

// multiplication of images for flat field correction

   MultiImage<T>& operator *=(const T& t);

   MultiImage<T>& operator *=(const Image<T>& im);

   MultiImage<T>& operator *=(const MultiImage<T>& im);

   MultiImage<T>& operator /=(const Image<T>& im);

   MultiImage<T>& operator /=(const MultiImage<T>& im);

   MultiImage<T> cut(const Rect& a)const;
      // reduces the size of the image by cutting it down to the
      // rectangular subframe defined by Rect a.

   MultiImage<T> cut(R xL, R xU, R yL, R yU)const;
      // reduces the size of the image by cutting the borders. All
      // arguments have to be in the interval [0,1].
      // The part of the image which is left after cut has normal
      // coordinates (x,y) with respect to the original which satisfy
      //   xL <= x <= xU
      //   yL <= y <= yU
      // Normal coordinate system: origin lower-left, x-axis
      // to the right, y-axis vertically ascending. The upper-right
      // image corner has coordinates x=1, y=1.

   MultiImage<T> cut(Z2 cLL, Z2 cUR)const;
      // similar to the previous function. Here the lower left corner
      // and the upper right corner are given as pixel positions in
      // graphics format.

   MultiImage<T> clip(R clipL, R clipU)const;
      // x.clip(a,b) takes values between a*x.min() and b*x.max()

   MultiImage<T>& operator -=(const MultiImage<T>& im);
```

```
      // returns *this-im, the pixelwise difference of images.
      // *this and im are assumed to coincide in numbers of planes,
      // lines, and columns

   MultiImage<T>& apply(const MultiImage<T>& im,
      const F< X2<T,T>, T>& f);
      // general operation of one multi-image onto another  multi-image
      // of the same pixel type

    MultiImage<T>& apply(const MultiImage<T>& im,
       const V<F<X2<T,T>,T> >& f);
      // Analogous to previous function. Uses different 'functions' for
      // combining the pixel values in the various planes of the
      // multi-images. These 'functions' are the components of the
      // function array f.

   MultiImage<T> operator -(const MultiImage<T>& im1)const
    { MultiImage<T> res=*this; return res-=im1;}
      // difference of images in normal notation

    MultiImage<T> operator +(const MultiImage<T>& im1)const
       // sum of images in normal notation
    {
       MultiImage<T> im2=im1;
       T fac(-1);
       im2*=fac;
       MultiImage<T> res=*this; return res-=im2;
    }

    MultiImage<T> overLay(MultiImage<T> const& im1, R fac=0.5)const
       //: over-lay
       // adds *this and a 'dimmed' (or intensified) version of im1
       // and returns the result.
    {
       MultiImage<T> im2=im1;
       T factor(-fac);
       im2*=factor;
       MultiImage<T> res=*this; return res-=im2;
    }

// comparison of images

   R norm1(void)const;
      // sum over the absolute values over all pixel values, divided by
      // number of pixels

   R norm2(void)const;
      // sum over the squares of all pixel values, divided by number of
      // pixels.
      // Square root of all of that. (Normalization is such, that the
```

```
   // norm of a constant image is equal to the pixel value)

X2<R_Vector,Z2> hotspot(Z dm=0, Z dn=0);
   // returns the pixel value for all planes at the point where the
   // sum of all pixel values is maximum
   // actually the mean of a (1+2*dm)*(1+2*dn) neighborhood is given

MultiImage<T> conv(const R_Matrix& a, Z si=0, Z sj=0);
   // Matrix a is taken as an convolution kernel with a(ci+si,cj+sj)
   // considered the center of the kernel.
   // Here
   // ci := 1+dim1(a)/2
   // cj := 1+dim2(a)/2
   // which is the actual center if the dimensions of a both are odd
   // which is the normal case.
   // No normalization will be done on a, since this can be done more
   // flexibly in advance.

MultiImage<T> conv(const R& sigma, Z dL=5);
   // The result is the convolution of this with a numerical
   // approximation to a rotationally symmetric Gaussian distribution
   // where sigma is the standard deviation of center distance r.
   // The distribution is approximated as a (2d+1)(2d+1)-matrix with
   // maximum values arround the center, where d is reasonably
   // calculated from sigma. dL is a limiter for d. d<=dL is
   // guaranteed.

MultiImage<T> conv(const V<Mask>& a)const;
   // modern form of convolution, defined by a family of masks,
   // one for each image plane

MultiImage<T> conv(const Mask& a)const;
   // modern form of convolution, defined by one mask,
   // to be applied to each image plane

MultiImage<T> deScreen(R lim=0.1, R rim=0.1, Z p=3)const;
   // Not yet mature algorithm. Good exercize to work with
   // Fourier transformed images. Showed the need for defining
   // a few new functions.
   // 0<=lim<=1, the smaller lim the more changes will be made
   // 0<=rim<=1, the smaller rim the lower the frequencies that can
   // be affected
   // The masks for detecting artificial maxima in momentum space
   // has total width 2*p+1 pixels

MultiImage<T> fourierFilter(const Fa<R2,R>& resp)const;
   // multiplies the fourier transform by function rep which
   // takes meaningful arguments in [0,1]x[0,1], where 0 corresponds
   // to frequency 0 and 1 to Nyquist frequency. The negative
   // frequencies (which are not independent) are treated
```

```
   // automatically in a consistent manner.
   // T is assumed to allow meaningful two-way conversion to R.
   // The resulting image has lines and columns equal to powers of
   // 2. The input image is not required to have this property.
   // For calculating its fourier transform we fill the missing pixels
   // in by 0.

MultiImage<T> fftFilter(const F<C,C>& resp)const;

R getPixConv(Z h, Z i, Z j, const R_Matrix& a, Z si=0, Z sj=0);
   // makes "Faltung" with R_Matrix in pixel plane
   // h: Colorplane, i,j: pixel

R_Vector getPixConvPla( Z i, Z j, const R_Matrix& a);
   // makes "Faltung" with R_Matrix in color palne
   // and returns a vector with all new colorvalues of the pixel i,j

MultiImage<T> cosmetic(Z Meth=1,Z s1=20,Z s2=50,Z s3=200,Z s4=230,
   Z w1=1,Z w2=0,Z w3=1);
   // cosmetics the image using functions "GetW, SignalValue,
   // getPixConv"
   // the funktion makes a multimage unsharp depending on the curve of
   // function GetW
   // the paramaters are only for then funktion "GetW"

MultiImage<T> cosmetic2(Z Meth=1,Z s1=20,Z s2=50,Z s3=200,Z s4=230,
   Z w1=1,Z w2=0,Z w3=1);
   // cosmetics the image using functions "GetW, SignalValue,
   // getPixConvPla"
   // the function takes color away from a multimage depending on the
   // curve of funktion GetW
   // the paramaters are only for then funktion "GetW"

Z SignalValue(Z Color,Z Lin,Z Col,Z Meth);
   // get a signal value
   // Color:   Number of colors
   // Lin:      Number of lins
   // Col:      Number of columns
   // Meth:      Method, how to get the value
   //            0 = MAX(R,G,B), 1 = 1/3(R+G+B)

R GetW(Z signal,Z s1,Z s2,Z s3,Z s4,Z w1,Z w2,Z w3);
   // Get a value W which depends on signal and this curve:
   //    W1______
   //            \                 ____W3
   //             \              /
   //              \            /
   //               \__________/W2
   //      0    S1  S2        S3 S4  255
```

```
   Histogram histogram(Z nBins=16,R BDown=0, R BUp=255);
      // makes a histogramm of an Image
      // nBins:   number of Sectors
      // BDown:   minimal value (normaly 0)
      // BUp:      maximal value (normaly 255)

   MultiImage<T> mal2()const;
      // multiplicate all Pixlevalues with 2.3
      // this function was only a test,
      // the better function is "equalization"

   MultiImage<T> equalization(R2 minmax)const;
      // stretches the Image over the whole area of Pixlevalues
      // from 0 to 255
      // minmax includes the area of the unstretched Image

   VV<T> cut(Z h, const Rect& rect)const;
      // creates a rectangular cut-ut of the h'th image plane of
      // *this as a a matrix.
      // See class Rect for details on the description of the
      // rectangular area.
};

typedef MultiImage<R> ImgR;
   // in analogy to Img24

template<class T>
MultiImage<T>::MultiImage(Z p, const Image<Typ2>& im, T (*f)(Z,Typ2)):
   V<Image<T> >(p,Image<T>(im))
{
   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j;
   for (h=1;h<=l;h++) for (i=1;i<=m;i++) for (j=1;j<=n;j++) {
      putPix(h,i,j,f(h,im.getPix(i,j)));
   }
}

#undef Typ2

template <class T>
void MultiImage<T>:: print(void)
{
   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   T t;
   Word w=nameOf(t);
```

```
   cout<<endl<<" This is a MultiImage<> with values of type "<<w;
   cout<<endl<<l<<" planes, "<<m<<" lines, "<<n<<" columns";
}

template <class T>
MultiImage<T>::MultiImage(VV<Img24> const& vv,  Z m, Z n ):
   V<Image<T> >(3,Image<T>(m,n))
{
   Z ny=vv.dim1();
   Z nx=vv.dim2();
   Iv ivx(0.,1.);
   Iv ivy(0.,1.);
   V<Iv> vivx=ivx.divide(nx);
   V<Iv> vivy=ivy.divide(ny);
   Z i,j,scale=1;
   // notice that Rect has a (x,y) understanding of layout,
   // whereas vv is to be understood as matrix layout
   for (i=1;i<=ny;++i){
      for (j=1;j<=nx;++j){
         MultiImage<T> mij(vv[i][j]);
         insert(mij,Rect(vivx[j],vivy[1+ny-i]),scale);
      }
   }
}

template <class T>
MultiImage<T>::MultiImage( const VVV<T>& y):
V<Image<T> >( y.dim1(),Image<T>(y.dim2(), y.dim3()))
{
   Z l=getPla();  // notice initialization
   Z m=getLin();
   Z n=getCol();
   Z h,i,j;
   T val;
   for (h=1;h<=l;h++){
      if (verbose){ cout<<endl<<"MultiImage<>(Tensor3&): plane "<<h;}
      for (i=1;i<=m;i++){
         if (verbose>9){ cout<<endl<<"MultiImage<>(Tensor3&): line "<<i;}
         for (j=1;j<=n;j++){
            val=y(h,i,j);
            putPix(h,i,j,val);
         }
      }
   }
}

template <class T>
MultiImage<T>::MultiImage( const VV<T>& y):
V<Image<T> >(1,Image<T>(y.dim1(), y.dim2()))
{
```

```
   Z m=getLin();
   Z n=getCol();
   Z i,j;
   T val;
   for (i=1;i<=m;i++){
      if (verbose>9){ cout<<endl<<"MultiImage<>(Matrix&): line "<<i;}
      for (j=1;j<=n;j++){
         val=y(i,j);
         putPix(1,i,j,val);
        }
    }
}

template<class T>
MultiImage<T> MultiImage<T>::interpolate(Z mI, Z nI, Z method)const

{
   Z l=getPla(),h;
   MultiImage<T> res(l); // trivial multi-image with l components
   // redefining the components.
   for (h=1;h<=l;h++){
      res[h]=((*this)[h]).interpolate( mI, nI, method);
   }
   return res;
}

template<class T>
R2 MultiImage<T>::minMax(R xL, R xU, R yL, R yU)const
{
   Z l=getPla(),h;
   R_Vector valMax(l);
   R_Vector valMin(l);
   R2 lim;
   for (h=1;h<=l;h++){
      lim=((*this)[h]).minMax(xL, xU, yL, yU);
      valMin[h]=lim.x1;
      valMax[h]=lim.x2;
   }
   R vmax=valMax.sup();
   R vmin=valMin.inf();
   R2 res(vmin,vmax);
   return res;
}

template<class T>
VVV<T> MultiImage<T>::toVVV()const
{
   Z l=getPla();
   Z m=getLin();
```

```
   Z n=getCol();
   VVV<T> res(l,m,n);
   Z h,i,j;
   for (h=1;h<=l;h++) for (i=1;i<=m;i++) for (j=1;j<=n;j++)
      res[h][i][j]=((*this)[h])[i][j];

   return res;
}

template<class T>
void MultiImage<T>::mark(Graph& gr, R2 lim, Z method, Z faithful)const
{
   Z mL=2;
   Word loc("MultiImage<>::mark(Graph,R2,Z,Z,Z)");
   CPM_MA
   Z l=getPla();
   if (l==1){
      cpmassert(method<=1,loc);
      if (method>1) method=1;
      ((*this)[1]).mark(gr,lim,method,faithful);
      CPM_MZ
      return;
   }
   if (l==2){
      cpmwarning(loc&": 2 planes not supported");
      CPM_MZ
      return;
   }
   Z m=getLin(), n=getCol();
   Z h,i,j;
   R width=1., hight=1.;
   if (faithful){ // we assume square pixels on screen
      R alpha_=((*this)[1]).getAspRat(); // aspect ratio of image
      R alpha=gr.getAspRat(); // aspect ratio of frame
      if (alpha_ > alpha){ // image has to be placed full width
         width=1.;
         hight=alpha/alpha_;
      }
      else{ // image has to be placed full hight
         width=alpha_/alpha;
         hight=1.;
      }
   }
   V<R> lattice1=ImagingTools::pixelCenters(0.,width,n);
   V<R> lattice2=ImagingTools::pixelCenters(0.,hight,m);
   R lx1=lim.x1;
   R lx2=lim.x2;
   R ga=Color::getGamma();
   if (method<=1){
      Z2 index;
```

```
   R2 p;
   Z color;
   R val, pI, qI;
   Z left,right,lower,upper;
   for (i=1;i<=gr.getLin();i++){
      index.x1=i;
      for (j=1;j<=gr.getCol();j++){
         index.x2=j;
         p=gr.getxFyF(index); // point in frame system
         if (p.x2>hight || p.x1>width) continue;
         ImagingTools::find(lattice1, p.x1, pI, left, right);
         ImagingTools::find(lattice2, p.x2, qI, lower, upper);
         // left ... upper refer to discretized xF, yF
         lower=1+m-lower;
         upper=1+m-upper;
         // now left ... upper refer to the matrix index grid
         // pixel values are converted to R for (optional)
         // interpolation and input to ImagingTools::response
         R a11,a12,a21,a22;
         CpmRootX::Z1 rVal,gVal,bVal;
         for (h=1;h<=3;h++){
            a11=getPix(h,upper,left);
            a12=getPix(h,upper,right);
            a21=getPix(h,lower,left);
            a22=getPix(h,lower,right);

            val=ImagingTools::int2(a11,a12,a21,a22,pI,qI,method);
            color=(Z)response(lx1,lx2,0,255, ga,val);
            switch(h){
               case 1:
                  rVal=color; break;
               case 2:
                  gVal=color; break;
               case 3:
                  bVal=color; break;
               default:
               ;
            }

         }   // h loop finished
         rgb c(rVal,gVal,bVal);
         gr.put(index,Color(c));
      }
   }
}
else if (method>1 && l==3){
   Z nGrid=m*n, k=1;
   V<R2> x(nGrid);
   V<R3> y(nGrid);
   for (i=1;i<=m;++i){
```

```
         Z iy=m-i+1;
         R ly=lattice2[iy];   // 2 <--> y
         for (j=1;j<=n;++j){
            R lx=lattice1[j]; // 1 <--> x
            x[k]=R2(lx,ly);
            // this is the position to which the pixel value
            // getPix(*,i,j); belongs, where position is what it
            // is for function Graph::show(F<R2,R3> const&,...)
            R3 val;
            Z ix=i;
            R v1=getPix(1,i,j);
            R v2=getPix(2,i,j);
            R v3=getPix(3,i,j);
            val[1]=ImagingTools::response(lx1,lx2,0,255,ga,v1);
            val[2]=ImagingTools::response(lx1,lx2,0,255,ga,v2);
            val[3]=ImagingTools::response(lx1,lx2,0,255,ga,v3);
            y[k++]=val;
         }
      } // these two loops arn't the bottle neck.
      CpmGeo::DisField<R2,R3>::setGrid(x);
      CpmGeo::DisField<R2,R3> df;
      df.setY(y);
         // now df knows x and y
      F<R2,R3> fc=df.contField();
         // for this kind of object we have a show-method in class
         // Graph. Each evaluation of fc is hard work even for small
         // images. So this is only an experimental method.
      bool noDisplay=true;
      gr.show(fc,0,width,0,hight,noDisplay);
   }
   else{
      cpmwarning(loc&": invalid argument method");
   }
   CPM_MZ
}

static R limxfT;
static R limyfT;
static R rLfT;
static R rUfT;

static Z respfT(R x)
{
   static Z first=1;
   R dx=limyfT-limxfT;
   if (dx==0){
      if (first){
         cpmmessage("MultiImage<T>::setRange(): degenerate range");
         first=0;
      }
```

```
      dx=1;
   }
   R y=(x-limxfT)/dx;
   R y1=rLfT+y*(rUfT-rLfT);
   y1*=255;
   y1+=0.5;
   if (y1>255) return 255;
   if (y1<0) return 0;
   return Z(y1);
}

template <class T>
void MultiImage<T>::setRange(T valMin, T valMax)
{
   R2 lim=minMax();
   R ymin=lim[1];
   R ymax=lim[2];
   R d=(ymax-ymin);
   assert(d>0.);

   R vmin=valMin;
   R vmax=valMax;

    R f=(vmax-vmin)/d;

   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j;
   R val;
    T valt;
   for (h=1;h<=l;h++){
      if (verbose){ cout<<endl<<"setRange(): treating pane "<<h;}
      for (i=1;i<=m;i++){
      if (verbose>9){ cout<<endl<<"setRange(): treating line "<<i;}
         for (j=1;j<=n;j++){
            val=getPix(h,i,j);   // conversion to R
            valt=vmin+(val-ymin)*f;
            putPix(h,i,j,valt);
         }
      }
   }
}

template <class T>
MultiImage<T>::MultiImage(const Img24& img)
{
   cpmmessage("MultiImage(Img24) started");
   Z m=img.getLin();
   Z n=img.getCol();
```

```
   MultiImage<T> res(3,m,n);
   Z i,j;
   R prfrac=1./m;
   for (i=1;i<=m;i++){
      cpmprogress(" putPix()-loop in constructor",i*prfrac);
      for (j=1;j<=n;j++){
         rgb pixel=img.getPel(j-1,i-1);
         res.putPix(1,i,j,(L)pixel.r);
         res.putPix(2,i,j,(L)pixel.g);
         res.putPix(3,i,j,(L)pixel.b);
      }
   }
   *this=res;
   cpmmessage("MultiImage(Img24) done");
}

template <class T>
MultiImage<T>::MultiImage(const Frame& fr)
{
   *this=MultiImage<T>(fr.getImg());
}

template<class T>
MultiImage<T>& MultiImage<T>::operator *=(const T& t)
{

   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j;
   if (verbose){ cout<<endl<<"Multiplication of image started";}
   for (h=1;h<=l;h++){
      if (verbose){ cout<<endl<<"treating pane "<<h;}
      for (i=1;i<=m;i++){
         if (verbose>9){ cout<<endl<<"treating line "<<i;}
         for (j=1;j<=n;j++){
            putPix(h, i, j, t*getPix(h,i,j) );
         }
      }
   }
   return *this;
}

template<class T>
MultiImage<T>& MultiImage<T>::operator *=(const Image<T>& im)
{

   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
```

```
   Z h,i,j;
   assert(m==im.getLin());
   assert(n==im.getCol());
   if (verbose){ cout<<endl<<"Multiplication of image started";}
   for (h=1;h<=l;h++){
      if (verbose){ cout<<endl<<"treating pane "<<h;}
      for (i=1;i<=m;i++){
         if (verbose>9){ cout<<endl<<"treating line "<<i;}
         for (j=1;j<=n;j++){
            putPix(h, i, j, getPix(h,i,j)*im.getPix(i,j) );
         }
      }
   }
   return *this;
}

template<class T>
MultiImage<T>& MultiImage<T>::operator *=(const MultiImage<T>& im)
{

   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j;
   assert(l==im.getPla());
   assert(m==im.getLin());
   assert(n==im.getCol());
   if (verbose){ cout<<endl<<"Multiplication of image started";}
   for (h=1;h<=l;h++){
      if (verbose){ cout<<endl<<"treating pane "<<h;}
      for (i=1;i<=m;i++){
         if (verbose>9){ cout<<endl<<"treating line "<<i;}
         for (j=1;j<=n;j++){
            putPix(h, i, j, getPix(h,i,j)*im.getPix(h,i,j) );
         }
      }
   }
   return *this;
}

template<class T>
MultiImage<T>& MultiImage<T>::operator /=(const Image<T>& im)
{

   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j;

   assert(m==im.getLin());
```

```
   assert(n==im.getCol());
   T val;
   if (verbose){ cout<<endl<<"Division of image started";}
   for (h=1;h<=l;h++){
      if (verbose){ cout<<endl<<"treating pane "<<h;}
      for (i=1;i<=m;i++){
         if (verbose>9){ cout<<endl<<"treating line "<<i;}
         for (j=1;j<=n;j++){
            val=im.getPix(i,j);
            if (val==0){
               putPix(h, i, j, 0 );
            }
            else{
               putPix(h, i, j, getPix(h,i,j)/val );
            }
         }
      }
   }
   return *this;
}

template<class T>
MultiImage<T>& MultiImage<T>::operator /=(const MultiImage<T>& im)
{

   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j;
   assert(m==im.getLin());
   assert(n==im.getCol());
   T val;
   if (verbose){ cout<<endl<<"Division of image started";}
   for (h=1;h<=l;h++){
      if (verbose){ cout<<endl<<"treating pane "<<h;}
      for (i=1;i<=m;i++){
         if (verbose>9){ cout<<endl<<"treating line "<<i;}
         for (j=1;j<=n;j++){
            val=im.getPix(h,i,j);
            if (val==0){
               putPix(h, i, j, 0 );
            }
            else{
               putPix(h, i, j, getPix(h,i,j)/val );
            }
         }
      }
   }
   if (verbose){ cout<<endl<<"Division of image finished";}
   return *this;
```

```
}

template<class T>
MultiImage<T> MultiImage<T>::cut(R xL, R xU, R yL, R yU)const
{
   Z l=getPla();
   MultiImage<T> res(l);
   for (Z i=1;i<=l;i++){
      res[i]=((*this)[i]).cut(xL,xU,yL,yU);
   }
   return res;
}

template<class T>
MultiImage<T> MultiImage<T>::cut(Z2 cLL, Z2 cUR)const
{
   Z l=getPla();
   MultiImage<T> res(l);
   for (Z i=1;i<=l;i++){
      res[i]=((*this)[i]).cut(cLL,cUR);
   }
   return res;
}

template<class T>
MultiImage<T>& MultiImage<T>::operator -=(const MultiImage<T>& im)
   // returns *this-im, the pixelwise difference of images.
   // *this and im are assumed to coincide in numbers of planes, lines,
   // and columns
{
   Word loc("MultiImage<T>::operator -=");
   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j;
   cpmassert(l==im.getPla(),loc);
   if (verbose){ cout<<endl<<"Subtraction of image started";}
   for (h=1;h<=l;h++){
      if (verbose){ cout<<endl<<"treating pane "<<h;}
      for (i=1;i<=m;i++){
         if (verbose>9){ cout<<endl<<"treating line "<<i;}
         for (j=1;j<=n;j++){
            putPix(h,i,j, getPix(h,i,j)-im.getPix(h,i,j) );
         }
      }
   }
   return *this;

}
```

```
template<class T>
MultiImage<T>& MultiImage<T>::apply(const MultiImage<T>& im,
   const F<X2<T,T>,T>& f)
{
   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j;
   assert(l==im.getPla());
   assert(m==im.getLin());
   assert(n==im.getCol());
   if (verbose){ cout<<endl<<"apply() of image started";}
   for (h=1;h<=l;h++){
      if (verbose){ cout<<endl<<"treating pane "<<h;}
      for (i=1;i<=m;i++){
         if (verbose>9){ cout<<endl<<"treating line "<<i;}
         for (j=1;j<=n;j++){
            X2<T,T> arg( getPix(h,i,j), im.getPix(h,i,j) );
            putPix(h, i, j, f(arg) );
         }
      }
   }
   return *this;
}

template<class T>
MultiImage<T>& MultiImage<T>::apply(const MultiImage<T>& im,
   const V<F<X2<T,T>,T> >& f)
{
   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j;
   assert(l==im.getPla());
   assert(m==im.getLin());
   assert(n==im.getCol());
   if (verbose){ cout<<endl<<"apply() of image started";}
   for (h=1;h<=l;h++){
      if (verbose){ cout<<endl<<"treating pane "<<h;}
      for (i=1;i<=m;i++){
         if (verbose>9){ cout<<endl<<"treating line "<<i;}
         for (j=1;j<=n;j++){
            X2<T,T> arg( getPix(h,i,j), im.getPix(h,i,j) );
            putPix(h, i, j, (f[h])(arg) );
         }
      }
   }
   return *this;
}
```

```
template<class T>
R MultiImage<T>::norm1(void)const
   // sum over the absolute values over all pixel values, divided by
   // number of pixels
{
   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j;
   R sum=0.;   // even if T is a integer type !
   R val;
   if (verbose){ cout<<endl<<"MultiImage<>::norm1 started";}
   for (h=1;h<=l;h++){
      if (verbose){ cout<<endl<<"treating pane "<<h;}
      for (i=1;i<=m;i++){
         if (verbose>9){ cout<<endl<<"treating line "<<i;}
         for (j=1;j<=n;j++){
            val=getPix(h,i,j);
            R valr=CpmRoot::absVal<T>(val);
            sum+=valr;
         }
      }
   }
   R nPix=1.*l*m*n;
   assert(nPix>0);
   sum/=nPix;
   return sum;
}

template<class T>
R MultiImage<T>::norm2(void)const
   // sum over the squares of all pixel values, divided by number of
   // pixels.
   // Square root of all of that. (Normalization is such, that the norm
   // of
   // a constant image is equal to the pixel value)
{
   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j;
   R sum=0., val;
   if (verbose){ cout<<endl<<"MultiImage<>::norm2 started";}
   for (h=1;h<=l;h++){
      if (verbose){ cout<<endl<<"treating pane "<<h;}
      for (i=1;i<=m;i++){
         if (verbose>9){ cout<<endl<<"treating line "<<i;}
         for (j=1;j<=n;j++){
            val=getPix(h,i,j);
            sum+=val*val;
```

```
         }
      }
   }
   R nPix=1.*l*m*n;
   assert(nPix>0);
   sum/=nPix;
   assert(sum>=0);
   sum=sqrt(sum);
   return sum;
}

template<class T>
void MultiImage<T>::gamma(R gammaI)
{
   Z l=getPla(),h;
   R2 b=minMax();
   for (h=1;h<=l;h++){
      ((*this)[l]).gamma(b.x1,b.x2,b.x1,b.x2,gammaI);
         // same transformation to all planes
   }
}

template<class T>
X2<R_Vector,Z2> MultiImage<T>::hotspot(Z di, Z dj)
{
   R hu=1e12;

   R valMax=-hu;
   Z iMax,jMax;

   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j;

   R val;
   for (i=1;i<=m;i++){
      for (j=1;j<=n;j++){
         val=0.;
         for (h=1;h<=l;h++){
            val+=getPix(h,i,j);
               // sum over all channels defines total brightness
         }
         if (val>valMax){
            valMax=val;
            iMax=i;
            jMax=j;
         }
      }
   }
```

```
   R normfac=(1+2*di)*(1+2*dj);
      // number of terms in summation
   normfac=1./normfac;
   R_Vector res1(l);
   Z si,sj;
   Z Di=di;
   Z Dj=dj;
   for (h=1;h<=l;h++){
      val=0.;
      for (si=-Di;si<=Di;si++){ // does not work for di,dj instead of
      // Di,Dj
         for (sj=-Dj;sj<=Dj;sj++){
            val+=((*this)[h]).getPixCont(iMax+si,jMax+sj);
               // no problem with leaving the boundary due to getPixCon
         }
      }
      res1[h]=val*normfac;
   }
   Z2 res2(iMax,jMax);
   return X2<R_Vector,Z2>(res1,res2);
}

template<class T>
Image<T> MultiImage<T>::combine( R (*f)(const R_Vector&) )const
{
   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j;
   R_Vector x(l);
   Image<T> res(m,n);
   for (i=1;i<=m;i++){
      for (j=1;j<=n;j++){
         for (h=1;h<=l;h++){
            x[h]=getPix(h,i,j);
         }
         res.putPix(i,j,f(x));
      }
   }
   return res;
}

template<class T>
MultiImage<T> MultiImage<T>::conv(const R_Matrix& a, Z si, Z sj)
{
   Z l=getPla();
   MultiImage<T> res(l);
   for (Z h=1;h<=l;h++) res[h]=((*this)[h]).conv(a,si,sj);
   return res;
}
```

```
static R gauss(R r2, R sigma)
   // 2-dimensional radially symmetric Gauss distribution as a function
   // of r*r (in our particular case r*r is easier to calculate than r)
{
   const static R PiInv=1./Pi;
   cpmassert(sigma>0.,"gauss(R,R)");
   R factorArg=1./(sigma*sigma*2.);
   R factor=sqrt(factorArg*PiInv);
   return factor*exp(-r2*factorArg);
}

template<class T>
MultiImage<T> MultiImage<T>::conv(const R& sigma, Z dL)
{
   static const R t=1./sqrt(3.);
   // see Abramowitz Stegun p. 892, second formula of 25.4.62
   // for the numerical integration formula to be used here
   // calculation of the convolution kernel as a matrix
   static const R tiny=1e-3;
   if (sigma<tiny) return *this;
      // no convolution required
   Z d=2*sigma;
   if (d>dL) d=dL;
   Z n=2*d+1;
   R_Matrix a(n,n);
   Z i,j;
   R x, x0=0.5*n, x1;
   R y, y0=x0, y1;
   R r2, w, sum=0.;
   for (i=1;i<=n;i++){
      x=i-0.5-x0;
      for (j=1;j<=n;j++){
         // we calculate the weight of a pixel by numerical
         // integration using a 4-point formula ( Abramowitz Stegun
         // as cited above)
         y=j-0.5-y0;
         x1=x+t;
         y1=y+t;
         r2=x1*x1+y1*y1;
         w=gauss(r2,sigma);

         x1=x+t;
         y1=y-t;
         r2=x1*x1+y1*y1;
         w+=gauss(r2,sigma);

         x1=x-t;
         y1=y+t;
         r2=x1*x1+y1*y1;
```

```
         w+=gauss(r2,sigma);

         x1=x-t;
         y1=y-t;
         r2=x1*x1+y1*y1;
         w+=gauss(r2,sigma);
         w*=0.25;
         sum+=w;
         a[i][j]=w;
      }
   }

   sum=1./sum;
   a*=sum;   // normalizing to sum over all pixels = 1

   return conv(a);
 }

template<class T>
MultiImage<T> MultiImage<T>::conv(const V<Mask>& a)const
{
   Z l=getPla();
   Z d=a.dim();
   assert(d==l);
   MultiImage<T> res(l);
   for (Z h=1;h<=l;h++) res[h]=((*this)[h]).conv(a[h]);
   return res;
}

template<class T>
MultiImage<T> MultiImage<T>::cut(const Rect& a)const
{
   Z l=getPla();
   MultiImage<T> res(l);
   for (Z h=1;h<=l;h++) res[h]=((*this)[h]).cut(a);
   return res;
}

template<class T>
MultiImage<T> MultiImage<T>::conv(const Mask& a)const
{
   Z l=getPla();
   MultiImage<T> res(l);
   for (Z h=1;h<=l;h++) res[h]=((*this)[h]).conv(a);
   return res;
}

template<class T>
MultiImage<T> MultiImage<T>::clip(R clipL, R clipU)const
{
```

```
   Z l=getPla();
   MultiImage<T> res(l);
   for (Z h=1;h<=l;h++) res[h]=((*this)[h]).clip(clipL, clipU);
   return res;
}

static Z part(Z m, const R& x)
{
   Z res=(Z)(1.+m*x);
   return (res>m ? m : res);
}

template<class T>
void MultiImage<T>::insertPattern(
   R X, R Y, R dX,   const F<C,Va<T> >& f, const F<C,B>& g,
   R x, R y, R dx, R dy, B average )
      // as previous function, but pixels are set only if g(z)=trueUM
      // otherwise pixel value is not changed
{
   const Z nTacit=4;
      // controls frequency of messages in verbose mode

   R ar=((*this)[1]).getAspRat();
   assert(ar!=0.);
   R dY=dX/ar;

   R XU=X+dX;
   R YU=Y+dY;

   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j,i_;

   R u,v;
   R du=dX/(4.*n);
   V<R> iLattice=ImagingTools::pixelCenters(Y,YU,m);
      // matrix index i varies in y direction
   V<R> jLattice=ImagingTools::pixelCenters(X,XU,n);
      // matrix index j varies in x direction

   C z,w;
//   R r;
   Z lineCounter=0;
   R mInv=1./m;
   B valid;

   Z iU=m-part(m,y)+1;
   Z iL=m-part(m,y+dy)+1;
   Z jL=part(n,x);
```

```
Z jU=part(n,x+dx);
Va<R> sum(l);
Va<T> fval;
T val;
Z count;

for (i=iL;i<=iU;i++){
   i_=m-i+1;
   v=iLattice[i_];
   if (verbose==1)
   {
      lineCounter++;
      if (lineCounter==nTacit){
         lineCounter=0;
         R fr=100.*i*mInv;
         cout<<endl<<"Image "<<fr<<" % completed";
      }
   }

   for (j=jL;j<=jU;j++){
      u=jLattice[j];
      if (average){
         sum.set_(0);
         count=0;
         for (Z eu=-1;eu<=1;eu+=2)for (Z ev=-1;ev<=1;ev+=2){
            z=C(u+eu*du,v+ev*du);
            valid=g(z);
            if (valid){
               count++;
               fval=f(z); // transformation C to Va<T,Z>
               for (h=1; h<=l; h++){
                  sum[h]+=fval[h];
               }
            }
         }
         if (count>0){
            for (h=1; h<=l; h++){
               val=sum[h]/count;
               putPix(h,i,j,val);
            }
         }
      }
      else{
         z=C(u,v);
         valid=g(z);
         if (valid){
            fval=f(z);
            for (h=1; h<=l; h++){
               putPix(h,i,j,fval[h]);
            }
```

```
            }
         }

      }  // j loop
   }   // i loop
}

template<class T>
MultiImage<T> MultiImage<T>::fourierFilter(const Fa<R2,R>& resp)const
{
   Z mL=1;
   static Word loc("MultiImage<>::fourierFilter(Fa<R2,R>)");
   CPM_MA
   Z l=getPla();
   MultiImage<T> res(l);
   for (Z h=1;h<=l;h++){
      res[h]=((*this)[h]).fourierFilter(resp);
      cpmstatus("plane "&cpm(h)&" filtered",2);
   }
   if (verbose>10){
      Frame f;
      res.mark(f,1,1);
   }
   CPM_MZ
   return res;
}

template<class T>
MultiImage<T> MultiImage<T>::fftFilter(const F<C,C>& resp)const
{
   Z mL=1;
   static Word loc("MultiImage<>::fftFilter(F<C,C>)");
   CPM_MA
   Z l=getPla();
   MultiImage<T> res(l);
   for (Z h=1;h<=l;h++){
      res[h]=((*this)[h]).fftFilter(resp);
      cpmstatus("plane "&cpm(h)&" filtered",2);
   }
   if (verbose>10){
      Frame f;
      res.mark(f,1,1);
      f.display();
   }
   CPM_MZ
   return res;
}

template<class T>
MultiImage<T> MultiImage<T>::deScreen(R lim, R rim, Z p)const
```

```
{
   Z mL=1;
   static Word loc("MultiImage<>::deScreen()");
   CPM_MA
   Z l=getPla(),i,j,k;
   V< VV<C> > sp(l);
   sp[1]=((*this)[1]).fourierTransform();
   Z m=sp[1].dim1(), n=sp[1].dim2();
   Image<R> mono(m,n,0.);
   for (i=1;i<=m;++i) for (j=1;j<=n;++j)
      mono[i][j]=(sp[1][i][j]).absVal();
   for (k=2;k<=l;++k){
      sp[k]=((*this)[k]).fourierTransform();
      for (i=1;i<=m;++i) for (j=1;j<=n;++j)
         mono[i][j]+=(sp[k][i][j]).absVal();
   }
   Z nMask=2*p+1;
   V<R> w(nMask,1./nMask);
   V<Z> da(nMask,0);
   V<Z> db(nMask);
   Z p_=-p;
   for (i=1;i<=nMask;++i){
      db[i]=p_;
      p_++;
   }
   Mask mas1(da,db,w);
   Mask mas2(db,da,w);
   Image<R> mono1=mono.conv(mas1);
   Image<R> mono2=mono.conv(mas2);
   R gamma=0.25;
   VV<R> fac(m,n,1);
   Z ml=cpmround(m*rim);
   Z mu=m-ml;
   Z nl=cpmround(n*rim);
   Z nu=n-nl;
   R red=0.33;
   for (i=ml;i<=mu;++i) for (j=nl;j<=nu;++j){
      R v0=mono[i][j];
      R v1=mono1[i][j];
      R v2=mono2[i][j];
      R d1=v0-v1;
      R d2=v0-v2;
      R dr=lim*0.5*(v1+v2);
      if (d1>dr) fac[i][j]*=red;
      if (d2>dr) fac[i][j]*=red;
   }
   for (i=1;i<=m;++i) for (j=1;j<=n;++j){
      R fij=fac[i][j];
      for (k=1;k<=l;++k) sp[k][i][j]*=fac[i][j];
   }
```

```
   MultiImage<T> res(l);
   for (k=1;k<=l;++k){
      res[k]=Image<R>(sp[k]);
   }
   CPM_MZ
   return res;
}

template<class T>
R MultiImage<T>::getPixConv(Z h, Z i, Z j, const R_Matrix& a, Z si, Z sj)
{
   Z ma=a.dim1(),na=a.dim2(), ia, ja;
   Z ci=1+ma/2, cj=1+na/2;
   Z i0=ci+si, j0=cj+sj;
   R val=0.;

   for (ia=1;ia<=ma;ia++){
      for (ja=1;ja<=na;ja++){
            val+=(R)((*this)[h]).getPixCont(i+ia-i0,j+ja-j0)*a[ia][ja];
      }
   }
   return val;
}

template<class T>
R_Vector MultiImage<T>::getPixConvPla( Z i, Z j, const R_Matrix& a)
{
   Z l=getPla();
   Z h_;
   R_Vector res(l);

   for (h_=1;h_<=l;h_++)
   {
      res[h_]=getPix(h_,i,j);
   }
 //   res=a*res;
   return a*res;
}

template<class T>
MultiImage<T> MultiImage<T>::cosmetic(Z Meth,Z s1,Z s2,Z s3,Z s4,Z w1,
   Z w2,Z w3)
{
    cpmmessage("Function cosmetic");

    Z l=getPla();
    Z m=getLin();
    Z n=getCol();

    Z h,i,j;
```

```
    Z signal;
    R W;
    MultiImage<T> res(l,m,n);
    R val;
    Z dm=3;
    R_Matrix a(dm,dm);
    R vm=1./(dm*dm);
    for(i=1;i<=dm;i++) for(j=1;j<=dm;j++) a[i][j]=vm;

    for(i=1;i<=m;i++)
    {
      for(j=1;j<=n;j++)
      {
            signal=SignalValue(l,i,j,Meth);
            W=GetW(signal,s1,s2,s3,s4,w1,w2,w3);
            for (h=1;h<=l;h++)
               {
               val=(1-W)*getPix(h,i,j)+W*getPixConv(h,i,j,a,0,0);
               res.putPix(h,i,j,val);
               }
      }
    }
    return res;
}

template<class T>
MultiImage<T> MultiImage<T>::cosmetic2(Z Meth,Z s1,Z s2,Z s3,Z s4,
   Z w1,Z w2,Z w3)
{
    cpmmessage("Function cosmetic2");

    Z l=getPla();
    Z m=getLin();
    Z n=getCol();

    Z h,i,j;
    Z signal;
    R W;
    MultiImage<T> res(l,m,n);
    R val;
    R_Matrix a(l,l);
    R vm=1./l;
    for(i=1;i<=l;i++) for(j=1;j<=l;j++) a[i][j]=vm;

    for(i=1;i<=m;i++)
    {
      for(j=1;j<=n;j++)
      {
            signal=SignalValue(l,i,j,Meth);
            W=GetW(signal,s1,s2,s3,s4,w1,w2,w3);
```

```
            R_Vector temp;
            temp=getPixConvPla(i,j,a);
            for (h=1;h<=l;h++)
               {

               val=(1-W)*getPix(h,i,j)+W*temp[h];
               res.putPix(h,i,j,val);
               }
      }
    }
    return res;
}

template<class T>
Z MultiImage<T>::SignalValue(Z Color,Z Lin,Z Col,Z Meth)
{

   Z i,value;
   Z NewValue=0;
   for(i=1;i<=Color;i++)
   {
      value=getPix(i,Lin,Col);
      if (Meth == 1) NewValue+=0.33 * value;
      else
         if(value > NewValue) NewValue = value;
   }
   return NewValue;
}

template<class T>
R MultiImage<T>::GetW(Z signal,Z s1_,Z s2_,Z s3_,Z s4_,Z w1_,Z w2_,Z w3_)
{

   R NewW;
   R s1=s1_;
   R s2=s2_;
   R s3=s3_;
   R s4=s4_;
   R w1=w1_;
   R w2=w2_;
   R w3=w3_;

   if (signal < s1)
         NewW = w1;
   if ((signal >= s1) && (signal < s2))
         NewW = w1+(w2-w1)/(s2-s1)*(signal-s1);
   if ((signal >= s2) && (signal < s3))
         NewW = w2;
   if ((signal >= s3) && (signal < s4))
         NewW = w2+(w3-w2)/(s4-s3)*(signal-s3);
```

```
   if (signal > s4)
         NewW = w3;

return NewW;
}

template<class T>
MultiImage<T> MultiImage<T>::mal2()const
{
   cpmmessage("Funktion mal2");

   R fac=2.3;

   Z l=getPla();
   Z m=getLin();
   Z n=getCol();
   Z h,i,j,Val;

   MultiImage<T> a(l,m,n);
   for(i=1;i<=m;i++)
   {
      for(j=1;j<=n;j++)
      {
         for(h=1;h<=l;h++)
         {
            Val=getPix(h,i,j);
            Val*=fac;
            if(Val<=0) Val=1;
            if(Val>255) Val=255;
            a.putPix(h,i,j,Val);
         }
      }
   }
   return a;
}

template<class T>
Histogram MultiImage<T>::histogram(Z nBins, R BDown, R BUp )

{
   cpmmessage("Function histogram");

   Z l=getPla();
   Z m=getLin();
   Z n=getCol();

   Z h,i,j;
   R Val;
   Histogram b(l,nBins,BDown,BUp);
   for(i=1;i<=m;i++)
```

```
   {
      for(j=1;j<=n;j++)
      {
         for(h=1;h<=l;h++)
         {
            Val=getPix(h,i,j);
            b.put(h,Val);
         }
      }
   }

   return b;
}

template<class T>
MultiImage<T> MultiImage<T>::equalization(R2 minmax)const
{
    cpmmessage("Function equalization");

    R newUp=255;
    R newDown=0;
    R oldUp=minmax.x1;
    R oldDown=minmax.x2;
    R old,New;

    Z l=getPla();
    Z m=getLin();
    Z n=getCol();

    Z h,i,j;
    MultiImage<T> res(l,m,n);
    assert(oldUp>oldDown);
    R b=(newUp-newDown)/(oldUp-oldDown);
    R a=newDown-b*oldDown;

    for(i=1;i<=m;i++){
      for(j=1;j<=n;j++){
         for(h=1;h<=l;h++){
            old=getPix(h,i,j);
            New=a+b*old;
            if (New<newDown) New=newDown;
            if (New>newUp) New=newUp;
            res.putPix(h,i,j,New);
         }
      }
   }
   return res;
}

template<class T>
```

```
VV<T> MultiImage<T>::cut(Z h, const Rect& rect)const
{
   Z l=getPla();
   assert(h>=1 && h<=l);
   return ((*this)[h]).cut(rect);
}

template<class T>
void MultiImage<T>::insert(MultiImage<T> const& img,
   Rect const& a, Z scale)
{
   Z mL=4;
   Word loc("MultiImage<T>::insert(MultiImage<T>,...)");
   CPM_MA
   const Z method=1;
      // use bilinear interpolation when scaling
//   cpmmessage(mL,loc&" started");
   Z l=getPla();
   cpmassert(l==img.getPla(),loc);
   Z m=getLin();
   Z n=getCol();
   Z mi=img.getLin();
   Z ni=img.getCol();
   R xL=a.getxL();
   R xU=a.getxU();
   R yL=a.getyL();
   R yU=a.getyU();
   Roi ra(m,n,xL,xU,yL,yU);
   Z ma=ra.m();
   Z na=ra.n();
   Z h,i,j;
   if (scale==0){
      Z m1=CpmRootX::inf(mi,ma);
      Z n1=CpmRootX::inf(ni,na);
      for (h=1;h<=l;h++){
         for (i=1;i<=m1;i++){
            for (j=1;j<=n1;j++){
               putPix(h,ra.i(i),ra.j(j),img.getPix(h,i,j));
              }
           }
      }
   }
   else{
      R ara=((R)na)/ma;
      R ari=((R)ni)/mi;
      cpmassert(ara>0,loc);
      cpmassert(ari>0,loc);
      Z niNew, miNew;
      if (ara<=ari){   // then img is 'more landscape' than a
         // and thus ni has to be adjusted to na
```

```
         niNew=(Z)na;
         miNew=(Z)(niNew/ari);
      }
      else{   // then img is 'more portrait' than a
         // and thus mi has to be adjusted to ma
         miNew=(Z)ma;
         niNew=(Z)(miNew*ari);
      }
      MultiImage<T> imgNew=img.interpolate(miNew,niNew,method);
      Z m2=CpmRootX::inf(miNew,ma);
      Z n2=CpmRootX::inf(niNew,na);
      for (h=1;h<=l;h++){
         for (i=1;i<=m2;i++){
            for (j=1;j<=n2;j++){
               putPix(h,ra.i(i),ra.j(j),imgNew.getPix(h,i,j));
            }
         }
      }
   }
   CPM_MZ
}

template<class T>
T MultiImage<T>::getPix(Z i, Z j )const
{
   Z l=getPla();
   R res=0.;
   for (Z h=1;h<=l;h++) res+=getPix(h,i,j);
   res/=l;
   return res;
}

template<class T>
T MultiImage<T>::operator()(R x, R y, Z method)const
{
   Z l=getPla();
   R res=0.;
   for (Z h=1;h<=l;h++) res+=operator()(h,x,y,method);
   res/=l;
   return res;
}

} // namespace

#endif
```

124 cpmnr.h

```
//? cpmnr.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_NR_H_
#define CPM_NR_H_
/*
   Description: Gives a definition of natural numbers based on
   recursion. This is closer to usefulness than my first
   implementation in class NatNum (see cpmmathfound.h)
*/

#include <string>
#include <iostream>

namespace CpmMathFound{ // foundation of mathematics

   using std::string;
   using std::ostream;

class Nr{ // natural numbers, based on recursion
   // Here 0 is not a member of Nr (actually 0 is the first
   // addition of human abstraction and imagination to the
   // 'really natural' system of positive numbers).
   // Implementing numbers with more digits than 0 and 1
   // (e.g. 0,1,2,3,4,5,6,7,8,9) can easily be done along
   // the same lines. However, then a slight complication
   // arrises from the fact that then multiplication of single-
   // digit numbers can't be represented without 'carry'.
   // Notice that using a single allowed digit ('stone age coding')
   // would make the definition of all the functions here
   // a bit simpler. However, this representation needs such ammounts
   // of 'paperspace' for large numbers that it is not of
   // practial value.
   // When starting this project 4 days ago, I intended only to
   // make sure that normal recursive C++ functions (general recursive
   // functions in the sense of mathematics) are sufficient
   // to implement natural number arithmetics. Now it ended up
   // with an implementation that seems to be even practicable
   // in the sense that it mutiplies numbers of 1000 binary
   // digits within seconds.

   string rep;
      // string of symbols \in {0,1} that constitute the 'state'
      // of the quantity (number) and that allows to compute
      // basic functions such as Peano's successor function
      // by an algorithm acting on the state descriptors.
      // This is a much more powerful, and according to
```

```
      // my judgement also more natural approach than the one prefered
      // in mathematical logic, which is to charcterize the individuality
      // of a particular number by the arithmetic (normalized) expression
      // that creates the number.
      // Based on this state description, the basic arithmetic
      // functions +,*,== can be coded as recursions over the length
      // of the state descriptor. This is much faster than the
      // standard recursion over the series of numbers as
      // defined by the successor function. In present recursion
      // scheme the append function app() plays much the same role as
      // the successor function suc() in the 'naive' recursive
      // definition of number arithmetics.

public:

   explicit Nr(string const& s="1");
      // constructor from a string of 1's and 0's where the leading
      // digit has to be a 1 (if s does not respect these rules
      // the call creates an error. Notice that Nr does not contain 0,
      // so 1 is the only distinquished element, which here is
      // created by the default constructor, so a static Nr::unit
      // or Nr::one is not needed.

   explicit Nr(int i);
      // so far definition by counting, not by directly setting rep from
      // the bits of i. So this is very slow for large numbers.
      // Notice that large random numbers are available from function
      // random.

   Nr suc()const;
      // successor function of Peano
      // defined as a recursive function (the implementation code
      // calls suc)

   Nr cut()const;
      // If the string is at least two symbols long we returns the string
      // with the last digit removed. If there is only one symbol
      // we return the string unchanged (i.e. Nr("1").cut()==Nr("1")).

   Nr pre()const;
      // predecessor function
      // defined as a recursive function (the implementation code
      // calls pre). Defined such that Nr("1").pre()==Nr("1").
      // This function is given here for symmetry with respect to
      // function suc(). It is not needed for an recursive definition
      // of + and *. (Needed, however, for the non-recursive definition
      // of *)

   char last()const{ return rep[rep.length()-1];}
      // the corresponding 'first' is not needed to define
```

```
   Nr app(char x)const;
      // appending a digit at the end, error if x is not '0' or '1'

   bool sin()const{ return rep.length()==1;}
      // says whether the string has a single component

   bool operator==(Nr const& n)const;
      // equality defined as a recursive function
      // not making use of the corresponding function defined by
      // the system for string

   bool operator<(Nr const& n)const;
      // relation '<' defined as a recursive function.
      // not making use of the corresponding function defined by
      // the system for string

   Nr operator+(Nr const& n)const;
      // recursive definition, which is much faster than the one based on
      // 'countdown'

   Nr operator*(Nr const& n)const;
      // recursive definition which is much faster than the one
      // based on successive addition

   bool prnOn(ostream&)const;
      // prints the representation string res

   friend ostream& operator<<(ostream& out, Nr const& n)
      { n.prnOn(out); return out;}

   static int test(int z=137);
      // returns the number of errors made, so it has to come out
      // as 0. z gives the maximum number of digits of the random
      // test numbers under consideration

   static Nr random(int z);
      // random binary number with at most z digits, successive calls
      // yield a random sequence (periodicity is that of built-in
      // data type unsigned long int)
};

} // namespace

#endif
```

125 cpmnr.cpp

```
//? cpmnr.cpp
//? Status of work 2008-10-25.
//?
#include <cpmnr.h>
#include <cpmtypes.h>

using CpmMathFound::Nr;
using namespace std;
using namespace CpmRoot;
using namespace CpmSystem;
using namespace CpmRootX;

Nr::Nr(string const& s):rep(s)
{
   Z mL=3;
   cpmdbg=2; // makes the program stop if cpmassert fails
   static Word loc="Nr::Nr(string)";
   CPM_MA
   int l=rep.length();
   cpmassert(l>=1,loc);
   char x=rep[0];
   cpmassert(x=='1',loc);
   for (int i=1;i<l;++i){
      x=rep[i];
      cpmassert(x=='0' || x=='1',loc);
   }
   CPM_MZ
}

Nr::Nr(int i)
{
   Nr aux; // =1
   if (i>1){
      for (int k=1; k<i;++k) aux=aux.suc();
   }
   *this=aux;
}

Nr Nr::app(char x)const
{
   cpmassert(x=='0' || x=='1',"Nr::app(char)");
   string res=rep+x;
   return Nr(res);
}

Nr Nr::cut()const
{
```

```
   if (sin()) return *this;
   int n=rep.length();
   n--;
   string res=rep;
   res.erase(n,1);
   return Nr(res);
}

Nr Nr::suc()const
{
   if (sin()){
      char x=last();
      cpmassert(x=='1',"Nr::suc()");
      return Nr("10");
   }
   else{
      char x=last();
      Nr c=cut();
      if (x=='0') return c.app('1');
      else return c.suc().app('0'); // here recursion takes place
   }
}

Nr Nr::pre()const
{
   if (sin()) return *this;
   else if (rep=="10") return Nr("1");
   else if (rep=="11") return Nr("10");
   else{
      char x=last();
      Nr c=cut();
      if (x=='1') return c.app('0');
      else return c.pre().app('1'); // here recursion takes place
   }
}

bool Nr::operator==(Nr const& n)const
{
   bool b=n.sin();
   if (sin()){
      if (b) return last()==n.last();
      else return false;
   }
   else{
      if (b) return false;
      else return (last()==n.last() && cut()==n.cut());
         // here recursion takes place
   }
}
```

```
bool Nr::operator<(Nr const& n)const
{
   bool b=n.sin();
   if (sin()){
      if (b) return false;
      else return true;
   }
   else{
      if (b) return false;
      else{
         Nr c1=cut();
         Nr c2=n.cut();
         char x1=last();
         char x2=n.last();
         if (c1<c2) return true;
         else if (c1==c2 && x1=='0' && x2=='1') return true;
         else return false;
      }
   }
}

/*
Nr Nr::operator+(Nr const& n)const
// non-recursive implementation based on 'counting down'
// very slow!! as could be expected
{
   Nr res=n.suc();
   if (sin()) return res;
   else{
      Nr x1=*this;
      while (!x1.sin()){
         x1=x1.pre();
         res=res.suc();
      }
      return res;
   }
}
*/

Nr Nr::operator+(Nr const& n)const
// much faster recursive implementation
{
   if (sin()) return n.suc();
   else if (n.sin()) return suc();
   else{
      Nr c1=cut();
      char x1=last();
      Nr c2=n.cut();
      char x2=n.last();
      char x;
```

```
      if (x1=='0') x=x2;
      else if (x2=='0') x=x1;
      else{
         x='0';
         c1=c1.suc();
      }
      return (c1+c2).app(x);
   }
}

/*
Nr Nr::operator*(Nr const& n)const
// non-recursive implementation based on successive addition
// slow, as could be expected
{
   Nr res=n;
   if (sin()) return res;
   else{
      Nr x1=*this;
      while (!x1.sin()){
         x1=x1.pre();
         res=res+n;
      }
      return res;
   }
}
*/

Nr Nr::operator*(Nr const& n)const
{
   if (sin()) return n;
   else if (n.sin()) return *this;
   else{
      Nr c1=cut();
      char x1=last();
      Nr c2=n.cut();
      char x2=n.last();
      char x;
      Nr res=(c1*c2).app('0').app('0'); // here recursion takes place
      if (x1=='1') res=res+c2.app('0');
      if (x2=='1') res=res+c1.app('0');
      if (x1=='1' && x2=='1') res=res+Nr();
      return res;
   }
}

bool Nr::prnOn(ostream& str)const
{
   return str<<rep<<endl;
}
```

```
Nr Nr::random(int z)
{
   static unsigned long int l=137;
      // never overflows on ++
   cpmassert(z>=1,"Nr::random(int)");
   int zAct=randomZ(z,l++);
   bool b;
   string res="1";
   for (int k=1;k<zAct;++k){
      b=(randomR(l++)>0.5);
      if (b) res+='1';
      else res+='0';
   }
   return Nr(res);
}

int Nr::test(int z)
{
// we test statements that are true and others that are wrong
// so that a corrupted function == which gives 'true'
// always can't fake success.

   int err=0;
   Nr n1=Nr::random(z);
   cout<<endl<<"n1="<<n1;

   bool b=(n1==n1); // should hold
   if (!b) err++;
   cpmmessage("err1="&cpm(err));

   b=(n1<n1); // should nothold
   if (b) err++;
   cpmmessage("err2="&cpm(err));

   b=(n1==n1.suc()); // should not hold
   if (b) err++;
   cpmmessage("err3="&cpm(err));

   b=(n1<n1.suc()); // should hold
   if (!b) err++;
   cpmmessage("err4="&cpm(err));

   Nr n2=Nr::random(z);
   cout<<endl<<"n2="<<n2;
   b=n1+n2==n2+n1;
   if (!b) err++;
   cpmmessage("err5="&cpm(err));

   int sum=0;
```

```
   if (n1<n2) sum++;
   if (n1==n2) sum++;
   if (n2<n1)sum++;
   if (sum!=1) err++;
   cpmmessage("err6="&cpm(err));

   Nr n3=Nr::random(z);
   cout<<endl<<"n3="<<n3;
   b=(n1*(n2+n3)==n2*n1+n1*n3); // should hold
   if (!b) err++;
   cpmmessage("err7="&cpm(err));

   b=(n1*(n2.suc()+n3)==n2*n1+n1*n3); // should not hold
   if (b) err++;
   cpmmessage("err8="&cpm(err));

   n1=Nr::random(z);
   cout<<endl<<"n1="<<n1;
   n2=n1.suc();
   b=(n1==n2);
   if (b) err++;
   cpmmessage("err9="&cpm(err));
   n2=n1;
   b=(n1==n2);
   if (!b) err++;
   cpmmessage("err10="&cpm(err));

   n2=n1.suc().suc().suc().suc();
   n3=n1+Nr(4);
   b=(n2==n3);
   if (!b) err++;
   cpmmessage("err11="&cpm(err));

   n3=n1+Nr(3);
   b=(n2==n3);
   if (b) err++;
   cpmmessage("err12="&cpm(err));

   if (err==0){
      cpmmessage("test OK");
   }
   else{
      cpmmessage("test failed");
   }
   return err;
}
```

126 cpmnumbers.h

```
//? cpmnumbers.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_NUMBERS_H_
#define CPM_NUMBERS_H_
/*
   Description: Introduces a common infrastucture for types
      Z, N, R, Rh, L, bool, string (summerized as bb-types:
      basic built-in types). Thus no longer restricted to numbers.
      Provides most of the content of CpmRoot, cpmword adds the
      rest.

      We try to avoid the unsigned number types and thus don't introduce
      Nh, Nd as in earlier versions of this class library.
      (See. Bjarne Stroustroup: The C++ Programming Language,
      Third Edition, Addison-Wesley p. 70-71 last and first few lines.
      This book will referred to as BS3, the second edition as BS2 )
      Certainly it is highly desirably to have the basic number
      types defined as classes. These seems however be possible only by
      reducing speed by factor 8, or so, in numerics intensive
      programs. This is not considered acceptable here.

      We do not yet make use of the class CpmRoot::Word in this file.

*/
// making available the basic C++ standard library facilities
#include <string>
#include <iostream>
#include <fstream>  // for ifstream, ofstream
#include <sstream>
#include <cmath>
#include <limits>

#include <cpmbasictypes.h>
   // In this files the types Z, N, L, R, and Rh are defined via typedef.
   // It includes <cpmdefinitions.h> for some macros which control
   // compilation and also the storage size of Z and R.

namespace CpmStd{
// This is the part of the standard library which we use frequently
   using std::string;
   using std::pair;
   using std::ostream;
   using std::istream;
   using std::ofstream;
   using std::ifstream;
   using std::ostringstream;
```

```
   using std::istringstream;
   using std::stringstream;
   using std::endl;
   using std::cout;
   using std::cin;
}

namespace CpmRoot{

   using namespace CpmStd;

// Specific numbers

   const R Pi=4.*atan(1.);
      // number pi=3.14159...

   const R Pi2=Pi*2;

   const R PiHalf=Pi*0.5;

   const R PiInv=1./Pi;

   const R Pi2Inv=1./Pi2;

   const R AngDeg=Pi/180;
      //: angular degree

// Formatting matters

   extern Z wrtPrc;
      //: writing precision
      // value for decimals written to strings for real
      // numbers of type R. Notice that all data communicated
      // between processes in my MPI-based parallel computing
      // functions use writing on streams; so numerical errors in
      // communication are ignorable only if this number is large
      // e.g. 16. Now these functions set wrtPrc high and reset
      // it afterwards to a normal value (automatically!)

   extern bool wrtTit;
      //: write title
      // if this is true, all but the elementary tyes get written on
      // stream with a title preceeded by a comment indicator such as
      // '//'

   bool writeTitle(const string& , ostream&);
      // write title
      // uses the string argument to write a title which is precceced
      // by a comment line. Is under the controll of wrtTit
```

```
   bool startsComment(char c);
      // starts comment
      // returns true if c is a character which gives a line the
      // meaning of a comment if it is its first non-whitespace
      // character.
      // Presently startsComment(c)==true for
      // c \in { ';','/','#','*'}

   bool eatComments(istream& in);
      // eat comments
      // returns false if no character not belonging to a comment
      // was found. If return is true, the stream is now in a
      // state that the next reading access will find something
      // not belonging to a comment

// Unified interface to types Z,N,R,Rh,L,bool,string, which we refer to
// as bb-types (basic built-in types). Notice that the types
// Z, N, R, Rh, and L are defined in cpmbasictypes.h via typedef and thus
// are no classes.
// The types bool and string from standard C++ are useful as the stand,
// although they will be wrapped into C+- as class CpmRootX::B in
// cpmtypes.h and class CpmRoot::Word in file cpmword.h.
// --------

// Topic 1: reading instances of bb-types from streams that may contain
// comments

   bool read(Z&, istream&);
   //: read
   // reads a Z from a istream; in doing so it looks out for characters
   // having the meaning of 'comment indicators'. After having identified
   // such a character, the reading action will jump to the beginning of
   // the next line (thus not reading the rest of the previous line,
   // which is supposed to be a comment not containing data for reading).
   // Of course, a comment indicator there, will result in skipping that
   // line too. See function startsComment for the characters that are
   // considered comment indicators.
   // If no number can be read the function returns
   // false otherwise true. Providing a program with numerical data from
   // a text file created by the programmer is useful only if the file is
   // structured by comments explaining the meaning of the data. Thus
   // reading from files containing comment lines is crucial.
   // On ENUMERATIONS:
   // Not only quantities of type Z but also enumerations will be
   // read by this function. Then the syntax of writing and reading
   // should be as follows:
/*
   enum Corners{LL,UL,LR,UR};
   Corners c;
   write((Z)c,cout);
```

```
   ...
   Z temp;
   read(temp,cin);
   Corners c2=(Corners)temp;
*/
   bool read(N&, istream&);
   // reads a N from a istream. Comment handling as in the previous
   // function.

   bool read(R&, istream&);
   // reads a R from a istream. Comment handling as in the previous
   // function.

   bool read(Rh&, istream&);
   // reads a R from a istream. Comment handling as in the previous
   // function.

   bool read(L&, istream&);
   // reads a whitespace-terminated unsigned character value from a
   // istream.
   // Comment handling as in the previous function.

   bool read(bool&, istream&);
   // reads a whitespace-terminated boolean value from a istream.
   // Comment handling as in the previous function.
   // The values are 0 and 1, not strings true or false

   bool read(string& str, istream&);
   // reads a whitespace-terminated word from a istream. Comment handling
   // as in the previous function.

   bool readLine(string& str, istream&);
   // reads a non-comment line from a istream into str. Trailing
   // whitespace such as the most annoying such species CR (carriage
   // return) will not find a way to spoil str. Leading whitespace
   // will not be ignored here since it is essential for copying
   // texts in a readable form by reading and printing lines

// Topic 2: writing instances of bb-types to streams
// in a similar syntax, to make large prnOn and scanFrom functions of
// classes more alike

   bool write(Z, ostream&);
      // enumerations should be explicitely converted to Z
      // before writing
   bool write(N, ostream&);
   bool write(R, ostream&);
   bool write(Rh, ostream&);
   bool write(L, ostream&);
      // characters will be written to streams as integers.
```

```
      // This avoids that a written character '/' or ';' will be
      // mistaken in reading as a comment-line indicator
   bool write(bool, ostream&);
      // expect 'true' to be written as 1 and 'false' as 0
   bool write(string const& , ostream&);

// Miscellanea

   bool isVal(Z x);
   bool isVal(N x);
   bool isVal(R x);
   bool isVal(Rh x);
      // no modes of invalidity of types L, bool, or string
      // known to me

   Z round(R x);
      // returns the nearest integer. If x is exactly between
      // two integers, the smaller one is selected (not the one
      // closer to 0)

   bool apprInt(R x, R tol);
      // returns true if the distance between x and the nearest
      // integer is <= tol (if tol<0, this is never the case)

   L getByte(Z z, Z n);
      // z is an (typically 32 bit)-integer and n varies over 1,2,3,4
      // returned is the first, second, third, forth byte (counted from
      // left to right)

   string skipLeadingWhitespace(string const& str);
      // clear from name

   string skipTrailingWhitespace(string const& str);
      // clear from name

template <class T>
T posVal(T t){ return t<T(0) ? -t : t;}
   //: positive value
   // added 2005-09-10. Similar to absVal but not necessarily
   // R-valued. Not intended to be used for aggregated data types

#define cpmpi     CpmRoot::Pi
#define cpm2pi    CpmRoot::Pi2
#define cpmround  CpmRoot::round
#define cpmpos    CpmRoot::posVal

//////////////////////////////////////////////////////////////////////////
// unified interface to the bb-types and most of the other C+-
// classes.
//////////////////////////// class IO<> ////////////////////////////////
```

```
// Here we describe for the important example of stream IO
// (marshalling) the mehod by which C+- succeeds to treat
// the (non-class or pure C++) bb-types and C+- classes on equal
// footing when these are used as template arguments of
// C+- array classes.
// For a C+- class T it is normal to define the member functions
// (1) bool T::prnOn(ostream& str)const;
// (2) bool T::scanFrom(istream& str);
// Their declaration (together with the definition of some
// related non-member functions) is provided by using the
// macro CPM_IO after a statement 'typedef T Type;'
// For a C+- class template, e.g. CpmArrays::V<T> it is  n o t
// natural to assume that T defines the member functions
// (1) and (2) since T could be Z or R and thus not a class at all.
// Instead, all C+- class templates which implement
// interaction with streams assume that the template argument T
// defines functions
// (3) bool CpmRoot::IO<T>().o(T const& t, ostream& str)const;
// (4) bool CmRoot::IO<T>().i(T& t, istream& str)const;
// There are two possibilities for these functions to get defined:
// (i) It could be that functions (1),(2) are defined for T
//    e.g. since T is a C+- class implementing CPM_IO or T
//    is any user class that favors the names introduced in
//    the NIH class libary. Then the following un-specialized
//    version of the class template IO<T> does the job.
// If, however, T is not a C+- class and cannot be modified by
// adding member functions (1),(2) then
// (ii) a specialization of the class template
//    IO has to make IO<T> defined: Such specializations
//    are given in the following for the  bb-types.
//    Partial specializations (in which T is a class template)
//    can be defined where needed. An example is the dfinition
//    of IO< std::vector<T> > in cpmvl.h.
// Finally there are class-less function templates (near the end of the
// file)
// bool prnOn(T const& t, ostream& str)
// bool scanFrom(T& t, istream& str)
// which will be called in the covenient read and write macros
// #define cpms(X) if (!CpmRoot::scanFrom((X),str)) return false
// #define cpmp(X) if (!CpmRoot::prnOn((X),str)) return false
// fom file cpminterface.h. These functions can also be used in
// template code instead of the more clumpsy functions (3),(4).
// It should be noted that none of he interface service classes
// IO, Comp, Inv, AbsSqr, AbsVal, DisVal, TestVal, RanVal,
// HashVal, Conj, Neutrals contains own data, so that no transport
// of data is involved in their use. Notice that there are two more
// such service classes: Name and ToWord in cpmword.h.
// On Name the C+- type naming system is based, and ToName provides
// convenient short output string representation of values.
// The send and receive functions for MPI usage are based on the
```

```
// present IO-class (see macros CPM_MPI_O and #define CPM_MPI_I
// in cpminterfaces.h).
// For all such service classes except IO there is only one member
// function (not two, namely i and o, in IO) and this is chosen to
// be operator().

template<class T>
class IO{ // input output class template
public:
   IO(){}
   bool o(T const& t, ostream& str)const
      { return t.prnOn(str);}
   bool i(T& t, istream& str)const
      { return t.scanFrom(str);}
};

template<>
class IO<Z>{ // specialization of IO
public:
   IO(){}
   bool o(Z const& t, ostream& str)const
      { return CpmRoot::write(t,str);}
   bool i(Z& t, istream& str)const
      { return CpmRoot::read(t,str);}
};

template<>
class IO<N>{ // specialization of IO
public:
   IO(){}
   bool o(N const& t, ostream& str)const
      { return CpmRoot::write(t,str);}
   bool i(N& t, istream& str)const
      { return CpmRoot::read(t,str);}
};

template<>
class IO<R>{ // specialization of IO
public:
   IO(){}
   bool o(R const& t, ostream& str)const
      { return CpmRoot::write(t,str);}
   bool i(R& t, istream& str)const
      { return CpmRoot::read(t,str);}
};

template<>
class IO<Rh>{ // specialization of IO
public:
   IO(){}
```

```
   bool o(Rh const& t, ostream& str)const
      { return CpmRoot::write(t,str);}
   bool i(Rh& t, istream& str)const
      { return CpmRoot::read(t,str);}
};

template<>
class IO<L>{ // specialization of IO
public:
   IO(){}
   bool o(L const& t, ostream& str)const
      { return CpmRoot::write(t,str);}
   bool i(L& t, istream& str)const
      { return CpmRoot::read(t,str);}
};

template<>
class IO<bool>{ // specialization of IO
public:
   IO(){}
   bool o(bool const& t, ostream& str)const
      { return CpmRoot::write(t,str);}
   bool i(bool& t, istream& str)const
      { return CpmRoot::read(t,str);}
};

template<>
class IO<string>{ // specialization of IO
public:
   IO(){}
   bool o(string const& t, ostream& str)const
      { return CpmRoot::write(t,str);}
   bool i(string& t, istream& str)const
      { return CpmRoot::read(t,str);}
};

//////////////////////////// class Comp ////////////////////////////////////////
// class is named Comp instead of Com in order to avoid confusion
// with CpmMPI::Com
// Ruby style comparison
// com = 'compared' corresponds to the negative of
// Ruby's intelligent operator <=> (only the symbol '<=>' together
// with the ordered list -1,0,1 suggests Ruby's choice; if one is not
// bound by a symbol it is more natural to associate a<b with
// a.com(b)==1 and a>b with a.com(b)==-1.
// Thus our normation:
// a.com(b) ==  0  for  a == b
// a.com(b) ==  1  for  a <  b
// a.com(b) == -1  for  a >  b
```

```
template<class T>
class Comp{ // compare
public:
   Z operator()(T const& x, T const& y){ return x.com(y);}
};

template<>
class Comp<Z>{
public:
   Z operator()(Z const& x, Z const& y)
   { if (x<y) return 1; if (x>y) return -1; return 0;}
};

template<>
class Comp<N>{
public:
   Z operator()(N const& x, N const& y)
   { if (x<y) return 1; if (x>y) return -1; return 0;}
};

template<>
class Comp<R>{
public:
   Z operator()(R const& x, R const& y)
   { if (x<y) return 1; if (x>y) return -1; return 0;}
};

template<>
class Comp<Rh>{
public:
   Z operator()(Rh const& x, Rh const& y)
   { if (x<y) return 1; if (x>y) return -1; return 0;}
};

template<>
class Comp<L>{
public:
   Z operator()(L const& x, L const& y)
   { if (x<y) return 1; if (x>y) return -1; return 0;}
};

template<>
class Comp<bool>{
public:
   Z operator()(bool const& x, bool const& y)
   { if (x<y) return 1; if (x>y) return -1; return 0;}
};

template<>
class Comp<string>{
```

```
public:
   Z operator()(string const& x, string const& y)
   { if (x<y) return 1; if (x>y) return -1; return 0;}
};

//////////////////////// class Inv ////////////////////////////////////

template<class T>
class Inv{ // inverse
public:
   Inv(){}
   T operator()(T const& t)const{ return t.inv();}
};

template<>
class Inv<Z>{ // inverse
public:
   Inv(){}
   Z operator()(Z const& t)const{ return 0;}
};

template<>
class Inv<N>{ // inverse
public:
   Inv(){}
   N operator()(N const& t)const{ return 0;}
};

template<>
class Inv<L>{ // inverse
public:
   Inv(){}
   L operator()(L const& t)const{ return t;}
};

template<>
class Inv<bool>{ // inverse
public:
   Inv(){}
   bool operator()(bool const& t)const{ return !t;}
};

template<>
class Inv<string>{ // reverse string
public:
   Inv(){}
   string operator()(string const& t)const
   {
      string::const_reverse_iterator p1=t.rbegin(), p2=t.rend();
      return string(p1,p2);
```

```
   }
};

R invFunc(R);

template<>
class Inv<R>{ // inverse
public:
   Inv(){}
   R operator()(R const& t)const{ return invFunc(t);}
};

template<>
class Inv<Rh>{ // inverse
public:
   Inv(){}
   Rh operator()(Rh const& t)const{ return (Rh)invFunc(R(t));}
};

///////////////// class AbsSqr ////////////////////////////////////////

template<class T>
class AbsSqr{ // absoute (value) squared
public:
   AbsSqr(){}
   R operator()(T const& t)const{ return t.absSqr();}
};

template<>
class AbsSqr<Z>{ // absoute (value) squared
public:
   AbsSqr(){}
   R operator()(Z const& x)const{ return R(x)*R(x);}
};

template<>
class AbsSqr<N>{ // absoute (value) squared
public:
   AbsSqr(){}
   R operator()(N const& x)const{ return R(x)*R(x);}
};

template<>
class AbsSqr<R>{ // absoute (value) squared
public:
   AbsSqr(){}
   R operator()(R const& x)const{ return x*x;}
};

template<>
```

```
class AbsSqr<Rh>{ // absoute (value) squared
public:
   AbsSqr(){}
   R operator()(Rh const& x)const{ return R(x)*R(x);}
};

template<>
class AbsSqr<L>{ // absoute (value) squared
public:
   AbsSqr(){}
   R operator()(L const& x)const{ return R(x)*R(x);}
};

template<>
class AbsSqr<bool>{ // absoute (value) squared
public:
   AbsSqr(){}
   R operator()(bool const& x)const
      { return x==false ? R(0) : R(1);}
};

template<>
class AbsSqr<string>{
   // here string is interpreted as an array of L's
public:
   AbsSqr(){}
   R operator()(string const& x)const
   { // string components are cased to type L
      R res=0;
      string::const_iterator i;
      for (i=x.begin();i!=x.end();++i) res+=AbsSqr<L>()((L)(*i));
      return res;
   }
};

///////////////// class AbsVal ////////////////////////////////////////

template<class T>
class AbsVal{ // absoute value
public:
   AbsVal(){}
   R operator()(T const& t)const{ return t.absVal();}
};

template<>
class AbsVal<Z>{ // absoute value
public:
   AbsVal(){}
   R operator()(Z const& x)const
      { return x<0 ? (R)(-x) : R(x) ;}
```

```
};

template<>
class AbsVal<N>{ // absoute value
public:
   AbsVal(){}
   R operator()(N const& x)const
      { return R(x) ;}
};

template<>
class AbsVal<R>{ // absoute value
public:
   AbsVal(){}
   R operator()(R const& x)const
      { return x<0 ? -x : x ;}
};

template<>
class AbsVal<Rh>{ // absoute value
public:
   AbsVal(){}
   R operator()(Rh const& x)const
      { return x<0 ? R(-x) : R(x) ;}
};

template<>
class AbsVal<L>{ // absoute value
public:
   AbsVal(){}
   R operator()(L const& x)const
      { return R(x) ;}
};

template<>
class AbsVal<bool>{ // absoute value
public:
   AbsVal(){}
   R operator()(bool const& x)const
      { return x==false ? R(0) : R(1) ;}
};

template<>
class AbsVal<string>{
   // here string is interpreted as an array of L's
public:
   AbsVal(){}
   R operator()(string const& x)const
   { return ::sqrt(AbsSqr<string>()(x)) ;}
};
```

```
//////////////////////// class DisVal ////////////////////////////////////////
// For T t1,t2 the real number disVal(t1,t2) belongs to [0,1]
// and tries to assess whether t1 and t2 are significantly different
// or differ only by numerical noise. Thus for discrete types we
// set 0 for t1==t2 and 1 else. For 'continuous types' we use
// the natural construct disVal(x,y)=Min( |x-y|, |x-y|/(|x|+|y|) )
// = distFunc(|x|,|y|,|x-y|)
// based on the concepts of absolute value and difference.

R distFunc(R a, R b, R d);
   // Function for implementing function disVal
   // needs also be known to cpminterfaces.h
   // continuous function, but not reasonable from a geometrical
   // point of view since the distance between points based naturally
   // on this function is not translation invariant.

template<class T>
class DisVal{
public:
   DisVal(){}
   R operator()(T const& x, T const& y){ return x.disVal(y);}
};

template<>
class DisVal<Z>{
public:
   DisVal(){}
   R operator()(Z const& x, Z const& y)
   { return x!=y ?  1. : 0.;}
};

template<>
class DisVal<N>{
public:
   DisVal(){}
   R operator()(N const& x, N const& y)
   { return x!=y ?  1. : 0.;}
};

template<>
class DisVal<L>{
public:
   DisVal(){}
   R operator()(L const& x, L const& y)
   { return x!=y ?  1. : 0.;}
};

template<>
class DisVal<bool>{
```

```
public:
   DisVal(){}
   R operator()(bool const& x, bool const& y)
   { return x!=y ?  1. : 0.;}
};

template<>
class DisVal<string>{
public:
   DisVal(){}
   R operator()(string const& x, string const& y)
   { return x!=y ?  1. : 0.;}
};

template<>
class DisVal<R>{
public:
   DisVal(){}
   R operator()(R const& x, R const& y)
   { return distFunc(posVal<R>(x),posVal<R>(y),posVal<R>(x-y));}
};

template<>
class DisVal<Rh>{
public:
   DisVal(){}
   R operator()(Rh const& x, Rh const& y)
   { return distFunc(posVal<Rh>(x),posVal<Rh>(y),posVal<Rh>(x-y));}
};

//////////////////////// class TestVal ////////////////////////////////////
// the first argument of the function is used only to carry the
// type information
// The intented usage is as follows:
//    T t;
//    Z complexity=3; // for example
//    T tc=CpmRoot::testVal(t,complexity);
//    T t1=CpmRoot::ranVal(tc,1);
//    T t2=CpmRoot::ranVal(tc,2);
//    T t3=CpmRoot::ranVal(tc,3);
// which is an unbiased way to create an arbitrarily large
// set of T-values

template<class T>
class TestVal{ // test value
public:
   TestVal(){}
   T operator()(T const& t, Z complexity)const
   { return t.testVal(complexity);}
};
```

```
template<>
class TestVal<Z>{
public:
   TestVal(){}
   Z operator()(Z const& x, Z complexity)const
   { x; return complexity;}
};

template<>
class TestVal<N>{
public:
   TestVal(){}
   N operator()(N const& x, Z complexity)const
   { x; return (N)complexity;}
};

template<>
class TestVal<L>{
public:
   TestVal(){}
   L operator()(L const& x, Z complexity)const
   { x; Z cp=posVal(complexity); return L(cp%256);}
};

template<>
class TestVal<bool>{
public:
   TestVal(){}
   bool operator()(bool const& x, Z complexity)const
   { x; Z cp=posVal(complexity); return cp%2==1;}
};

template<>
class TestVal<R>{
public:
   TestVal(){}
   R operator()(R const& x, Z complexity)const
   { x; return 1.123456789*complexity;}
};

template<>
class TestVal<Rh>{
public:
   TestVal(){}
   Rh operator()(Rh const& x, Z complexity)const
   { x; return (Rh)1.123456789*complexity;}
};

template<>
```

```
class TestVal<string>{
public:
   TestVal(){}
   string operator()(string const& x, Z complexity)const
   {
      x;
      if (complexity<=1)
         return "az";
      else if (complexity==2)
         return "abcdefghijklmnopqrstuvwxyz";
      else if (complexity==3)
         return "abcdefghijklmnopqrstuvwxyz0123456789";
      else return
         "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
   }
};

////////////////////// class RanVal ////////////////////////////////////
R randomR(Z j=0);
   // returns uniformly distributed numbers in [0,1)
Z randomZ(Z n, Z j=0);
   // returns uniformly distributed integers in the natural
   // index range [1,n]. Thus stops with error if n<1.
   // Function IvZ::ran allows to set lower limit and upper limit
   // arbitrarly.
extern R facRanGen; // factor random generator
      // a method parameter for the basic random generator
      // has no influence only for metRanGen=2
extern Z metRanGen;
      // method selection for the basic random generator randomR
      // 1: sine-floor
      // 2: 160807 congruence method
      // else: randomX, an experimental variation of 1.
      //    also reacting on facRanGen

template<class T>
class RanVal{ // random value
public:
   RanVal(){}
   T operator()(T const& t, Z j)const
   { return t.ranVal(j);}
};

template<>
class RanVal<Z>{
public:
   RanVal(){}
   Z operator()(Z const& i, Z j)const
   {
      R y=randomR(j);
```

```
      Z ia=(i>=0 ? i : -i);
      if (ia==0) ia=1;
      // without this modification we would get 0,0,0,... for i=0. This
      // 'random' sequence will probably never be needed in a serious
      // context.
      // More probably i itself will result from a random process and
      // nevertheless ranVal(i,i2) will be expected to be random in
      // i2.
      // Thus we increase ia in order to get the random sequence with
      // values
      // in {-1,0,1}.
      Z n=2*ia+1;
      y*=n; // belongs to [0,2|i|+1)
      Z res=(Z)y;
      if (res==n) res--; // should not happen
      return res-ia;
   }
};

template<>
class RanVal<N>{
public:
   RanVal(){}
   N operator()(N const& i, Z j)const
   {
      R y=randomR(j);
      y*=i; // belongs to [0,i)
      N res=(N)y; // belongs to {0,1,..., i-1}=
      if (res==i) res--; // should not happen
      return res;
   }
};

template<>
class RanVal<L>{
public:
   RanVal(){}
   L operator()(L const& x, Z j)const
   {
      R y=randomR(j);
      y*=256; // belongs to [0,256)
      Z zy=(Z)y; // belongs to {0,1,..., 255}=
      if (zy==256) zy--; // should not happen
      return (L)zy;
   }
};

template<>
class RanVal<bool>{
public:
```

```
   RanVal(){}
   bool operator()(bool const& b, Z j)const
   {
      R y=randomR(j);
      if (y<=0.5) return b; else return !b;
   }
};

template<>
class RanVal<R>{
public:
   RanVal(){}
   R operator()(R const& x, Z j)const
   {
      R y=2*randomR(j)-1;
      return y*AbsVal<R>()(x);
   }
};

template<>
class RanVal<Rh>{
public:
   RanVal(){}
   Rh operator()(Rh const& x, Z j)const
   {
      R y=2*randomR(j)-1;
      return (Rh)(y*AbsVal<Rh>()(x));
   }
};

template<>
class RanVal<string>{
public:
   RanVal(){}
   string operator()(string const& a, Z i)const
   {
      string res=a;
      Z l=i;
      string::const_iterator j;
      string::iterator k;
      for (j=a.begin(),k=res.begin();j!=a.end(),k!=res.end();++j,++k){
         *k=*j^CpmRoot::RanVal<L>()(*j,*j+l++);
      }
      return res;
   }
};

//////////////////////// hashVal ////////////////////////////////////////
// If for T t1,t2 one has hashVal(t1)==hashVal(t2) there
// should be a overwhelming probability that t1 is equal
```

```
// to t2 (unless t2 is obtained from t1 by only a minor
// modification, or even by a modification which is aware
// of the algorithm for hashVal.

template<class T>
class HashVal{ // hash value
public:
   HashVal(){}
   Z operator()(T const& t)const
   { return t.hashVal();}
};

template<>
class HashVal<Z>{
public:
   HashVal(){}
   Z operator()(Z const& x)const
   {  return x-x%137+(x/13)*7;}
};

template<>
class HashVal<N>{
public:
   HashVal(){}
   Z operator()(N const& x)const
   {  return HashVal<Z>()(x);}
};

template<>
class HashVal<L>{
public:
   HashVal(){}
   Z operator()(L const& x)const
   {
      Z y=(Z)x+117;
      return HashVal<Z>()(y);
   }
};

template<>
class HashVal<bool>{
public:
   HashVal(){}
   Z operator()(bool const& x)const
   { return x ? 137 : 13;}
};

template<>
class HashVal<string>{
public:
```

```
   HashVal(){}
   Z operator()(string const& x)const
   {
      Z sum=137;
      string::const_iterator i;
      for (i=x.begin();i!=x.end();++i) sum+=(Z)(*i);
      return HashVal<Z>()(sum);
   }
};

template<>
class HashVal<R>{
public:
   HashVal(){}
   Z operator()(R const& x)const
   {
      const R reg=100000;
      R a=(x<0. ? -x : x);
      return (Z)(x+reg/(1.+a*reg));
   }
};

template<>
class HashVal<Rh>{
public:
   HashVal(){}
   Z operator()(Rh const& x)const
   {
      return HashVal<R>()((R)x);
   }
};

//////////////////////// conj ////////////////////////////////////////
// This describes a concept that does not convincingly arrises
// from the types under present consideration. It will
// represent complex conjugation of complex numbers, transposition
// of real matrices, and Hermitean conjugation of complex
// matrices.

template<class T>
class Conj{ // conjugation
public:
   Conj(){}
   T operator()(T const& t)const
   { return t.con();}
};

template<>
class Conj<Z>{
public:
```

```
   Conj(){}
   Z operator()(Z const& t)const
   { return t;}
};

template<>
class Conj<N>{
public:
   Conj(){}
   N operator()(N const& t)const
   { return t;}
};

template<>
class Conj<L>{
public:
   Conj(){}
   L operator()(L const& t)const
   { return t;}
};

template<>
class Conj<bool>{
public:
   Conj(){}
   bool operator()(bool const& t)const
   { return t;}
};

template<>
class Conj<string>{
public:
   Conj(){}
   string operator()(string const& t)const
   { return t;}
};

template<>
class Conj<R>{
public:
   Conj(){}
   R operator()(R const& t)const
   { return t;}
};

template<>
class Conj<Rh>{
public:
   Conj(){}
   Rh operator()(Rh const& t)const
```

```
   { return t;}
};

////////////////////// neutrals ////////////////////////////////////////
// for T t; the quantity CpmRoot::net<T>(0) is the
// instance of T that is zero or comes as close as possible
// to realising the concept of a zero of type T.
// Correspondingly CpmRoot::net<T>(1) with zero
// replaced by unity. The name comes from the fact that
// zero is a neutral element for addition and
// unity is a neutral element for multiplication.
// This renders the concept ambiguous for T==string since
// here we do not refer only to addition (concatenation)
// but not to multiplication.

template<class T>
class Neutrals{ // neutral elements
public:
   Neutrals(){}
   T operator()(T const& t, Z i)const
   { return t.net(i);}
};

template<>
class Neutrals<Z>{
public:
   Neutrals(){}
   Z operator()(Z const& t, Z i)const
   { return i==1 ? 1 : 0;}
};

template<>
class Neutrals<N>{
public:
   Neutrals(){}
   N operator()(N const& t, Z i)const
   { return i==1 ? 1 : 0;}
};

template<>
class Neutrals<L>{
public:
   Neutrals(){}
   L operator()(L const& t, Z i)const
   { return i==1 ? L(1) : L(0);}
};

template<>
class Neutrals<bool>{
public:
```

```
   Neutrals(){}
   bool operator()(bool const& t, Z i)const
   { return i==1 ? true : false;}
};

template<>
class Neutrals<string>{
public:
   Neutrals(){}
   string operator()(string const& t, Z i)const
   { return i==1 ? "1" : "";}
};

template<>
class Neutrals<R>{
public:
   Neutrals(){}
   R operator()(R const& t, Z i)const
   { return i==1 ? R(1) : R(0);}
};

template<>
class Neutrals<Rh>{
public:
   Neutrals(){}
   Rh operator()(Rh const& t, Z i)const
   { return i==1 ? Rh(1) : Rh(0);}
};

//////////////////// class-less function templates ////////////////////////
// this is the most convenient form for using the service functions.
// We thus list them together:

// implementation of CPM_IO
template<class T>
bool prnOn(T const& t, ostream& str)
   //: print on
{ return CpmRoot::IO<T>().o(t,str);}

template<class T>
bool scanFrom(T& t, istream& str)
   //: scan from
{ return CpmRoot::IO<T>().i(t,str);}

// implementation of CPM_ORDER
template<class T>
Z com(T const& t1, T const& t2){ return Comp<T>()(t1,t2);}
   //: compare

// implementation of CPM_TEST_ALL
```

```
template<class T>
T inv(T const& t){ return Inv<T>()(t);}
   //: inverse

template<class T>
R absSqr(T const& x){ return AbsSqr<T>()(x);}
   //: absolute (value) squared

template<class T>
R absVal(T const& x){ return AbsVal<T>()(x);}
   //: absolute value

template<class T>
R disVal(T const& t1, T const& t2)
   //: distance value
{ return DisVal<T>()(t1,t2);}

template<class T>
T testVal(T const& t, Z complexity)
   //: test value
{ return TestVal<T>()(t,complexity);}

template<class T>
T ranVal(T const& t, Z j=0){ return RanVal<T>()(t,j);}
   //: random value

template<class T>
Z hashVal(T const& t){ return HashVal<T>()(t);}
   //: hash value

template<class T>
T con(T const& t){ return Conj<T>()(t);}
   //: conjugate

template<class T>
T net(T const& t, Z i){ return Neutrals<T>()(t,i);}
   // neutrals

} // namespace

// convenient abbreviations for the service functions
// See also corresponding abbreviations
// cpmnam, cpmtow in cpmword.h and
// cpmswp, cpmord, cpminf, cpmsup, cpmtin, cpmhug in cpmtypes.h

#define cpmprn    CpmRoot::prnOn
#define cpmscn    CpmRoot::scanFrom
#define cpmcom    CpmRoot::com
#define cpminv    CpmRoot::inv
#define cpmabs    CpmRoot::absVal
```

```
#define cpmab2    CpmRoot::absValSqr
#define cpmdis    CpmRoot::disVal
#define cpmran    CpmRoot::ranVal
#define cpmcon    CpmRoot::conj
#define cpmtes    CpmRoot::test
#define cpmhas    CpmRoot::hash
#define cpmnet    CpmRoot::neutrals

#endif
```

127 cpmnumbers.cpp

```
//? cpmnumbers.cpp
//? Status of work 2008-10-25.
//?
#include <cpmnumbers.h>
#include <cpmsystem.h>
#include <cpmsystemdependencies.h>
// defines wrtPrc, wrtTit

using namespace CpmStd;
using CpmRoot::R;
using CpmRoot::Rh;
using CpmRoot::Z;
using CpmRoot::N;
using CpmRoot::L;
using CpmRoot::Word;

Z CpmRoot::round(R x)
{
   R fl=::floor(x);
   R h=x-fl;
   Z res=(Z)fl;
   if (h>0.5) res++;
   return res;
}

bool CpmRoot::apprInt(R x, R tol)
{
   R d=x-round(x);
   return ( d>=0 ? d<=tol : -d<=tol);
}

namespace{
   L getbits(N x, N p, unsigned n)
   // Kernighan Ritchie: Programming in C, section 2.9
   { return (L)((x>>(p+1-n))& ~(~0<<n));}
}

L CpmRoot::getByte(Z z, Z n)
{
   N x=(N)z;
   return getbits(x, (n-1)*4,4);
}

#if defined(CPM_WRITE_PRECISION)
   Z CpmRoot::wrtPrc=CPM_WRITE_PRECISION;
#else
   Z CpmRoot::wrtPrc=10;
```

```
#endif

#if defined(CPM_WRITE_TITLE)
   bool CpmRoot::wrtTit=true;
#else
   bool CpmRoot::wrtTit=false;
#endif

bool CpmRoot::writeTitle(string const& s , ostream& out)
      // uses the string argument to write a title which is precceced
      // by a comment line
{
   if (wrtTit){
      CpmRoot::write(string("//")+s,out);
   }
   return (out!=0);
}

string CpmRoot::skipLeadingWhitespace(string const& str)
{
   Z n=(Z)str.size();
   if (n==0) return str;
   char cTest=str[0];
   if ( !isspace(cTest)) return str;
      // fast part of the process, no leading whitespace there
// now the first character is space
   Z i=0;
   while(i<n && isspace(str[i])) i++;
// eliminate the i leading characters
   string res=str;
   res.erase(0,i);
   return res;
}

string CpmRoot::skipTrailingWhitespace(string const& str)
   // made 2004-10-12 in analogy to the previous function
{
   Z n=(Z)str.size();
   if (n==0) return str;
   char cTest=str[n-1];
   if ( !isspace(cTest)) return str;
      // fast part of the process, no trailing whitespace there
// now the last character is space
   Z iTest=n-1;
   Z found=0;
   while(iTest>=0 && isspace(str[iTest])){
      iTest--;
      found++;
   }
   // now iTest is -1 or the index of a character
```

```
   // so iTest++ is the insex of the first of the trailing
   // whitespaces
   iTest++;
// eliminate the found trailing characters
   string res=str;
   res.erase(iTest,found);
   return res;
}

bool CpmRoot::startsComment(char c)
{ return c=='/' || c==';'|| c=='*'|| c=='#';}

bool CpmRoot::eatComments(istream& in)
{
//   if (!wrtTit) return true;
      // this first line creates misbehaviour in reading  if
      // Word::scnLin==true; Not completely understood; my idea
      // was to work with Word::scnLin==false anyway
   const Z mL=3;
   Word loc("CpmRoot::eatComments(istream&)");
   char c=0;
   string buff;
   if (!in){
      cpmmessage(mL,loc&": bad stream before reading");
      return false;
   }
LOOP:
   if (!(in>>c)){
      ostringstream ost;
      ost<<loc<<":  bad stream after reading character c="<<(Z)c;
      cpmmessage(mL,Word(ost.str()));
      return false;
   }
   if (startsComment(c)){
      std::getline(in,buff); // reads the line till end of line
         // including the '\n' into the self-adjusting buff
      goto LOOP;
   }  // now c is no longer the potential beginning of a comment
   in.putback(c);
   if (!in){
      cpmmessage(mL,loc&": bad stream after reading");
      return false;
   }
   return true;
}

bool CpmRoot::readLine(string& line , istream& in)
{
   if  (!eatComments(in)) return false;
   std::getline(in,line);
```

```
   line=skipTrailingWhitespace(line);

   return in ? true : false;
}

#define CPM_FB\
   if  (!eatComments(in)) return false;\
   if (in>>x) return true; else return false;

bool CpmRoot::read(Z& x, istream& in){CPM_FB}

bool CpmRoot::read(N& x, istream& in){CPM_FB}

bool CpmRoot::read(R& x, istream& in){CPM_FB}

bool CpmRoot::read(Rh& x, istream& in){CPM_FB}

bool CpmRoot::read(string& x, istream& in){CPM_FB}

bool CpmRoot::read(bool& x, istream& in){CPM_FB}
// BS: The C++ programming language 3. edition, p. 615:
// 'By default, the bool values true and false are represented by
// 1 and 0, respectively. Thus no reading success for e.g. 2

bool CpmRoot::read(L& x, istream& in)
{
   Z z;
   bool suc=CpmRoot::read(z,in);
   x=(L)z;
   return suc;
}

#undef CPM_FB
//#define CPM_FB out<<x;
//#define CPM_FB out<<endl<<x<<endl; return (out!=0);
   // 2001-1-23 endl added at the end. This eliminated occasional
   // failure in reading R's which were written previously by
   // the write function
//#define CPM_FB out<<endl<<x<<std::flush; return (out!=0);

#define CPM_FB out<<x<<endl; return (out!=0);
//#define CPM_FB out<<x; return (out!=0);

bool CpmRoot::write(Z x, ostream& out){CPM_FB; }

bool CpmRoot::write(N x, ostream& out){CPM_FB; }

#if defined(WIN32)
bool CpmRoot::write(R r, ostream& out)
// experimental avoidance of writing too small numbers
```

```
// which caused trouble under Windows
{
   const R wrTiny=1e-304;
      // 1e-305 works, 1e-310 works not
   out.precision(wrtPrc);
   R x=r;
   if (x>0){
      if (x<wrTiny) x=wrTiny;
   }
   else if (x<0){
      if (x>-wrTiny) x=-wrTiny;
   }
   else{
      x=0;
   }
   CPM_FB
}
#else
bool CpmRoot::write(R x, ostream& out){out.precision(wrtPrc);CPM_FB}
#endif

bool CpmRoot::write(Rh x, ostream& out){out.precision(wrtPrc);CPM_FB}
bool CpmRoot::write(const string& x, ostream& out){CPM_FB}
bool CpmRoot::write(bool x, ostream&  out){ CPM_FB}
bool CpmRoot::write(L x, ostream&  out)
{
   return write((Z)x,out);
}

#undef CPM_FB

#define CPM_FB\
   if (cpmdbg>1){\
   ostringstream ost;\
   ost<<x;\
   string s=ost.str();\
   bool b1=s.find("I")==string::npos;\
   bool b2=s.find("N")==string::npos;\
   return b1&&b2;}\
   else return true;
   // Universal, but potentially ineffective method. With cpmdbg==1,
   // which is the setting for 'production', it causes only negligible
   // action. Test for INF added 2003-03-21, test for INF and NAN changed
   // into one for I and N since g++ writes Inf and (?) Nan

bool CpmRoot::isVal(R x){CPM_FB}

bool CpmRoot::isVal(Z x){CPM_FB}

bool CpmRoot::isVal(N x){CPM_FB}
```

```
bool CpmRoot::isVal(Rh x){CPM_FB}

#undef CPM_FB

////////////////////// function inv ////////////////////////////////////

R CpmRoot::invFunc(R x)
{
   const Z maxMes=100;
   static Z mes=1;
   if (x==0.){
      if (mes==maxMes){
         mes++;
         cpmmessage("inv<R>: argument is 0 ... messages discontinued");
      }
      if (mes<maxMes){
         mes++;
         cpmwarning("inv<R>: argument is 0, 0 returned");
      }
      return 0.;
   }
   else return 1./x;
}

////////////////////// disVal ////////////////////////////////////////

R CpmRoot::distFunc(R a, R b, R d)
{
   R s=a+b;
   if (s==0.) return 0.;
   R d1=d/s;
   return (d < d1 ? d  : d1);
}

R CpmRoot::facRanGen=1e6;
Z CpmRoot::metRanGen=1;

/************ sawtooth replacement for experimentation ************/
// test 2001-1-13

namespace{
   R randomX(Z i)
   {
      const R per=CpmRoot::Pi;
      const R perInv=1./per;
      R x=R(i);
      R x1=x*perInv;
      R x2=floor(x1);
      x-=(x2*per);
```

```
      x*=perInv;
      R pp=1.2;
      x=pow(x,pp);
      R y=x*CpmRoot::facRanGen;
   //   y*=y; // some non-linear operation on y improves behavior
      return y-=floor(y);
   }

   R random16807(Z j)
// the '*16807 generator' as described by Stauffer
// for 32 bit integers
// j has no influence
{
   static bool firstrun=true;
   const Z iSeed=9;
   const R fac=pow(2.,-32);
   static Z ibm;
   if (firstrun){
      Z szi=sizeof(ibm);
      if (szi!=4) cpmerror("randomR(Z): no 32 bit integers");
      ibm=2*iSeed-1;
      firstrun=false;
   }
   j; // no influence
   ibm*=16807; // =7^5
   return 0.5+fac*ibm;
}
} // namespace

R CpmRoot::randomR(Z j)
// returns randomly distributed numbers in the interval [0,1)
// as a function of j=1,2,3,.... The validity of this statement can be
// considered evident; it seems to be hard to translate this 'evidence'
// into something that can be proved.
{
      // 2001-1-11: the 'cube emptying test' (see e.g.
      // Dietrich Stauffer: Random Number Generation. In:
      // K.H.Hoffmann, M. Schreiber (Eds.): Computational Physics,
      // Springer 1996) p. 6-7, seems always to detect deviations
      // from ideal behavior if the cube is large enough. My old value
      // fac=5000 started to misbehave for cube site n=45.
      // fac=50000 was close to OK for n=100
      // fac=1e6 is perfect for n=100.
      // Replacing sin by a sawtooth-curve makes the cube emptying
      // test fail completely. So the curvature of the curve
      // seems to be essential. This came as a surprise to me.
   static N j0=137;
// I had Z j0 till 2004-11-02.
// As Ellis and Stroustrup say (p. 23): Unsigned integers, declared
// unsigned, obey the laws of arithmetic modulo 2^n where n is the
```

```
// number of bits in the representation. This implies that unsigned
// arithmetic does not overflow. The choice 137 is mere idiosyncrasy.

   if (CpmRoot::metRanGen==1){
      R x=(j!=0 ? j : ++j0);
      R y=CpmRoot::facRanGen*::sin(x);
      return y-=floor(y);
   }
   else if (CpmRoot::metRanGen==2){
      return random16807(j);
   }
   else{
      return randomX(j);
   }
}

Z CpmRoot::randomZ(Z n, Z j)
{
   if (n<1) cpmerror("n<1 in CpmRoot::randomZ(Z n, Z j)");
   R y=n*randomR(j);
   Z res=(Z)y;
   return ((res==n) /* should not happen */ ? n : res+1);
}
```

128 cpmobservable.h

```
//? cpmobservable.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_OBSERVABLE_H_
#define CPM_OBSERVABLE_H_
/*
   Description: objects for describing quantum mechanical observables
      as Hilbert space operators
*/
#include <cpmv.h>
#include <cpmf.h>
#include <cpmrmatrix.h>
#include <cpmstate.h>
#include <cpmangle.h>
#include <cpmp.h>
#include <cpmalgorithms.h>
#include <cpmrecord.h>

namespace CpmQuantumMechanics{

   using namespace CpmArrays;
   using namespace CpmFunctions;
   using namespace CpmGraphics;

   using CpmRootX::R1;
   using CpmRootX::Record;
   using CpmGeo::Angle;
   using CpmLinAlg::R3;
   using CpmAlgorithms::CyclicAlarm;

//////////////////// class SpectralObject ////////////////////////////

// combines a set of (complex) spectral values and a set of orthonormal
// set of vectors into a single object which according to the spectral
// theorem results from spectral analysis of a normal operator.
// Here we know only how to generate such an object from a self-adjoint
// operator.
// The most important constructive method is function
// SpectralObject
//    Observable::spectralRepresentation(bool,bool,Z,R,R,R)const;
// for creation from an Observable (which is basicly a complex matrix).
// There are, however, elementary observables for which it is
// simpler to define them as SpectralObjects (see constructor
// SpectralObject(const Word& w, Z d_);

   class Observable;
```

```
class SpectralObject:public QMBase{//eigenvectors/values in combination
      Z d_;
      Vl<C> spectrum;
      Vl<State> eigenVectors;
         // complete set of orthonormal vectors
   public:
      typedef SpectralObject Type;
      CPM_IO
      CPM_NAM(SpectralObject)
      friend class Observable;
      Z dim()const{ return d_;}
      const C& operator()(Z i)const{ return spectrum[i];}
         // eigenvalues by (), notice that i has to be in
         // {0,...,d_-1}.
      const State& operator[](Z i)const{ return eigenVectors[i];}
         // eigenvectors by [], notice that i has to be in
         // {0,...,d_-1}, the order is such that the ground-state of
         // the harmonic oscillator is obtained for i=d_-1
         // is this OK ?

      SpectralObject(){}

      explicit SpectralObject(Z d);
         // has the 'canonical vectors' as eigenvectors
         // and the canonical count index 0,1,.... d-1 as eigenvectors

      SpectralObject(const Word& w, Z d);
         // for w="zero" or "0":  zero operator
         // for w="unit" or "1":  unit operator
         // for w="space frequency" or "f": space frequency operator
         // for w="wave vector" or "k": wave vector operator
         // for w="momentum" or "p": momentum operator
         // for w="kinetic energy" or "eKin": operator of kinetic energy
         // for w="relativistic kinetic energy" or "eKinRel":
         //    operator of relativistic kinetic energy T=p^2/(2*mass)
         //    for which we give here the nice implicit equation
         //    T=p^2/(2*mass + T/c^2) which is an easy consequence
         //    of (E/c)^2 - p^2 = (mass*c)^2 and T=E-mass*c^2.
         // for w="position" or "x": position operator
         // for w="boost" or "n": non-relativistic boost operator

         // if w="f" or "k" or "p" or "eKin" or one of the full word
         // equivalents, d_ has to be a power of two. This is since the
         // construction then is based on passage to momentum space by
         // discrete (fast) Fourier transform.
   protected:
      SpectralObject(const Vl<C>& s, const Vl<State>& e):
         d_(s.dim()),spectrum(s),eigenVectors(e)
         {cpmassert(d_==e.dim(),"SpectralObject(...)");}
         // unsafe construction which only tests
```

```
      // fit of the dimensions but not that e is orthonormal

   SpectralObject(const Vl<R>& s, const Vl<State>& e);
      // unsafe construction which only tests
      // fit of the dimensions but not that e is orthonormal

public:
   R_Vector getSpectrum()const
   {
      R_Vector res(d_);
      for (Z i=1;i<=d_;i++) res[i]=spectrum[i-1][1];
      return res;
   }

   R spectralRadius()const
   {
      R_Vector rs(d_);
      for (Z i=0;i<d_;i++) rs[i+1]=spectrum[i].absVal();
      return rs.sup();
   }

   void setSpectrum_(C z){ for (Z i=0;i<d_;i++) spectrum[i]=z;}

   void show(Frame& f, Z mult=-1, Z iShift=0, Z jShift=0)const;

   void showSpectrum(Graph& gr)const;
      // real part of eigenvalues, as autoscaled graph

   bool writeSpectrum(ostream& str)const;
      // writes the eigenvalues to str, returns true
      // if writing was successful

   void map(const F<C,C>& f)
      // f acts on the spectrum
   {for (Z i=0;i<d_;i++) spectrum[i]=f(spectrum[i]);}

   void map(const F<R,C>& f)
      // f acts on the spectrum
   {for (Z i=0;i<d_;i++) spectrum[i]=f(spectrum[i][1]);}

   void expi(R t);
      // changes *this into exp(i*t*(*this))

   void map_(const R_Func& f)
      // real function f acts on the real part of the spectrum
   {
      for (Z i=0;i<d_;i++)
         spectrum[i]=C(f(spectrum[i][1]),spectrum[i][2]);
   }
```

```
R test()const;
   // returns a error sum which is large if the spectral data
   // are not correct

void normalize_()
   { for (Z i=0;i<d_;i++) eigenVectors[i].normalize_();}
// normalizes all eigenvectors

State operator *(const State&)const;
// SpectralObject acts on state just as the underlying observable

X2<R,State> reduction(State const& psi, R p=1.)const;
// performs the 'collapse of the wave packet' of the state
// given by the argument and the observable of spectral
// representation *this. Returned is the 'measured value' and the
// state after measurement. The parameter gives the probability
// for actually doing the reduction upon function call.

State dePhase(const State& psi, R delta=0., R p=0.25,
   Z iter=8)const;
// dephasing of a state due to interaction with spectral object
// *this. delta is the 'decision variable'
// See U. Mutze: Modeling the 'Collapse of the Wave Function' as a
// Deterministic Process. Developing further this collapse model
// could benefit from using my new approach to computational
// quantum mechanics as mentioned in U.Mutze: Predicting Classical
// Motion Directly from the Action Principle III and coded in
// cpmpsi.h, cpmpsi.cpp (in d:\cpm\cpm1). Since my present form of
// the dephasing mechanism yet relies on spectral decomposition and
// not entierly on applying operators of the form exp(itA) on
// states, the class Psi has first to be equiped with spectral
// theory or the latter has to be replaced by a more operational
// alternative.

State represent(const State& psi)const;
 // represents psi as a 'eigenfunction expansion with
 // respect to *this'

static State dePhase(const State&,R,const Vl<SpectralObject>&,
   const Vl<R1>&);
// dephasing of a state due to interaction with a family of
// spectral objects. The result has a cleaner meaning if the
// spectral objects in the family commute with each other.

SpectralObject order()const;
   // returns a new SpectralObject which results form *this
   // by common re-ordering of eigenvalues and eigenvectors
   // such that the real parts of the eigenvalues are in increasing
   // order with the index. Thus for the harmonic oscillator
   // the ground state corresponds to index 0.
```

```
         // Quite elegant implementation!
   };


//////////////////////// class Observable ////////////////////////

class Observable: public QMBase{ // observable as matrix
   // not assumed selfadjoint in order
   // to include unitary operators, and perhaps others

private:
   Vl< Vl<C> > rep;

   Observable transpose(void)const;
      // is not a canonical operation in Hilbert space, only con() is
   Z adjust(Z i)const;
      // returns a 'version of i' in the range 0<=i<dim();
public:
   typedef Observable Type;
   CPM_IO
   CPM_PURE_VECTOR_SPACE_INTERFACE(Type,C)
   CPM_SCALAR_PRODUCT_INTERFACE(Type,C)
   CPM_PURE_SCALAR_INTERFACE(Type,R)
      // Scalar Product and norm of Hilbert-Schmidt operators
   CPM_NAM(Observable)
   Observable(const Vl< Vl<C> >& rep_):rep(rep_){}
      // was private till 2006-07-12
   Observable operator *(Observable const&)const;
      // The strategy is to express the matrix elements of the result in
      // terms of binary expressions of component vectors x[i], y[j]
      // Notice (x*y)[i][j]=sum over k x[i][k] * (y.transpose())[j][k] =
      // bilinear scalar product of x[i] and (y.transpose())[j].
      // This is the most efficient way since such functions are
      // implemented as a fast *p++-run over pointers without making use
      // of indexing

   Observable prodStar(Observable const&)const;
      // returns the value of x*y.con()
      // This is the fastest product operation, since it has not to
      // perform a transposition or conjugation in order to write the new
      // matrix elements as binary expressions of component vectors.
      // Had we available column vectors as representors of matrixes,
      // then the combination x.con()*y would be the one most
      // economically expressible in terms of binary expressions of
      // component vectors Comparison of prodStar and operator * gave 30%
      // slower execution for prodStar after optimized compilation! Not
      // clear why that way, where an opposit result was expected. We
      // learn at least that it is not worth the effort to replace
      // products of hermitean operators by prodStar systematically.
```

```
Observable(){}

explicit Observable(Z d):rep(d, Vl<C>(d)){}
   // Thus d is the dimension of the underlying State
   // created is the zero operator

Observable( R r, Z d):rep(d, Vl<C>(d))
   { for (Z i=0;i<d;++i) rep[i][i]=C(r);}
   // creating a real multiple of the unit operator.

Observable( C z, Z d):rep(d, Vl<C>(d))
   { for (Z i=0;i<d;++i) rep[i][i]=z;}
   // creating a complex multiple of the unit operator.

explicit Observable(const SpectralObject&);
   // forming the observable out of its spectral representation

Observable(Word const& w, State const& s);
   // for w="potential" we create an observable obs such that
   // obs(i,j)=s(i)*krn(i,j)
   // s also provided the dimension
   // for w="projector" we create the one-dimensional projection
   // operator corresponding to the argument state

Observable(const Word& w, Z d);
   // An appended F in names refers to 'finite difference'
   // (not 'Fourier'!)
   // for w="zero" or w="0",  zero operator = Observable(0.,d)
   // for w="unit" or w="1", unit operator = Observable(1.,d)
   // for w="position" or w="x", position operator
   // for w="space frequency" or w="f", space frequency operator F
   // for w="wave vector" or w="k",'wave vector' operator, K=2*pi*F
   //    equals the
   //    momentum operator in any system of units in which h_quer=1.
   // for w="momentum" or w="p", momentum operator P=hQuer*K=hPlanck*F
   // for w="momentumF" or w="pF", momentum operator as a finite
   // difference which approximates the derivative in position space.
   // pF and eKinF are the only arguments which are not based on
   // spectral representation and thus are not allowed in
   // as arguments in SpectralObject(Word,Z).
   // It turns out that major shifts generated by this kind of
   // momentum operator are accompanied by major distortions for all
   // but the smoothest wave functions.

   // for w="kinetic energy" or w="eKin", kinetic energy via
   // Fourier-defined P*P/(2*mass), where P is the Fourier-defined
   // momentum operator

   // for w="kinetic energy F" or w="eKinF", kinetic energy via
   // discrete Laplacian = Pr*Pl/(2*mass), where Pr is the right-
```

```
// difference operator and Pl the left-difference operator.

// for w="eKinFd", kinetic energy via just as "eKinF" but
// not with periodic boundary conditions as there but with
// wave function vanishing outside the box. This corresponds to
// the default continuation of arrays, thus 'd'.

// for w="boost" or w="n" generator of boosts N=mass*X
// (non-relativisic, see Habilschrift (2.11); see (2.25) for the
// relativistic generalization)

// For s:=QMBase::size we consider the interval [-s/2,s/2] and
// divide it into d equal subintervals (of length  delta=s/d). The
// coordinates of the centers of these subintervals are the d
// eigenvalues of our position operator. These thus range from
// -s/2+delta/2 to s/2-delta/2 with a spacing equal to delta. This
// implies that our position operator X is the natural
// discretization of  t h e  position operator for the interval
// [-s/2, s/2] in physical position space.
//
// The space frequency operator F, as defined here for w=="f",
// has a equidistant lattice of d eigenvalues with spacing f0=1/s.
// The maximum eigenvalue is fc=d/(2*s)=(d/2)*f0,a smallest
// non-zero,
//
// non-negative eigenvalue f0, followed by 0, -f0,..., -fc+f0
// Then we have exp(i*a*f0)=1 for a=2Pi*s and thus exp(i*a*F)=1.
// Since s is the periodicity of the base interval of our
// 'periodic position space' this corresponds to a shift by s or
// -s.
// Thus for the spatial translation operator T:
// T(-s)=exp(i*(2Pi*s)*F) and hence in general T(b)=exp(-i*2Pi*b*F).
//
// The normal quantum mechanical relation between spatial shift and
// Momentum operator is T(b)=exp(-ib*P/h_quer) (e.g. J.J.Sakurai:
// Modern Quantum Mechanics, p. 67). Therefore, P/h_quer=2Pi*F
// P/h = F.
// We therefore define the momentum operator P as h*F and the wave
// number operator K as 2Pi*F. Thus K = 2Pi*F = (2Pi/h)*P , hence
// P = hQuer*K as it has to be. Therefore, in any system of units
// hQuer=1, we have K=P. The space frequency operator is considered
// here only since it is closely related to the implementation
// of these operators in terms of the discrete (fast)
// Fourier transform. Thus for the definition of either of F,P,and
// K. d has to be a power of two.
// We expect [X,K] = i, the Heisenberg commutation
// relation, to hold 'as good as finite dimensional matrices
// can do it'. Actually they do very well. It is very instructive
// to inspect A = [K,X]*i which would be the identity operator
// if the Heisenberg commutation relation would hold exactly.
```

```
      // Actually most  of the eigenvalues of are numerically
      // indistiquishable from 1. But there are rather symmetrical
      // strong deviations at both ends of the spectrum.
      // When looking to the color coded matrix
      // elements (e.g. by the program CpmVQM2::App2 for builtX==4)
      // one would hardly see the close relationship of this matrix to
      // unity matrix. Notice also that this matrix puts a hard case
      // for spectral representation due to the obvious degeneration of
      // the spectrum.

   Observable(const Word& w, Z d, const R_Func& f);
      // meaning of w, d, s as above (except the case "d")
      // but directly creates a function of the operator. This avoids
      // double application of spectral decomposition.

   Observable(const Word& w, Z d, const F<R,C>& f);
      // Just as the last function, but the function is allowed to be
      // complex-valued in order to create e.g. unitary operators

// operators related to spin 1/2, thus d=2
   explicit Observable(const Word& w);
      // w="sx", "sy", "sz". Gives spin-1/2 operators for h=1
      // (not hQuer=1!).
      // w="sigma_x", "sigma_y", "sigma_z" Pauli matrices ( e.g. Landau,
      // Lifschitz, III (55.7)) sx=sigma_x/(4*Pi), ...

   Observable(const Angle& phi, const R3& n);
      // unitary representation operator belonging to the
      // spatial rotation described by angle phi and axis n
      // n has not to be normalized but only the unit vector counts.

   C operator()(Z i, Z j)const{ return rep[adjust(i)][adjust(j)];}
      // does not assume 0<=i,j<dim()

   C getLean(Z i, Z j)const
      { return rep.cui(i).cui(j);}
   // fast read access to matrix elements without index range check

   void setLean(Z i, Z j, const C& z)
      { rep.cui(i).cui(j)=z;}
   // fast write access to matrix elements without index range check

   void set(Z i, Z j, const C& z){ rep[adjust(i)][adjust(j)]=z;}

   void set(Z i, const State& s){ rep[adjust(i)]=s.rep_;}
      // a opservable can be considered as an array of rows, these rows
      // being instances of State. This function places s at row position
      // (starting with 0 !) i

   Observable symmetricPart(void)const
```

```
{ return (*this + (*this).con())*C(0.5);}

Observable unitaryVersion()const;
   // if *this is nearly unitary, the return value will be so
   // to much better approximation (not certain !!!!)

Observable cut()const;

Observable ranVal(Z j=0)const;

Z dim(void)const{return rep.dim();}

R autoStep(State const& x)const;

State operator *(State const& x)const;
   // Operator acts on state x

State operator()(State const& x)const{ return (*this)*x;}

SpectralObject spectralRepresentation(bool enfSym=false,
   bool test=false, Z iterMax=128, R eps=0,
   R eps1=1e-7, R eps2=1e-6)const;
   // returns a complete list of eigenvectors (second component of
   // return value together with a list of the corresponding
   // eigenvalues.
   // The algorithm works properly (and gives meaningfull results)
   // only if the observable *this is self-adjoint.
   // If the argument 'enforce symmetry' is true only
   // the self-adjoint part of it will be used for calculating
   // the spectral representation. If one can be sure that *this is
   // self-adjoint, one may use the argument.
   // The iterMax parameter limits the allowed iterations in a
   // subroutine which diagonalizes three-band matrices. Reaching the
   // limit stops the program with an error.
   // eps and n control phase adjustment of the eigenvectors to make
   // the visual representation of the eigenvectors regular
   // (see State::adjPhase(...))
   // The two last argument control the transition from the real form
   // to the complex one and should normally not need to be set
   // different from their default value. Only for consciouse
   // experimentation.

R_Matrix toRealForm()const;
   // returns a (2*d,2*d) matrix

void fromRealForm(const R_Matrix& r);
   // changes *this into the one resuling from the real matrix r

Observable operator!()const
   // pseudoinverse of *this based on corresponding function of real
```

```
   // form
{
   Observable res;
   res.fromRealForm(!toRealForm());
   return res;
}

Observable cayley()const;
   // returns the Cayley transform of *this

State expi(R t, State const& s, R eps=1e-6);
   // A.expi(t,s) is exp(i*t*A)*s according to the Chebyshev
   // single step method advocated by Hans De Raedt

void show(Frame& f, Z mult=-1, Z iShift=0, Z jShift=0)const;
   // one matrix element will be represented as a square of
   // mult*mult pixels. The whole image will be shifted by
   // (iShift,jShift) with respect to the left upper corner
   // of the graphics window f. ( i: vertical, j: horizontal
   // as in matrices)
   // If mult<0, then the pixel-size will be chosen such that
   // the frame f get filled as much as possible

void showImg(Graph& f, Z ipolMeth=1)const;
   // show by means of a C+- image class
   // similar to present method but makes automaticly
   // best use of size of f and interpolates bilinearly
   // for ipolMeth=1 (for 0 we use simple pixelation)
   // for ipolMeth=2 averaging interpolator (not yet
   // correctly working, but implementation OK in structure.

R sup()const;
   // maximum absolut value of a matrix element of *this

R normIter(R eps=1e-3)const;
   // Norm of the observable, computed by the power iteration method.
   // eps is a parameter which controls finishing the iteration.

R expVal(const State& a)
   { State a1=a; a1.normalize_(); return (a1|((*this)*a1))[1];}
   // expectation value determined by state and observable
   // a needs not to be normalized.

Type operator ||(const Type&)const;
   // tensor product; member function is preferred due to more
   // transparent behavior in the presence of namespaces

Type prdFock()const;
   // Returns the operator (*this)||unit + unit||(*this)
```

```
   Type prdFock2()const;
      // Returns the operator (*this)||(*this)

   static Type pot(R_Func const& f, Z d, R lambda=1.);
      // Defines an observable obs such that
      // obs(i,j)=f(positions(i)/size)*krn(i,j)*lambda.
      // So the function f is (as always in a similar context)
      // assumed to be meaningfully defined on Iv(-1/2,+1/2)
      // Since this observable is diagonal in position space,
      // it also could be obtained as a function of the position
      // operator

    static Type pot2(R_Func const& f, Z d, R lambda=1.);
      // This is a quite different concept of a potential
      // compared to the previous one. It is designed to
      // be a valid input to
      //    FieldState::cook2(Observable const&)
      // and
      //    FieldOpertor::cook2(Observable const&)
      // that then describes pair interaction by the same
      // potential which as pot(f,d,lambda) would describe
      // interaction with a fixed force center at the origin.
      // Notice that then the same function f had to be used
      // in pot(f,...) and potential(f,....) but that in the
      // latter case arguments of f in Iv(-1,1) will occur.
};

//////////////////// class SpectralMeasure ////////////////////////

// combines a set of real spectral values and a set of orthonormal
// set of projection operators into a single object which according to
// the spectral theorem results from spectral analysis of a self-adjoint
// operator.

   class SpectralMeasure{ // eigenvalues and spectral projectors
      Z d;
         // dim of statespace
      Z ds;
         // dim of spectrum
      Vl<R> spectrum;
      Vl<Observable> projectors;
   public:
      typedef SpectralMeasure Type;
      CPM_IO
      CPM_NAM(Type)
      Z dim()const{ return d;}
      Z dimSpc()const{ return ds;}
      const R& operator()(Z i)const{ return spectrum[i];}
         // eigenvalues by ()
      const Observable& operator[](Z i)const{ return projectors[i];}
```

```
         // eigenvectors by []

      SpectralMeasure():d(0),ds(0),spectrum(ds),projectors(ds){}

      explicit SpectralMeasure(const Observable&);
         // generation from a Observable

      explicit SpectralMeasure(const SpectralObject&);
         // and from a SpectralObject

      void map(const R_Func& f)
      {for (Z i=0;i<ds;i++) spectrum[i]=f(spectrum[i]);}

      R test()const;
         // returns a error sum which is large if the spectral data
         // are not correct

      State operator *(const State&)const;
      // SpectralMeasure acts on state just as the underlying observable

      pair<R,State> reduction(const State&)const;
      // performs the 'collapse of the wave packet' of the state
      // given by the argument and the observable of spectral
      // representation *this. Returned is the 'measured value' and the
      // state after measurement
   };

} // namespace
#endif
```

129 cpmobservable.cpp

```
//? cpmobservable.cpp
//? Status of work 2008-10-25.
//?
#include <cpmv.h>
#include <cpmobservable.h>
#include <cpmrmateig.h>
#include <cpmconstphys.h>
#include <cpmangle.h>
#include <cpmframes.h>
#include <cpmmultiimage.h>

using CpmQuantumMechanics::QMBase;
using CpmQuantumMechanics::State;
using CpmQuantumMechanics::Observable;
using CpmQuantumMechanics::SpectralObject;
using CpmQuantumMechanics::SpectralMeasure;

using namespace CpmRoot;
using namespace CpmArrays;
using namespace CpmFunctions;
using namespace CpmGraphics;

using CpmRootX::R1;
using CpmRootX::Record;
using CpmGeo::Angle;
using CpmLinAlg::R3;
using CpmLinAlg::Z2;
using CpmImaging::Color;
using CpmGraphics::Frames;

//////////////////////// class SpectralObject ////////////////////////////

SpectralObject::SpectralObject(Z d):d_(d),spectrum(d),eigenVectors(d)
{
   for (Z i=0;i<d_;i++){
      spectrum[i]=C(i);
      eigenVectors[i]=State(d_,i);
   }
}

SpectralObject::SpectralObject(const Word& w, Z d):
   d_(d),spectrum(d_),eigenVectors(d_)
{
   Z mL=2;
   Word loc( "SpectralObject(" );
   loc&=( w&", "&cpm(d_)&")" );
   CPM_MA
```

```
cpmassert(d_>0,loc);
Z i;
for (i=0;i<d_;i++) eigenVectors[i]=State(d_,i);
   // normalized states

if (w=="space frequency"||w=="f"){
   Vl<R> fr=frequencies(d_);
   for (i=0;i<d_;++i){
      spectrum[i]=C(fr[i]);
      eigenVectors[i]=eigenVectors[i].frmMomSpc();
   }
   normalize_();
}
else if (w=="momentum"||w=="p"){
   Vl<R> fr=frequencies(d_);
   for (i=0;i<d_;++i){
      spectrum[i]=C(fr[i]*hPlanck);
      eigenVectors[i]=eigenVectors[i].frmMomSpc();
   }
   normalize_();
}
else if (w=="wave vector"||w=="k"){
   Vl<R> fr=frequencies(d_);
   for (i=0;i<d_;++i){
      spectrum[i]=C(fr[i]*2*Pi);
      eigenVectors[i]=eigenVectors[i].frmMomSpc();
   }
   normalize_();
}
else if (w=="kinetic energy"||w=="eKin"){
   Vl<R> fr=frequencies(d_);
   R fac=1./(2*mass);
   for (i=0;i<d_;++i){
      R pi=fr[i]*hPlanck; // fr[i]=1/lambda
      spectrum[i]=C(pi*pi*fac);
      eigenVectors[i]=eigenVectors[i].frmMomSpc();
   }
   normalize_();
}
else if (w=="relativistic kinetic energy"||w=="eKinRel"){
   R e0=mass*c*c;
   Vl<R> fr=frequencies(d_);
   for (i=0;i<d_;++i){
      R si=fr[i]*hPlanck/c; // pi/c
      R vali=::sqrt(e0*e0-si*si)-e0;
      spectrum[i]=C(vali);
      eigenVectors[i]=eigenVectors[i].frmMomSpc();
   }
   normalize_();
}
```

```
   else if (w=="zero"||w=="0"){
      for (i=0;i<d_;i++) spectrum[i]=C(0);
   }
   else if (w=="unit"||w=="1"){
      for (i=0;i<d_;i++) spectrum[i]=C(1);
   }
   else if (w=="position"||w=="x"){
      Vl<R> pos=positions(d_);
      for (i=0;i<d_;i++){
         spectrum[i]=C(pos[i]);
      }
   }
   else if (w=="boost"||w=="n"){
      Vl<R> pos=positions(d_);
      for (i=0;i<d_;i++){
         spectrum[i]=C(pos[i]*mass);
      }
   }
   else{ // undefined
      cpmerror(loc&": unvalid value "&w&" for w");
   }
   if (cpmverbose>mL){
      ostringstream ost;
      ost<<"spectrum.first()="<<spectrum.first();
      ost<<"spectrum.last()="<<spectrum.last();
      cpmmessage(ost);
   }
   CPM_MZ
}

SpectralObject::SpectralObject(const Vl<R>& s, const Vl<State>& e):
         d_(s.dim()),spectrum(d_),eigenVectors(e)
{
   cpmassert(d_==e.dim(),"SpectralObject(...)");
   for (Z i=0;i<d_;i++) spectrum[i]=C(s[i],0);
}

State SpectralObject::operator *(const State& a)const
{
   State res=eigenVectors[0]*(spectrum[0]*(eigenVectors[0]|a));
   for (Z i=1;i<d_;i++){
      C ci=spectrum[i]*(eigenVectors[i]|a);
      res+=eigenVectors[i]*ci;
   }
   return res;
}

static void display(const State& s, Frame& f, Z iOffset, Z mult, Z d)
{
   Z i,j;
```

```
   Z d_=d*mult;
   Z i1=1+iOffset;
   Z i2=mult+iOffset;
   for (i=i1;i<=i2;i++) for (j=0;j<d_;j++)
      f.put(Z2(i,j),Color(C(s[j/mult])));
}

void SpectralObject::show(Frame& f, Z mult, Z is, Z js)const
{
   Z i,j;
   Z d=dim(),pix=mult;
   Z shiftH=2,gap=2;
   Z dTot=d+shiftH+gap;
   if (mult<0){
      Z nRow=f.getLin()-is;
      Z nCol=f.getCol()-js;
      Z nOK=cpminf(nCol,nRow);
      pix=nOK/(dTot);
      if (pix<1) pix=1;
   }
   State sp(d);
   for (i=0;i<d;i++){
      sp[i]=spectrum[i];
   }
   Z pixH=pix<3 ? 3 : pix;
   sp.show(f,pix,pixH,0,shiftH);
   Observable aux(d);
   for (i=0;i<d;i++){
      for (j=0;j<d;j++){
         C val=eigenVectors[i][j];
         aux.set(i,j,val);
      }
   }
   aux.show(f,pix,0,pixH+shiftH+gap);
}

 void SpectralObject::showSpectrum(Graph& gr)const
 {
   Z d=dim();
   R_Vector sp(d);
   for (Z i=0;i<d;i++){
      sp[i+1]=spectrum[i][1];
   }
   gr.show(sp,GREEN,1);
 }

R SpectralObject::test()const
{
   R_Matrix sp(d_);
   for (Z i=0;i<d_;i++){
```

```
      for (Z j=0;j<d_;j++){
         sp[i+1][j+1]=(eigenVectors[i]|eigenVectors[j]).absVal();
      }
   }
   R_Matrix u=sp.net(1);
   return (sp-u).absVal();
}

bool SpectralObject::writeSpectrum(ostream& str)const
{
   CpmRoot::writeTitle("Eigenvalues of a SpectralObject:",str);
//   cpmp(d);
   V<C> aux1(spectrum);
   cpmp(aux1);
   return true;
}

bool SpectralObject::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("a SpectralObject:",str);
   cpmp(d_);
   V<C> aux1(spectrum);
   cpmp(aux1);
   V<State> aux2(eigenVectors);
   cpmp(aux2);
   return true;
}

bool SpectralObject::scanFrom(istream& in)
{
   CpmRoot::read(d_,in);
   V<C> aux1;
   aux1.scanFrom(in);
   spectrum=aux1.c();
   V<State> aux2;
   aux2.scanFrom(in);
   eigenVectors=aux2.c();
   return true;
}

/*

pair<R,State> SpectralObject::reduction(const State& a)const
{
   V<C> w(d_);
   Z i;
   for (i=0;i<d_;i++){
      w[i+1]=(eigenVectors[i]|a);
   }
   C zRand;
```

```
   R r1=1.;
   R phi=CpmRoot::ranVal(r1); // not yet an arc, in inteval (-1,+1)
   phi*=Pi; // arc phi
   zRand.polar(1.,phi); // random selection of a complex number of
      // absolute value 1 ('random phase')
   Vo<R> sig(d_);
      // to be used for a test on probabilistic correctness
      // not yet implemented !!!!!!!!!!!!!!!!!!!!!
   V<Z> accum(d_,0);
   Z nTest=20000;
   R nTestInv=1./nTest;
   Z iRes;
   for (i=1;i<=nTest;i++){
      cpmprogress("Test-loop",i*nTestInv);
      phi=CpmRoot::ranVal(r1,i); // not yet an arc, in inteval
      // (-1,+1)
      phi*=Pi; // arc phi
      zRand.polar(r1,phi);
      for (Z j=1;j<=d_;j++){
         R valSig=(w[j]*zRand)[1];
         valSig=CpmRoot::absVal(valSig);
         sig[j]=valSig;
      }
      iRes=sig.indSup();
      accum[iRes]+=1;
   }
   R_Vector res(d_);
   for (i=1;i<=d_;i++){
      R valW=w[i].absVal();
      valW*=valW;
      res[i]=accum[i];
   }
   CpmGraphics::Graph g;
   g.show(res);
   iRes--;
   return pair<R,State>(spectrum[iRes][1],eigenVectors[iRes]);
}
*/

X2<R,State> SpectralObject::reduction(State const& a, R p)const
{
   V<C> w(d_);
   Z i;
   for (i=0;i<d_;i++){
      w[i+1]=(eigenVectors[i]|a);
   }
   C zRand;
   R r1=1.;
   R phi=CpmRoot::ranVal(r1); // not yet an arc, in inteval (-1,+1)
   phi*=Pi; // arc phi
```

```
   zRand.polar(1.,phi); // random selection of a complex number of
      // absolute value 1 ('random phase')
   Vo<R> sig(d_);
      // to be used for a test on probabilistic correctness
      // not yet implemented !!!!!!!!!!!!!!!!!!!!
   V<Z> accum(d_,0);
   Z nTest=20000;
   R nTestInv=1./nTest;
   Z iRes;
   for (i=1;i<=nTest;i++){
      cpmprogress("Test-loop",i*nTestInv);
      phi=CpmRoot::ranVal(r1,i); // not yet an arc, in inteval
      // (-1,+1)
      phi*=Pi; // arc phi
      zRand.polar(r1,phi);
      for (Z j=1;j<=d_;j++){
         R valSig=(w[j]*zRand)[1];
         valSig=CpmRoot::absVal(valSig);
         sig[j]=valSig;
      }
      iRes=sig.indSup();
      accum[iRes]+=1;
   }
   R_Vector res(d_);
   for (i=1;i<=d_;i++){
      R valW=w[i].absVal();
      valW*=valW;
      res[i]=accum[i];
   }
   CpmGraphics::Graph g;
   g.show(res);
   iRes--;
   return X2<R,State>(spectrum[iRes][1],eigenVectors[iRes]);
}

State SpectralObject::dePhase(const State& psi, R delta, R p, Z iter\
)const
// most advanced version that gives the orthodox probabilities
// for a stochatically varying phase delta
{
   const R inv2pi=0.5/cpmpi;
   bool useDegrees=false;
   CpmGeo::Angle phi(delta,useDegrees);
   R xi=(cpmpi+phi)*inv2pi;
      // quantity in range (0,1]
// quantities which describe the relation of state psi to observable
// *this
   Vl<C> c(d_);
   Vl<R> w(d_);
   Vl<R> a(d_);
```

```
   Z i,j;
   for (i=0;i<d_;i++){
      C psi_i=(eigenVectors[i]|psi);
      c[i]=psi_i;
      w[i]=psi_i.absSqr();
      a[i]=spectrum[i].real();
   }
// random decision for an eigenvalue of *this to be used for
// demodulation
   R sum=w[0];
   Z i0=0;
   Z dLim=d_-1;
   while(sum<xi && i0<dLim){
      i0++;
      sum+=w[i0];
   }
      // a[i0] is the selected eigenvalue
// de-phasing and averaging; makes the wave packet collapse
   for (i=0;i<d_;i++){
      C z=c[i];
      C ciSum=z;
      C ph=C::expi(p*(a[i]-a[i0]));
      for (j=1;j<=iter;j++){
         z*=ph;
         ciSum+=z;
      }
      c[i]=ciSum;
   }
   State res(d_);
   for (i=0;i<d_;i++) res+=eigenVectors[i]*c[i];
   res.normalize_();
   return res;
}

State SpectralObject::dePhase(const State& psi,R delta,
   const Vl<SpectralObject>& cwp,const Vl<R1>& p)
{
   const R inv2pi=0.5/cpmpi;
   Z iter=8;
      // corresponds to n (or n-1 ?) in the paper
   bool useDegrees=false;
   CpmGeo::Angle phi(delta,useDegrees);
   R xi=(cpmpi+phi)*inv2pi;
      // quantity in range (0,1]
// quantities which describe the relation of state psi to observable
// *this
   Z dc=cwp.dim();
   Vl<Z> selInd(dc);
   Z d=cwp[0].dim();
   State res=psi;
```

```
   Z h,i,j;
   for (h=0;h<dc;h++){
      if (h>0){
         Z d_=cwp[h].dim();
         if (d!=d_)
            cpmerror("SpectralObject::dePhase(...): dimension mismatch");\

      }
      Vl<C> c(d);
      Vl<R> w(d);
      Vl<R> a(d);
      for (i=0;i<d;i++){
         C psi_i=(cwp[h][i]|res);
         c[i]=psi_i;
         w[i]=psi_i.absSqr();
         a[i]=cwp[h](i).real();
      }
// random decision for an eigenvalue of *this to be used for
// demodulation
      R sum=w[0];
      Z i0=0;
      Z dLim=d-1;
      while(sum<xi && i0<dLim){
         i0++;
         sum+=w[i0];
      }
      selInd[h]=i0;

// a[i0] is the selected eigenvalue
// de-phasing and averaging makes the wave packet collapse
      for (i=0;i<d;i++){
         C z=c[i];
         C ciSum=z;
         C ph=C::expi(p[h]*(a[i]-a[i0]));
         for (j=1;j<=iter;j++){
            z*=ph;
            ciSum+=z;
         }
         c[i]=ciSum;
      }
      for (i=0;i<d;i++) res+=cwp[h][i]*c[i];
   }
   res.normalize_();
   return res;
}

State SpectralObject::represent(const State& psi)const
       // represents psi as a 'eigenfunction expansion with
       // respect to *this'
{
```

```
   State res(d_);
   for (Z i=0;i<d_;i++) res[i]=(eigenVectors[i]|psi);
   res.normalize_();
   return res;
}

namespace{

C f_exp(C const& z, R const& t){ return exp(C(0,t)*z) ;}

}

void  SpectralObject::expi(R t)
{
   F<C,C> f=F1<C,R,C>(t)(f_exp);
   this->map(f);
}

namespace{
   R makeReal(C const& z){return z.real();}
}

SpectralObject SpectralObject::order()const
         // returns a new SpectralObject which results form *this
         // by common re-ordering of eigenvalues and eigenvectors
         // such that the real parts of the eigenvalues are in increasing
         // order with the index
{
   F<C,R> transf(makeReal);
      // going for the real part
   Vl<R> spr=spectrum(transf);
      // making an array that holds the real parts
   V<R> spr2(spr);
   Vo<R> sp(spr2);
      // making an auxiliary object that
      // allows to create an ordering permutation
   V<Z> per=sp.permutationForIncreasingOrder();
      // making the ordering permutation
// rest is simple
   V<C> spectrumTemp(spectrum);
   V<State> eigenVectorsTemp(eigenVectors);
   spectrumTemp=spectrumTemp.permute(per);
   eigenVectorsTemp=eigenVectorsTemp.permute(per);
   Vl<C> spectrumRes=spectrumTemp.c();
   Vl<State> eigenVectorsRes=eigenVectorsTemp.c();
   return SpectralObject(spectrumRes,eigenVectorsRes);
}

///////////////////////// class Observable ////////////////////////////
```

```
Z Observable::adjust(Z i)const
{
   Z d=rep.dim();
   while(i>=d) i-=d;
   while(i<0) i+=d;
   return i;
}

Observable::Observable(const SpectralObject& p):
   rep(p.dim(),Vl<C>(p.dim()))
{
   Z i,j,k,d=p.dim();
   for (i=0;i<d;i++){
      for (j=0;j<d;j++){
         C sum;
         for (k=0;k<d;k++){
            sum+=p(k)*p[k][i].con()*p[k][j];
         }
         set(i,j,sum);
      }
   }
}

Observable::Observable(const Word& w, Z d)
{
   Z mL=3;
   Word loc( "Observable(" );
   loc&=( w&", "&cpm(d)&")" );
   CPM_MA
   if (w=="zero"||w=="0"){
      *this=Observable(0.,d);
   }
   else if (w=="unit"||w=="1"){
      *this=Observable(1.,d);
   }
   else if (w=="momentumF" || w=="pF"){
      Observable aux(0,d);
      R diff=2*size/d;
      C fac=-ihDirac*(1./diff);
      for (Z i=0;i<d;++i){
         aux.set(i-1,i,fac);
         aux.set(i+1,i,-fac);
      }
      *this=aux;
   }
   else if (w=="kinetic energy F" || w=="eKinF"){
      Observable aux(0,d);
      R diff=size/d;
      R fac1=hDirac/diff;
      R fac=-fac1*fac1/(2*mass);
```

```
      C co1(fac);
      C co2=co1*(-2);
      for (Z i=0;i<d;++i){
         aux.set(i-1,i,co1);
         aux.setLean(i,i,co2);
         aux.set(i+1,i,co1);
      }
      // notice that this defines a symmetric operator
      // as it stands
      *this=aux;
   }
   else if (w=="eKinFd"){
      Observable aux(0,d);
      R diff=size/d;
      R fac1=hDirac/diff;
      R fac=-fac1*fac1/(2*mass);
      C co1(fac);
      C co2=co1*(-2);
      for (Z i=0;i<d;++i){
         if (i-1>=0) aux.setLean(i-1,i,co1);
         aux.setLean(i,i,co2);
         if (i+1<d) aux.setLean(i+1,i,co1);
      }
      // notice that this defines a symmetric operator
      // as it stands
      *this=aux;
   }
   else{
      SpectralObject so(w,d);
      *this=Observable(so);
   }
   CPM_MZ
}

Observable::Observable(const Word& w, Z d, const R_Func& f)
{
   Z mL=2;
   Word loc("Observable(const Word& w, Z d, const R_Func& f)");
   cpmmessage(loc&" started");
   SpectralObject so(w,d);
   so.map_(f);
   cpmmessage(loc&" done");
   *this=Observable(so);
}

Observable::Observable(const Word& w):rep(2,Vl<C>(2))
// initialization thus as a 2x2 zero matrix
// See L.I.Schiff: quantum mechanics, 3rd edition, p. 206,
// eq. (27.30)
{
```

```
   using CpmRoot::Pi;
   const R fac=0.25/Pi;
   const C re=C(fac,0);
   const C im=C(0,fac);

   if (w=="sx"){
      rep[0][1]=re;
      rep[1][0]=re;
   }
   else if (w=="sy"){
      rep[0][1]=-im;
      rep[1][0]=im;
   }
   else if (w=="sz"){
      rep[0][0]=re;
      rep[1][1]=-re;
   }
   if (w=="sigma_x"){
      rep[0][1]=C(1.);
      rep[1][0]=C(1.);
   }
   else if (w=="sigma_y"){
      rep[0][1]=-C::I;
      rep[1][0]=C::I;
   }
   else if (w=="sigma_z"){
      rep[0][0]=C(1.);
      rep[1][1]=C(-1.);
   }
   else{ // undefined
      cpmerror("Observable(Word w): unvalid value for w");
   }
}

Observable::Observable(const Angle& phi, const R3& n)
// Landau Lifschitz III, (58.1-2)
{
   Angle phiHalf=phi*R(0.5);
   R c=cos(phiHalf);
   C s=sin(phiHalf)*C::I;
   R3 nn=n;
   nn.normalize();
   *this=Observable("unit",2)*c+Observable("sigma_x")*s*nn[1]+
      Observable("sigma_y")*s*nn[2]+Observable("sigma_z")*s*nn[3];
}

Observable::Observable(Word const& w, State const& st):
rep(st.dim(), Vl<C>(st.dim()))
{
   Z mL=3;
```

```
   Word loc("Observable(Word, State)");
   CPM_MA
   Z i,j,d=rep.dim();
   if (w=="potential"){
      for (i=0;i<d;++i){
         rep.cui(i).cui(i)=st(i);
      }
   }
   else if (w=="projector"){
      State x=st;
      C xi,xj;
      x.normalize_();
      for (i=0;i<d;i++){
         xi=x.rep_.cui(i);
         for (j=0;j<d;j++){
            xj=x.rep_.cui(j);
            rep.cui(i).cui(j)=xi*xj.con();
         }
      }
   }
   else{
      cpmerror(loc&":"&w&" not allowed as first argument");
   }
   CPM_MZ
}

Observable Observable::ranVal(Z j)const
{
   Z j0=j;
   Z d=dim();
   C z(1,1);
   Observable res(d);
   for (Z i=0;i<d;i++){
      for (Z j=0;j<d;j++){
         j0++;
         res.rep.cui(i).cui(j)=z.ranVal(j0);
      }
   }
   return res.symmetricPart();
}


Observable Observable::pot(R_Func const& f, Z d, R lambda)
{
   Vl<R> x0=posNor(d);
   Observable res(d);
   for (Z i=0;i<d;i++){
      res.rep.cui(i).cui(i)=C(f(x0[i])*lambda);
   }
   return res;
```

```
}

Observable Observable::pot2(R_Func const& f, Z d, R lambda)
{
   Vl<R> x0=posNor(d);
   Observable res(d);
   for (Z i=0;i<d;i++){
      for (Z j=0;j<d;j++){
         R dij=x0[i]-x0[j];
         if (dij>0.5) dij-=1;
         else if (dij<-0.5) dij+=1;
         else ;
         res.rep.cui(i).cui(j)=C(f(dij)*lambda);
      }
   }
   return res;
}

// operations which transform rows into columns

Observable Observable::transpose(void)const
// fast implementation without index bounds check
{
   Z d=dim();
   Observable res(d);
   for (Z i=0;i<d;i++){
      for (Z j=0;j<d;j++){
         res.rep.cui(i).cui(j)=
         rep.cui(j).cui(i);
      }
   }
   return res;
}

Observable Observable::con(void)const
// fast implementation without index bounds check
{
   Z d=dim();
   Observable res(d);
   for (Z i=0;i<dim();i++){
      for (Z j=0;j<dim();j++){
         res.rep.cui(i).cui(j)=
         rep.cui(j).cui(i).con();
      }
   }
   return res;
}

// product of observables
```

```
Observable Observable::operator *(const Observable& y)const
{
   Z d=dim();
   cpmassert(d==y.dim(),"CpmQuantumMechanics::operator *");
   Observable res(d);
   Observable z=y.transpose();
   for (Z i=0;i<d;i++){
      for (Z j=0;j<d;j++){
         res.rep.cui(i).cui(j)=
         rep.cui(i).fAcc2(
            QMBase::prodFunc,QMBase::accFunc,z.rep.cui(j));
      }
   }
   return res;
}

/*********** unsophisticated implementation for tests *******
Observable Observable::operator *(const Observable& y)const
{
   Z d=dim(),i,j,k;
   cpmassert(d==y.dim(),"CpmQuantumMechanics::operator *");
   Observable res(d);
   for (i=0;i<d;++i){
      for (j=0;j<d;++j){
         C z;
         for (k=0;k<d;++k){
            z+=rep[i][k]*y(k,j);
         }
         res.set(i,j,z);
      }
   }
   return res;
}
*/

Observable Observable::prodStar(const Observable& y)const
{

   Z d=dim();
   cpmassert(d==y.dim(),"CpmQuantumMechanics::prodStar(...)");
   Observable res(d);
   for (Z i=0;i<d;i++){
      for (Z j=0;j<d;j++){
         res.rep.cui(i).cui(j)=
         rep.cui(i).fAcc2(
            QMBase::prodFuncConj2,QMBase::accFunc,y.rep.cui(j));
      }
   }
   return res;
}
```

```
// linear algebra of observables

Observable Observable::operator -(void)const
{
   Z d=dim();
   Observable res(d);
   for (Z i=0;i<d;i++){
      res.rep.cui(i)=rep.cui(i).fa(negFunc);
   }
   return res;
}

Observable& Observable::operator +=(const Observable& x)
{
   Z d=dim();
   cpmassert(d==x.dim(),"");
   for (Z i=0;i<d;i++){
      rep.cui(i).ff_(sumFunc,x.rep.cui(i));
   }
   return *this;
}

Observable& Observable::operator -=(const Observable& x)
{
   Z d=dim();
   cpmassert(d==x.dim(),"");
   for (Z i=0;i<d;i++){
      rep.cui(i).ff_(diffFunc,x.rep.cui(i));
   }
   return *this;
}

Observable& Observable::operator *=(C z)
{
   Z d=dim();
   for (Z i=0;i<d;i++){
      rep.cui(i).fc_(prodFunc,z);
   }
   return *this;
}

Observable& Observable::operator *=(R r)
{
   Z d=dim();
   for (Z i=0;i<d;i++){
      rep.cui(i).fc_(prodFuncReal,C(r));
   }
   return *this;
}
```

```
Observable Observable::operator +(const Observable& y)const
{
   Z d=dim();
   cpmassert(d==y.dim(),"operator +");
   Observable res(*this);
   return res+=y;
}

Observable Observable::operator -(const Observable& y)const
{
   Z d=dim();
   cpmassert(d==y.dim(),"operator -");
   Observable res(*this);
   return res-=y;
}

Observable Observable::operator *(C z)const
{
   Observable res(*this);
   return res*=z;
}

Observable Observable::operator *(R r)const
{
   Observable res(*this);
   return res*=r;
}

// Hilbert-Schmidt structure

C Observable::operator |(const Observable& x2)const
{
   C res;
   Z d=dim();
   cpmassert(d==x2.dim(),"");
   for (Z i=0;i<d;i++){
      res+=rep.cui(i).fAcc2(
         QMBase::prodFuncConj1,QMBase::accFunc,x2.rep.cui(i));
   }
   return res;
}

State Observable::operator *(const State& a)const
{
   Z mL=4;
   Word loc("State Observable::operator *(const State& a)const");
   cpmmessage(mL,loc&" started");
   Z d=dim();
   cpmassert(d==a.dim(),loc);
```

```
   State res(d);
   for (Z i=0;i<d;i++){
      res.rep_.cui(i)=
         rep.cui(i).fAcc2(prodFunc,accFunc,a.rep_);
   }
   cpmmessage(mL,loc&" done");
   return res;
}

static State complexFormOf(const R_Vector x)
{
   Z i,d2=x.dim();
   Z d=d2/2;
   cpmassert(d*2==d2,"");
   State res(d);
   for (i=1;i<=d;i++) res[i-1]=C(x[i],x[i+d]);
   return res;
}

static bool C_dependent(const State& x, const State& y, R eps)
{
   State y1=y*C(0,1);
   if ((x-y1).norm()<eps ) return true;
   y1=y*C(0,-1);
   if ((x-y1).norm()<eps ) return true;
   return false;
}

R Observable::autoStep(State const& x1)const
{
   R n1=x1.norm();
   State x2=(*this)*x1;
   R n2=x2.norm()*hDiracInv;
   return step(n1,n2);
}

SpectralObject Observable::spectralRepresentation(bool enfSym,
   bool test, Z iterMax, R eps, R eps1, R eps2)const
// does not yet work for *this=p1*p1   !!!!!!!!!!!!!!!!!!!! (00-01-24)
// does work for *this=p1*p1 and for *this=i[x1,p1]  !! (00-01-29)
// at least up to d=128
// does work for *this=i[x1,p1] at least up to d=512 !! (2004-10-02)
// The particularity of the commutator is that so many of its
// eigenvalues are so precisely equal to 1, 'it is very degenerate'
// I never found spectral representation non-working during the last
// weeks
// when the Observable was Hermitean
{
   Z mL=2;
   Word loc("Observable::spectralRepresentation(bool,bool,Z,R,Z)");
```

```
if (QMBase::getVerbose()>1) cpmmessage(loc&": started");
CPM_MA
R t1=cpmtime();
Z i,j,d=dim();
R supObs=sup();
if (supObs==0){
   cpmwarning(loc&": trivial result returned for the zero-observable")\
   ;
   SpectralObject res(d);
   res.setSpectrum_(C(0));
   CPM_MZ
   return res;
}
R_Matrix realRep = enfSym ? symmetricPart().toRealForm() : toRealForm(\
);\

   // this brings in the data from rep
R_Matrix realEig;
bool test1=test;
cpmdebug(test1);
cpmdebug(iterMax);
R_Vector realSpec=eigen(realRep,realEig,test1,iterMax);
   // function eigen does most of the work
R_Vector realSpecPos=realSpec.posCopy();
   // replaces all components by their absolute value
R supSpc=realSpecPos.sup();
cpmassert(supSpc>0,loc);
eps1*=supSpc;
   // introduced 2004-09-25 to make the algorithm
   // work also if numbers are very small as when doing quantum
   // mechanics in SI-units
   // don't scale eps2 since it operates on a construct
   // of unit-vectors
Z dR=realSpec.dim();
cpmassert(dR==2*d,loc);
V<R> specPrel(dR);
V<State> eigPrel(dR);
for (i=1;i<=dR;i++){ // preliminary, since each eigenvector and
// spectral
//
   // point appears twice
   specPrel[i]=realSpec[i];
   eigPrel[i]=complexFormOf(realEig[i]);
}
Z iCurr=1;
V<R> spec(1);
V<State> arr(1);
V< V<State> > list(1,arr);
spec[iCurr]=specPrel[1];
list[iCurr][1]=eigPrel[1];
```

```
 for (i=2;i<=dR;i++){
    if(CpmRoot::absVal(specPrel[i]-spec[iCurr])<eps1){
       list[iCurr]&=eigPrel[i];
    }
    else{
       iCurr++;
       spec&=specPrel[i];
       list&=arr;
       list[iCurr][1]=eigPrel[i];
    }
}
Z dSp=spec.dim();
for (i=1;i<=dSp;i++) list[i]=State::E_Schmidt(list[i],eps2);
V<Z> degen(dSp,0);
for (i=1;i<=dSp;i++) degen[i]=list[i].dim();
Z zs=degen[1];
for (i=2;i<=dSp;i++) zs+=degen[i];
Z ic=0;
Vl<C> complexSpec(d);
Vl<State> complexEig(d);
for (i=1;i<=dSp;i++){
   C spi=C(spec[i]);
   for (j=1;j<=degen[i];j++){
      if (ic>d-1) break;
      complexSpec[ic]=spi;
      State s=list[i][j];
      s.adjPhase(eps);
      complexEig[ic]=s;
      ic++;
   }
}
    // logic not too easy
if (test){
    const R tWait=2;
    Observable obs=enfSym ? symmetricPart() : *this;
    R e1=0;
    for (i=0;i<d;i++){
       e1+=(obs*complexEig[i]-complexEig[i]*complexSpec[i]).norm();
    }
    cpmcerr<<endl<<"Observable::spectralRepresentation(....,test==true)\
     e1="<<e1;
    Observable v(d);
    for (i=0;i<d;i++) v.set(i,complexEig[i]);
    Observable vt=v.con();
    Observable unit("unit",d);
    Observable o1=v*vt-unit;
    Observable o2=vt*v-unit;
    R e2=o1.norm();
    R d2Inv=1./(d*d);
    R e3=o2.norm();
```

```
      e2*=d2Inv;
      e3*=d2Inv;
      cpmcerr<<endl<<"Observable::spectralRepresentation(....,test==true)\
 e2="<<e2;
      cpmcerr<<endl<<"Observable::spectralRepresentation(....,test==true)\
 e3="<<e3;
      cpmvalue("spec rep main error",e1);
      cpmwait(tWait);
      cpmvalue("spec rep ortho error",e2+e3);
      cpmwait(tWait);
   }
   R t2=cpmtime();
   R tCalc=t2-t1;
   cpmmessage(loc&"tCalc="&cpm(tCalc));
   CPM_MZ
   if (QMBase::getVerbose()>1) cpmmessage(loc&": done");
   return SpectralObject(complexSpec, complexEig);
}

R_Matrix Observable::toRealForm()const
      // returns a real (2*d,2*d) matrix
{
   Z i,j,i2,j2,d=dim();
   Z d2=2*d;
   R_Matrix res(d2,d2);
   C val;
   R valr,vali;
   for (i=1;i<=d;i++){
      i2=i+d;
      for (j=1;j<=d;j++){
         j2=j+d;
         val=rep.cui(i-1).cui(j-1);
         valr=val[1];
         vali=val[2];
         res[i][j]=valr;
         res[i2][j2]=valr;
         res[i2][j]=vali;
         res[i][j2]=-vali;
      }
   }
   return res;
}

void Observable::fromRealForm(const R_Matrix& r)
      // changes *this into the one resuling from the real matrix r
{
   Z i,j,i2,d2=r.dim1();
   cpmassert(r.dim2()==d2,"");
   Z d=d2/2;
   cpmassert(2*d==d2,"");
```

```
   rep=Vl< Vl<C> >(d, Vl<C>(d));
   C val;
   R valr,vali;
   for (i=1;i<=d;i++){
      i2=i+d;
      for (j=1;j<=d;j++){
         valr=r[i][j];
         vali=r[i2][j];
         val=C(valr,vali);
         rep.cui(i-1).cui(j-1)=val;
      }
   }
}

void Observable::show(Frame& f, Z mult, Z is, Z js)const
{
   Z d=dim(),pix=mult;
   if (mult<0){
      Z nRow=f.getLin()-is;
      Z nCol=f.getCol()-js;
      Z nOK=cpminf(nCol,nRow);
      pix=nOK/d;
      if (pix<1) pix=1;
   }

   R valMax=sup();
   if (valMax==0) valMax=1.; // zero observable should be representable
   R valMaxInv=1./valMax;
   Z i,j;
   Z d_=d*pix;
   for (i=0;i<d_;i++){
      Z iEff=i/pix;
      for (j=0;j<d_;j++){
         C val=rep.cui(iEff).cui(j/pix);
         val*=valMaxInv;
         f.put(Z2(i+1+is,j+1+js),Color(val));
            // put replaced by putSave 2004-09-21
      }
   }
   f.display(); // added 2005-04-15
}

void Observable::showImg(Graph& f, Z ipolMeth)const
{
   Z d=dim();
   VVV<R> vvvr(3,d,d);
   for (Z i=1;i<=d;++i){
      for (Z j=1;j<=d;++j){
         C val=rep.cui(i-1).cui(j-1);
         Color col(val);
```

```
         vvvr[1][i][j]=col.Red();
         vvvr[2][i][j]=col.Green();
         vvvr[3][i][j]=col.Blue();
            // notice that the channel index is the first one
      }
   }
   CpmImaging::MultiImage<R> mi(vvvr);
   mi.mark(f,ipolMeth); // nice service of class MultiImage
   // iPolMeth==2 not yet working
}

R Observable::sup()const
{
   R res=-1;
   Z d=dim();
   for (Z i=0;i<d;i++) for (Z j=0;j<d;j++){
      R r=rep.cui(i).cui(j).absVal();
      if (r>res) res=r;
   }
   return res;
}

Observable Observable::cayley()const
{
   Z d=dim();
   Observable res("unit",d); // unit matrix
   res=(res+(*this))*!(res-(*this))*C(0,1);
      // although I expected this to be selfadjoint anyway this is not the
      // case
   return res.symmetricPart();
}

// trying to improve unitarity: does not work so far
static Observable iter(const Observable& x)
{ return (x.con()+!x)*C(0.5);}

Observable Observable::unitaryVersion()const
      // if *this is nearly unitary, the return value should be so
      // to much better approximation. Does not work that way, however
{
   Observable res=iter(*this);
   res=iter(res);
   return (*this+res)*C(0.5);
}

Observable Observable::cut()const
{
   Observable res(*this);
   Z iMax=res.dim()-1;
   Z iMin=0;
```

```
   C z;
   res.set(iMin,iMax,z);
   res.set(iMax,iMin,z);
   return res;
}

bool Observable::prnOn(ostream& out)const
{
   Z d=dim();
   out<<endl<<"// an Observable:";
   CpmRoot::write(d,out);
   for (Z i=0;i<d;i++)
      for (Z j=0;j<d;j++)
         getLean(i,j).prnOn(out);
   return true;
}

bool Observable::scanFrom(istream& in)
{
   Z d=dim();
   CpmRoot::read(d,in);
   rep=Vl< Vl<C> >(d,Vl<C>(d));
   C val;
   for (Z i=0;i<d;i++){
      for (Z j=0;j<d;j++){
         val.scanFrom(in);
         setLean(i,j,val);
      }
   }
   return true;
}

Observable Observable::operator ||(const Observable& s2)const
{
   Z d1=dim();
   Z d2=s2.dim();
   Observable res(d1*d2);
   Z i1,i2,j1,j2,k,l;
   k=0;
   for (i1=0;i1<d1;i1++){
      for (i2=0;i2<d2;i2++){
         l=0;
         for (j1=0;j1<d1;j1++){
            for (j2=0;j2<d2;j2++){
               C val=getLean(i1,j1)*s2.getLean(i2,j2);
               res.setLean(k,l,val);
               l++;
            }
         }
         k++;
```

```
      }
   }
   return res;
}

Observable Observable::prdFock( )const
{
   Z d=dim();
   Observable unit("unit",d);
   Observable ob1=(*this)|| unit;
   Observable ob2=unit||(*this);
   return ob1+ob2;
}

Observable Observable::prdFock2( )const
{
   return (*this)||(*this);
}

R Observable::normIter(R eps)const
{
   static Word loc("Observable::normIter(R)");
   Z mL=3;
   CPM_MA
   static const Z iterMax=100;
   Z n=dim();
   State st0(n);
   State st=st0.ranVal();
   R res=-1e20;
   bool work=true;
   Z iter=0;
   while (work){
      iter++;
      st=(*this)*st;
      R resNew=st.normalize_();
      R diff=disVal(res,resNew);
      if (diff<eps){
         work=false;
         res=0.5*(res+resNew);
      }
      else if (iter>iterMax){
         work=false;
         cpmerror(loc&": stopped after "&cpm(iter)&" iterations");
         res=resNew;
      }
      else{
         res=resNew;
      }
   }
   CPM_MZ
```

```
   return res;
}

State Observable::expi(R t, State const& s, R eps)
      // A.expi(t,s) is exp(i*t*A)*s according to the Chebyshev
      // single step method advocated by Hans De Raedt
{

   R nor1=normIter();
   R security=1.2;
   nor1*=security;
   R nor1Inv=cpminv(nor1);
   // assumes that *this is symmetric
   Observable skew=(*this)*(C::I*nor1Inv);
   // now skew is skew-symmetric and has norm<1
   Z kMax=(Z)(exp(1.)*0.5*nor1)+1;
   R z=t*nor1;
   State sum=s*besselJ(0,z);
   State sp=s; // previous
   State sa=skew*s; // actual
   for (Z k=1;k<=kMax;++k){
      sum+=sa*(besselJ(k,z)*2);
      State temp=sa;
      sa=(skew*temp)*2+sp;
      sp=temp;
   }
   return sum;
}

//////////////////////// class SpectralMeasure ////////////////////////

SpectralMeasure::SpectralMeasure(const SpectralObject& spo):d(spo.dim())
{
   static const R epsRel=1e-6;
      // discrimination accuracy for spectral values

   if (QMBase::getVerbose()>1)
      cpmmessage("SpectralMeasure(const SpectralObject&) started");

   cpmassert(d==spo.dim(),"");
   Vl<R> v(d);
   Vl<Observable> vobs(d);
   Word w("projector");
   Z i;
   for (i=0;i<d;i++) v[i]=spo(i)[1]; // spectrum is real anyway
   for (i=0;i<d;i++) vobs[i]=Observable(w,spo[i]); // spectrum is real
   // anyway
   Observable sumO(d);
   for (i=0;i<d;i++) sumO+=vobs[i];
   Observable unit("unit",d);
```

```
   cpmcerr<<endl<<"SpectralMeasure(const Observable&) test: err= "
      <<(sumO-unit).norm();
   R eps=(v[0]-v[d-1])*epsRel;
   Vl<R> sp(d);
   Observable zero(d);
   Vl<Observable> pr(d,zero);
   sp[0]=v[0];
   pr[0]=Observable(w,spo[0]);
   Z iAct=0;
   cpmassert(d>1,"");
   for (i=1;i<d;i++){
      if (CpmRoot::absVal(v[i]-v[iAct])<eps){
         pr[iAct]+=Observable(w,spo[i]);
      }
      else{
         iAct++;
         sp[iAct]=v[i];
         pr[iAct]=Observable(w,spo[i]);
      }
   }
   ds=iAct+1;
   spectrum=Vl<R>(ds);
   projectors=Vl<Observable>(ds);
   for (i=0;i<ds;i++){
      spectrum[i]=sp[i];
      projectors[i]=pr[i];
   }
   if (QMBase::getVerbose()>1)
      cpmmessage("SpectralMeasure(const SpectralObject&) done");
}

SpectralMeasure::SpectralMeasure(const Observable& obs):d(obs.dim())
{
   static const R epsRel=1e-6;
      // discrimination accuracy for spectral values

   if (QMBase::getVerbose()>1)
      cpmmessage("SpectralMeasure(const Observable&) started");

   SpectralObject spo=obs.spectralRepresentation(false,false);
   cpmassert(d==spo.dim(),"");
   Vl<R> v(d);
   Vl<Observable> vobs(d);
   Z i;
   Word w("projector");
   for (i=0;i<d;i++) v[i]=spo(i)[1]; // spectrum is real anyway
   for (i=0;i<d;i++) vobs[i]=Observable(w,spo[i]); // spectrum is real
   // anyway
   Observable sumO(d);
   for (i=0;i<d;i++) sumO+=vobs[i];
```

```
   Observable unit("unit",d);
   cpmcerr<<endl<<"SpectralMeasure(const Observable&) test: err= "
      <<(sumO-unit).norm();
   R eps=(v[0]-v[d-1])*epsRel;
   Vl<R> sp(d);
   Observable zero(d);
   Vl<Observable> pr(d,zero);
   sp[0]=v[0];
   pr[0]=Observable(w,spo[0]);
   Z iAct=0;
   cpmassert(d>1,"");
   for (i=1;i<d;i++){
      if (CpmRoot::absVal(v[i]-v[iAct])<eps){
         pr[iAct]+=Observable(w,spo[i]);
      }
      else{
         iAct++;
         sp[iAct]=v[i];
         pr[iAct]=Observable(w,spo[i]);
      }
   }
   ds=iAct+1;
   spectrum=Vl<R>(ds);
   projectors=Vl<Observable>(ds);
   for (i=0;i<ds;i++){
      spectrum[i]=sp[i];
      projectors[i]=pr[i];
   }
   if (QMBase::getVerbose()>1)
      cpmmessage("SpectralMeasure(Observable&) done");
}

pair<R,State> SpectralMeasure::reduction(const State& a)const
      // performs the 'collapse of the wave packet' of the state
      // given by the argument and the observable of spectral
      // representation *this. Returned is the 'measured value' and the
      // state after measurement
{

   if (QMBase::getVerbose()>2)
      cpmmessage("SpectralMeasure::reduction(...) started");

   State psi=a;
   psi.normalize_();
   Vl<R> w(ds);
   Z i;
   for (i=0;i<ds;i++){
      C cVal=(a|(projectors[i]*a)); // is real and positive anyway
      R rVal=cVal.absVal(); // makes this sure even in the presence
         // of numerical errors
```

```
      w[i]=rVal;
   }
   R accu=0;
   Vo<R> ws(ds+1); // class provides function locate
   for (i=1;i<=ds;i++){
      ws[i]=accu;
      accu+=w[i-1];
      if (accu>1.) accu=1; // should not ocurr
   }
   ws[ds+1]=1.;
      // now the interval [0,1] is subdivided into ds intervals of length
      // w[i]
   R wr=CpmRoot::ranVal(R(0.5))+R(0.5); // random number ranges from 0
   // to 1
   Z iRes=ws.locate(wr); // random selection of a interval
      // with probability equal to its lenght
   iRes--;
   R value=spectrum[iRes];
   State st=projectors[iRes]*psi;
   st.normalize_();
   if (QMBase::getVerbose()>2)
      cpmmessage("SpectralMeasure::reduction(...) done");
   return pair<R,State>(value,st);
}

State SpectralMeasure::operator *(const State& a)const
{
   State res=(projectors[0]*a)*spectrum[0];
   for (Z i=1;i<ds;i++){
      res+=(projectors[i]*a)*spectrum[i];
   }
   return res;
}

bool SpectralMeasure::prnOn(ostream& out)const
{
   cpmmessage("SpectralMeasure::prnOn(...) started");
   using CpmRoot::write;
   out<<endl<<"// a SpectralMeasure:";
   write(d,out);
   write(ds,out);
   V<R> aux1(spectrum);
   aux1.prnOn(out);
   for (Z j=0;j<ds;j++) projectors[j].prnOn(out);
   cpmmessage("SpectralMeasure::prnOn(...) done");
   return true;
}

bool SpectralMeasure::scanFrom(istream& in)
{
```

```
   cpmmessage("SpectralMeasure::scanFrom(...) started");
   using CpmRoot::read;
   read(d,in);
   read(ds,in);
   V<R> aux1;
   aux1.scanFrom(in);
   spectrum=aux1.c();
   projectors=Vl<Observable>(ds);
   for (Z j=0;j<ds;j++) projectors[j].scanFrom(in);
      // it is much faster to scan the components directly into
      // projectors, as done here, instead of scanning first into aux2
      // and then copying into projectors. Brings several hundered
      // percent in performance !!!!!
   cpmmessage("SpectralMeasure::scanFrom(...) done");
   return true;
}

R SpectralMeasure::test()const
{
   Observable sum(d); // default constructor makes zero observable
   for (Z i=0;i<ds;i++) sum+=projectors[i];
   Observable unit("unit",d);
   R err1=(sum-unit).norm();
   return err1;
}

///////////////////////// class State ////////////////////////////////////

C fExp(const R& t, const R& a)
{
   R x=a*t;
   return C(::cos(x),::sin(x));
}

State State::shift(R sf)const
// sign changed after test: 2004-10-13
{
   Z d=dim();
   bool b=Z1(d).isPowOfTwo();
   if (b){
      SpectralObject so("p",d);
      F<R,C> fsh=F1<R,R,C>(sf*hDiracInv)(fExp);
      so.map(fsh);
      return so*(*this);
   }
   else{
      Z sd=cpmround((sf/size)*d);
      State res(d);
      for (Z i=0;i<d;++i) res[i]=(*this)[i-sd];
         // safe indexing does periodic continuation
```

```
        return res;
    }
}
```

130 cpmoperator.h

```
//? cpmoperator.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_OPERATOR_H_
#define CPM_OPERATOR_H_
/*
   Description: objects for describing quantum mechanical operators
      as time-dependent functions from Space to Space
*/

#include <cpmtrj.h>

namespace CpmQuantumMechanics{

   using namespace CpmArrays;
   using namespace CpmFunctions;
   using namespace CpmGraphics;
   using CpmRootX::B;
   using CpmFunctions::R_Func;

//////////////////////// class Operator ////////////////////////

class Operator: public QMBase{ // observables as functions
   typedef Operator Type;
   typedef X2<State,R> XSR;
   Z d_;
   F< XSR ,State> f_;

   static State conInTime(XSR const& xsr, F<State,State> const& f)
   { return f(xsr.c1());}

   static State mul2(XSR const& xsr, Operator const& op1,
      Operator const& op2)
   {  return op1.f_(XSR(op2.f_(xsr),xsr.c2()));}

   static State add2(XSR const& xsr, Operator const& op1,
      Operator const& op2)
   { return op1.f_(xsr)+op2.f_(xsr);}

   static State dif2(XSR const& xsr, Operator const& op1,
      Operator const& op2)
   { return op1.f_(xsr)-op2.f_(xsr);}

   static State mulC(XSR const& xsr, Operator const& op, C const& z)
   { return op.f_(xsr)*z;}

   static State mulR(XSR const& xsr, Operator const& op, R const& r)
```

```
   {  return op.f_(xsr)*r;}

   Operator( Z d, F<XSR,State> const& f):d_(d),f_(f){}
   Operator(Z d, F<State,State> const& f);

public:

   CPM_PURE_VECTOR_SPACE_INTERFACE(Type,C)
   CPM_PURE_SCALAR_INTERFACE(Type,R)
   CPM_NAM(Operator)

   Operator operator*(Operator const& op)const;

   Operator(){}

   explicit Operator(Observable const& ob);

   State actOn(State const& st, R t)const{ return f_(XSR(st,t));}
      // Operator acts on state at time t

   State operator()(State const& st, R t=0)const
      // Operator acts on state at time t
      { return f_(XSR(st,t));}

   State operator*(XSR const& xsr)const{ return f_(xsr);}
      // another form of actOn;

   Observable toObservable(R t)const;

   static Operator sinParHam(R2_Func const& pot,Z d, R lambda=1.);

   static Operator freePar(Z d);
};

////////////////////// class TrjDMI_Opr ////////////////////////////////
// trajectory for time-dependent Hamiltonians.
// similar to TrjDMI but replaces Observables by
// Operators (in the definition of which time-dependence
// is allowed) and which are much more efficient in higher-
// dimensional spaces. Also, the dimension of the space
// needs not to be a power of two

class TrjDMI_Opr: public TrjDMI<State>{ // trajectories det. by operators
   Operator opr_;
   virtual State op(State const& x, R t)const{ return opr_.actOn(x,t);}
public:
   typedef TrjDMI_Opr Type;
   virtual TrjBase* clone()const{ return new TrjDMI_Opr(*this);}
   TrjDMI_Opr(){}
   TrjDMI_Opr(Operator const& opr, State const& x, TrjCon const& tc):
```

```
   TrjDMI<State>(x,tc),opr_(opr){}
   virtual Word text()const{ return Word("TrjDMI_Opr");}
};

} // namespace

#endif
```

131 cpmoperator.cpp

```
//? cpmoperator.cpp
//? Status of work 2008-10-25.
//?
#include <cpmoperator.h>

using CpmQuantumMechanics::QMBase;
using CpmQuantumMechanics::State;
using CpmQuantumMechanics::Operator;
using CpmQuantumMechanics::Observable;
using CpmQuantumMechanics::TrjCon;
using CpmQuantumMechanics::TrjB;
using CpmFunctions::F1;
using CpmFunctions::F2;
using CpmFunctions::R_Func;
using CpmFunctions::R2_Func;

//////////////////////// class Operator //////////////////////////

namespace{

   State xsrobs(X2<State,R> const& xsr, Observable const& ob)
   {
      return ob*xsr.c1();
   }

#define CPM_SC\
      static R hD=QMBase::get_hDirac();\
      static R hD2=hD*hD;\
      State psi=xsr.c1();\
      R t=xsr.c2();\
      Z d=psi.dim();\
      R dx=1./d;\
      R hInv2=d/QMBase::getSize();\
      hInv2*=hInv2;\
      R fac=-hD2*hInv2/(2*QMBase::getMass());\
      State res(d);\
      R xi=-0.5+0.5*dx;

   State xsrrfs(X2<State,R> const& xsr, R_Func const& pot)
   {
      CPM_SC
      for (Z i=0;i<d;++i){
         C val=psi[i-1]-psi[i]*2+psi[i+1];
         res[i]=val*fac+pot(xi)*psi[i];
         xi+=dx;
      }
      return res;
```

```
   }

   State xsrrfs2(X2<State,R> const& xsr, R2_Func const& pot)
   {
      CPM_SC
      for (Z i=0;i<d;++i){
         C val=psi[i-1]-psi[i]*2+psi[i+1];
         res[i]=val*fac+pot(C(xi,t))*psi[i];
         xi+=dx;
      }
      return res;
   }

#undef CPM_SC

}// namespace

Operator::Operator(Observable const& ob):d_(ob.dim())
{
   f_=F1<XSR,Observable,State>(ob)(xsrobs);
}

namespace{
   State sinParHamFunc(X2<State,R> const& st, R2_Func const& pot,
      R const& lambda)
   { return st.c1().sinParHam(pot,st.c2(),lambda);}

   State freeParFunc(State const& st)
   { return st.freeParHam();}
}

Operator Operator::sinParHam(R2_Func const& pot,Z d, R lambda)
{
   return
   Operator(d,F2<XSR,R2_Func,R,State>(pot,lambda)(sinParHamFunc));
}

Operator Operator::freePar(Z d)
{
   return Operator(d,F<State,State>(freeParFunc));
}

Operator::Operator(Z d, F<State,State> const& f):d_(d)
{
   f_=F1<XSR,F<State,State>,State>(f)(conInTime);
}

Operator Operator::operator*(Operator const& op)const
{
   return
```

```
   Operator(d_,F2<XSR,Operator,Operator,State>(*this,op)(mul2));
}

Operator Operator::operator+(Operator const& op)const
{
   return
   Operator(d_,F2<XSR,Operator,Operator,State>(*this,op)(add2));
}

Operator Operator::operator-(Operator const& op)const
{
   return
   Operator(d_,F2<XSR,Operator,Operator,State>(*this,op)(dif2));
}

Operator Operator::operator*(C z)const
{
   return
   Operator(d_,F2<XSR,Operator,C,State>(*this,z)(mulC));
}

Operator Operator::operator*(R r)const
{
   return
   Operator(d_,F2<XSR,Operator,R,State>(*this,r)(mulR));
}

Operator& Operator::operator+=(Operator const& fop)
{
   return *this=operator+(fop);
}

Operator& Operator::operator-=(Operator const& fop)
{
   return *this=operator-(fop);
}

Operator& Operator::operator*=(C z)
{
   return *this=operator*(z);
}

Operator& Operator::operator*=(R r)
{
   return *this=operator*(r);
}

Observable Operator::toObservable(R t)const
{
   Observable res(d_);
```

```
   for (Z i=0;i<d_;++i){
      State vsi= actOn(State(d_,i),t);
      res.set(i,vsi);
   }
   return res;
}

//////////////////////// class TrjTD ////////////////////////////////
/*
void TrjTD::adjustTD_()
{
   State opx=op_.actOn(x_,t_);
   enr_=(x_|opx);
   v_=opx*QMBase::get_ihDiracInv();
   R n1=x_.norm();
   n_1_=n1-1;
   R n2=v_.norm();
   dt_=QMBase::step(n1,n2);
   text_="Evolution by the Time Dependent Direct Midpoint Integrator";
}

C TrjTD::eTot()const
{
   return (x_|v_)*QMBase::get_ihDirac();
}

TrjTD::TrjTD(const Operator& a, const State& x, TrjCon const& tc):
TrjB(x,tc),op_(a)
{
   adjustTD_();
}

void TrjTD::step_(R h)
{
   static C fac=QMBase::get_ihDiracInv()*QMBase::get_ihDiracInv();

   if (autoStep()){ // due to time dependence of
      // the generator, the proposed time step will vary with time
      // and has to be computed for each step (rather cheaply)
      R n1=x_.norm();
      R n2=v_.norm();
      dt_=QMBase::step(n1,n2);
   }
   else dt_=h;

   R tau=0.5*dt_;
   R ti=t_+tau;
   State xi=x_+v_*tau;
   State yi=op_.actOn(xi,ti);
   State dv=op_.actOn(yi,ti)*(fac*dt_);
```

```
   x_+=(v_*dt_+dv*tau);
   v_+=dv;
   t_+=dt_;
   stepCon_+=1;
   norIfEnf_();
   enr_=eTot();
}

bool TrjTD::prnOn(ostream& str)const
{
   cpmwt("TrjTD");
   if (! TrjB::prnOn(str)) return false;
//   cpmp(op_);
   cpmp(v_);
   return true;
}

bool TrjTD::scanFrom(istream& str)
{
   if (! TrjB::scanFrom(str)) return false;
//   cpms(op_);
   cpms(v_);
   return true;
}
*/
```

132 cpmp.h

```
//? cpmp.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_P_H_
#define CPM_P_H_
/*

Purpose: defining classes of smart pointers, and polymorphic arrays.

*/

#include <cpmv.h>

namespace CpmArrays{

///////////////////////  class Pp< >  ////////////////////////////////////
//
// class of non-constant smart pointers implementing 'copy on write' (see
// cpmv.h) and polymorphism (indicated by the 'p'). Let us first discuss
// the 'mono-morphic' aspects, i.e. we discuss the properties of Pp<T>
// for arbitrary but fixed T and not for a variety of T's:
//
// Let T be any non-abstact class that defines a member function
//
//       Tb* T::clone(void)const{ return new T(*this);}  (ii)
// (Notice that T has to define a copy constructor for this to work)
//
// where Tb is either T itself or a base class of T which declares
//
//       virtual *Tb Tb::clone(void)const{ return new Tb(*this);}
// or
//       virtual *Tb Tb::clone(void)const=0;
//
// Then, let us say that T is cloneable, with clone base Tb. Then the
// main property of the Pb<> template is:
//
//    cloneable T ==> value class Pb<T>
//
// If the source code of a non-abstract T is open to modifications, it
// can be made cloneable by inserting the definition
//    T* T::clone()const{ return new T(*this);}
// (lets ignore the complication that T may already use the name
//  clone in a conflicting manner).
//
// Properties of Pp<T> as a 'polymorphic container':
// -------------------------------------------------
//
```

```
// We consider a finite set L of cloneable classes which all have the
// same clone base T0. Let T1 be one of those and let L(T1) denote the
// classes in L which are derived from T1. Then Pp<T1> is a polymorphic
// container for L(T1) in the following sense:
// Let pp be an instance of Pp<T1>, created e.g. as
//    Pp<T1> pp;
// and let T2 be a class belonging to L(T1), and let t2 be an instance of
// T2.
//
// Then t2 can be put into the container pp by the
// statement
//
//       pp=t2; (or also pp.set(t2);)
//
// Now pp carries just the information that was carried before (and is
// carried still) by t2.
// If t2 gets changed or destroyed (e.g. as a local variable going out of
// scope) pp, will conserve the information and can return it in two
// ways.
// 1. Reconstruction as object: T2 t2_=pp(T2()) will create an object t2_
//    which is equivalent to t2.
// 2. Reconstruction by action: Let
//
//       virtual void T1::show(Media&)const;
//
// be a function which creates a detailed textual or graphical
// description of the calling instance by interacting with a class Media
// that provides the basis for that. Then 'Media m; pp().show(m);' will
// leave m in a state that carries the same record that we would get from
// 'Media m; t2.show(m)'. Technically, pp() is a quantity that is typed
// T1&; the object it refers to is of type T2 however.
// Let T3 be another class derived from T1, then pp, after pp.set(t3)
// behaves as T3 object. Thus pp behaves changing its type depending on
// initialization.
// Pp<> is closely related to the surrogate classes as discussed in
// Andrew Koenig: Ruminations on C++, AT&T 1997 Chapter 5. As stated
// there, the main benefit of such a 'polymorphic class' is that its
// instances can be put into a strictly typed container. This will be
// carried out in defining the extremely useful class Vp<>.

template <class T>

class  Pp: public P<T>{ // polymorphic smart pointers
   typedef P<T> Base;
public:
    Pp(T* p_=0):P<T>(p_){}
      // carrying over the constructor from P. Notice that the
      // primary act of construction was to initialize p_. This can be by
      // T* p_=new Td(....) where Td is some derived class (or T itself)
```

```
 explicit Pp(T const& t):P<T>(static_cast<T*>(t.clone())){}
 // 2004-09-27: finally a pointer-less interface of Pp<> enabled!
 // The queer casting is needed in the general situation that
 // T0 and T1 are different (in the description above). This general
 // situation is probably not really important (I had it
 // unintentionally for T0=RigidObject and T1=ExtSys, but
 // simplified it to T0=T1=ExtSys without encountering problems)
 // Notice, however, that t.clone() involves call to the copy
 // constructor of T which might be expensive.

 Pp(const P<T>& x):P<T>(x){}
   // down-cast constructor

T& operator()(void){ return *Base::p;}
   // Pp<T> pp;
   // T t;
   // pp=t;
   // makes t.aVirtualFunction(...);
   // and
   // pp().aVirtualFunction(...);
   // calling the same function with the same data.

T const& operator()(void)const{ return *Base::p;}
   // not clear why the MS-compiler cannot infer this definition from
   // the base class.
   // Pp<T> pp;
   // T t=pp()
   // is the usual way to retreive the value of pp. If the value is
   // known to belong to a derived class Td, use the next function:
   //    Td t=pp(Td()).

template <class X>
   X operator()(X const& x)const
   // retreiving an element from the container as a copy of a
   // prescribed type.
   // The value of x has no influence, only the type matters.
   // T has to be a clone base of X.
   { X* px=static_cast<X*>(Base::p->clone()); return X(*px);}

template <class X>
bool get(X& x)const
   // retreiving the element from the container as a reference
   // to the prescribed type. The value of x at input has no
   // influence, only the type matters. T has to be a clone base of X.
{ X* px=static_cast<X*>(Base::p->clone()); x=X(*px); return true;}

void set(T const& x){ cow(); Base::p=static_cast<T*>(x.clone());}
   // loading an object of type T or of a derived type into Pp<T>
   // no longer needed since assignment works now
```

```
   Pp<T>& operator=(const T& x){ cow(); return *this=Pp<T>(x);}
      // 2004-09-27 together with a redefinition of
      // Vp<>::operator[]() this allows the intialization
      // of components of polymorphic arrays by assignment
      // so that the less idiomatic set(i,T const&)-functions
      // should no longer be needed. The problems with the old
      // attempts to set values of components via assignment are
      // now understood (and eliminated)
   Word nameOf()const
   {
      return Word("Pp< "&CpmRoot::Name<T>()(T())&" >");
   }

protected:
   void cow(void);
      // 'copy on write'. The first '*this-changing' statement in the
      // body of a non-constant member function has to be cow();
};

template <class T>
void  Pp<T>::cow(void)
{
   if (Base::u.makeonly()){
      if (Base::p!=0) Base::p=static_cast<T*>(Base::p->clone());
         // let Tb be the class closest to the root of the class
         // hierarchy where a virtual ... clone(void)const is defined,
         // then present clone is Tb* T::clone()const. Notice that
         // modern compilers sometimes accept T* T::clone()const.
         // Microsoft Visual C++ 5.0 is not of this kind,
         // so in defining clone() in a class, we have always to
         // remember the base. This is only a minor inconvenience.
      Base::u.startNew();
   }
}

//////////////////////////// class Po< > ////////////////////////////////

template <class T>
class  Po: public  Pp<T>{  // Adding order operations to Pp<T>
     // Allows to form, for instance, std::vector<Po<T> >
     // and thus allows to combine STL functionality with polymorphism
     // and banning of pointers
   typedef Pp<T> Base;
public:
   Po(T* p_=0): Pp<T>(p_){}
   Po(const Pp<T>& x): Pp<T>(x){}
   bool operator  <(const  Po& x)const{ return *Base::p <  *(x.p);}
   bool operator ==(const  Po& x)const{ return *Base::p == *(x.p);}
   bool operator  >(const  Po& x)const{ return *Base::p >  *(x.p);}
   bool operator !=(const  Po& x)const{ return *Base::p != *(x.p);}
```

```
   bool operator <=(const  Po& x)const{ return *Base::p <= *(x.p);}
   bool operator >=(const  Po& x)const{ return *Base::p >= *(x.p);}
};

///////////////////////////  class Vp< >  ////////////////////////////////
// This template behaves just as Pp<T> except that access is indexed
// which allows to store a number of instances and to retreive them:
// Let T,T1,T2,T3 as in the explanation of Pp<>. Then a typical usage is
//
// T1 t1(...);
// T2 t2(...);
//
// V<T3> t3(2);
// t3[1]=T3(...);
// t3[2]=T3(...);
//
// Vp<T1> vp(4);
// vp[1]=t1;
// vp[2]=t2;
// vp[3]=t3[1];
// vp[4]=t3[2];

// A detailed description (see Pp<>) of the the vp-components can be
// created like
// that:
//
// Z nItems=4;
// V<Media> m(nItems);
// (for Z i=1;i<=nItems;i++) vp(i).show(m[i]);
//
// Thus, despite their differing types, the t1, t2, t3[1], t3[2] get
// processed in a single loop; this capability is indispensible for
// polymorphism to be useful in non-trivial situations. The most
// expressive application is to use Vp<T1>-typed data members in classes:
//
// class ParticleMix{ // assume that Particle is clone base for both
//    // ParticleA and ParticleB
//    Vp<Particle> p;
// public:
//    ParticleMix(Z nA, Z nB):p(nA+nB)
//       // creates a mix of nA particles of type A and
//       // and nB particles of type B
//    {
//       Z i;
//       V<ParticleA> pA(nA); for (i=1;i<=nA;i++) p[i]=pA[i];
//       V<ParticleB> pB(nB); for (i=1;i<=nB;i++) p[i+nA]=pB[i];
//    }
//    void moveIndependently(R timeStep)
//    {
//       for (Z i=1;i<=p.dim();i++) p(i).moveIndependently(timeStep);
```

```
//              // applies the law of motion for A-particles if p(i) is
//              // a A-particle, and for B-particles correspondingly
//    }
// };
//
// Whereas in a 'stand alone'-control structure like the 'Media-loop'
// given above it would not be much more complicated to use a T1 pointer
// for implementing the loop, in the class example we gain more: Since
// Vp<Particle> is a value class, the compiler will properly and
// automatically define copy constructor, assignment operator, and
// destructor for us, wheras using pointers would leave this burden on
// us. Programmers who did this pointer implementation several times
// successfully might begin to love the procedure. I consider it wasted
// time and energy, and a lost opportunity to deal with complexity in a
// state-of-the-art manner.

template <class T>
class  Vp{ // polymorphic V template

   CpmArrays::V< Pp<T> > rep;

public:

    Vp(){}
      // notice that  both Pp<T> and V define a default constructor

   explicit Vp(Z n):rep(CpmArrays::V<Pp<T> >(n,Pp<T>())){}
      // No pre-setting of values occurs. If needed, this has to be
      // done by function void set(T const& x).

   Pp<T>& operator[](Z i){ return rep[i];}
      // We use [] for access to the true components, which
      // are instances of the true value class Pp<T>.
      // Typical usage:
      // Vp<T> vp(2); T x=..., y=...; vp[1]=x; vp[2]=y;
   Pp<T> const& operator[](Z i)const{ return rep[i];}

   T& operator()(Z i){ return rep[i]();}
   T const& operator()(Z i)const{ return rep[i]();}
      // We use () for access to the T& that will be in
      // most cases the objects needed for further
      // processes, as for usage as function arguments.
      // Typical usage:
      // Vp<T> vp=...; Word w1=vp(1).nameOf(), w2=vp(2).nameOf();

   template <class X>
   X operator()(Z i, const X& x)const
      // retreiving the i-th component from the container as a copy of a
      // prescribed type. The value of x has no influence, only the type
      // matters. T has to be a clone base of X.
```

```
      { X* px=static_cast<X*>(rep[i]().clone()); return X(*px);}

   template <class X>
   bool get(Z i, X& x)const
      // retreiving the i-th component from the container as a referemnce
      // to the prescribed type. The value of x at input has no
      // influence, only the type matters. T has to be a clone base of X.
   {
      if (i<1 || i>rep.dim()) return false;
      X* px=static_cast<X*>(rep[i]().clone()); x=X(*px);
      return true;
   }

   void set(Z i, const T& x){ rep[i]=x;}
      // no longer essential since the defining statement
      // is natural enough not to need a wrapper

   void set(T const& x){ for (Z i=1;i<=rep.dim();i++) rep[i]=x;}

   Z dim()const{ return rep.dim();}
   Z b()const{ return rep.b();}
   Z e()const{ return rep.e();}

   Word nameOf()const{
      Word nt=CpmRoot::Name<T>()(T());
      Word wi="Vp<";
      return wi&nt&">";
   }
};

} // namespace

#endif
```

133 cpmpardynsys.h

```
//? cpmpardynsys.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_PARDYNSYS_H_
#define CPM_PARDYNSYS_H_
/*
Description: Heterogeneous dynamical systems prepared for
   parallel computing.
   Useful as a base class for a granular system, where
   a large number of particles, and a variety of confining systems
   such as walls, movable pistons, semi-permeable membranes ...
   form a dynamical system.

   Development version with speed increasing features.
   Some of the comments are not updated and thus refer to the old
   class HetDynSys.
*/
#include <set>
#include <sstream>

#include <cpmrmatrix.h>
#include <cpmgraph.h>
#include <cpmm.h>
#include <cpmp.h>
#include <cpmvectormore.h>
#include <cpmalgorithms.h>
#include <cpmimagingtools.h>
#include <cpmrecordhandler.h>
#include <cpmproperties.h>
#include <cpmdynaux.h>
#include <cpmdiscretespace.h>
#include <cpmparticleviewer.h>
#include <cpmmacros.h>
#include <cpmpardef.h>

namespace CpmDynSys{

   // no quantities from namespaces CpmDim2 or CpmDim3
   // should be used here. The dimension comes in when template
   // arguments get selected e.g. Spc as CpmDim2::Spc or
   // CpmDim3::Spc. It was not always easy to keep the template arguments
   // general enough and not assuming the specific PaDyAn choice for
   // Spc, FieldSpace, VectorField, ....

   using namespace CpmStd;
   using std::set;
   using std::vector;
```

```
using CpmRoot::read;
using CpmRoot::write;
using CpmRoot::R;
using CpmRoot::Z;
using CpmRoot::Word;

using CpmSystem::Error;
using CpmSystem::OFileStream;
using CpmSystem::Timer;

using CpmArrays::R_Vector;
using CpmArrays::R_Matrix;
using CpmArrays::V;
using CpmArrays::VV;
using CpmArrays::Vp;
using CpmArrays::Pp;
using CpmArrays::Vo;
using CpmArrays::S;
using CpmArrays::M;
using CpmArrays::X2;
using CpmArrays::X3;

using CpmFunctions::F;
using CpmFunctions::F2;
using CpmFunctions::F3;
using CpmFunctions::F4;
using CpmFunctions::R2_Func;

using CpmRootX::Z1;
using CpmRootX::R1;
using CpmRootX::B;
using CpmRootX::hugeNumber;
using CpmRootX::Record;

using CpmLinAlg::R2;

using CpmGeo::Iv;
using CpmGeo::Angle;
using CpmGeo::DEG;

using CpmAlgorithms::CyclicAlarm;
using CpmAlgorithms::BurstAlarm;
using CpmAlgorithms::LatticeBox;

using CpmGraphics::Graph;
using CpmGraphics::Frame;
using CpmGraphics::Frames;

using CpmImaging::Color;
```

```
   using CpmImaging::Palette;
   using CpmImaging::INVISIBLE;
   using CpmImaging::WHITE;
   using CpmImaging::LIGHTBLUE;

   using CpmProperties::SysControl;
   using CpmProperties::SysControlState;

   using CpmPhysics::ParticleViewer;

   using CpmMPI::Cpm_com;

namespace aux{
   void adjust_dt(R&,Z);
}// aux

////////////////////// class ParDynSys ////////////////////////////////
// 'Parallel Dynamical System'

#define CPM_CN ParDynSys
   // this style of introducing the class name helps to define
   // new versions

#define CPM_TAL template <class Par, class Ext,\
   class Vec, class Spc, class Group, class Partition,\
   class For, class FieldSpace, class ScalarField, class VectorField,\
   class PointCharge, class SourceOfVectorField, class MagDipole,\
   class SphPart, class Camera>

#define CPM_FCN CPM_CN<Par,Ext,Vec,Spc,Group,Partition,\
   For,FieldSpace,ScalarField,VectorField,PointCharge,\
   SourceOfVectorField,MagDipole,SphPart,Camera>

//////////////////////////// class ParDynSys ////////////////////////////
CPM_TAL
class CPM_CN{// = ParDynSys dynamical system of particles, granular matter

   static bool fEqui(const Par& p1, const Par& p2)
   {
      return p1.get_id()==p2.get_id();
   }

protected:
// here we havn't grouped data into independent and dependent ones.
// The independent data are those which are handled in function
// prnOn(...). Function scanFrom(...), together with function
// prepareStep(...) defines the mechanism by which
// the dependent quantities get their values.

   StepControl stpCon;
```

```
      // book-keeping quantities for time stepping, these data also
      // include the time t, which, despite of its physical importance
      // is placed into one house together with some minor relatives

   V<Par> par;
      // here, polymorphism isn't essential, toner and carrier particles
      // can be reasonably described as instances of a single class

   Vp<Ext> ext;
      // here, the components may be a toning shell, a magnetic roller, a
      // photo-conductor .... In the old version these entites were all
      // combined in a single entity since polymorphic arrays were not at
      // hand. All these types of external systems have to be derived
      // from class Ext.

   Z1 np;
      // number of particles, not really independent, related to par
      // makes I/O easier if considered independent. The way of
      // initializing np and ne in the constructor asks np to be placed
      // here later than par and ne later than ext.

   Z1 ne;
      // number of external systems; see np

   V<Spc> xPar;
      // list of particle positions, needed to monitor motion
      // which helps to decide when certain arrays have to be updated

   vector< set<Z> > nearList;
      // nearList is an array of index sets.
      // since 2006-11-09 I use std::vector again here since V would
      // require that std::set would define nameOf.
      // nearList[i-1] holds all indexes of particles which
      // have distance from particle i smaller than nearLength.

   R nearLength;
      // see nearList

// data and functions supporting parallelism. Notice that the partition
// of the whole system into parts and describing the interaction between
// different parts by electromagnetic fields which can be represented
// by relatively few data is a contribution to speeding up the dynamical
// algorithm even if the various parts can't be computed in parallel.

   mutable Z rank;
   mutable Z size;
      // declared mutable for technical reasons: the function that
      // is suitable for initializing these happens to be favorably
      // declared const
```

```
M<Z,Color> showColor;
   // Map (associative list) of particle colors to be used in marking
   // particles if different from default. The key is to be
   // understood as the particle ID number, not the index in the
   // array par.

Partition partition;
   // partitions the available space into regions for which a process
   // takes responsibility. The whole space available to the particles
   // of the system is divided into disjoint subsets. And each such
   // subset (region) is associated with just one process. To each
   // region there is an extended region which contains its region so
   // that the extended regions cover the space available to the
   // particles in a manner that allows overlap. Let the regions be
   // denoted X(i), the extended regions Y(i) (see cpmpartitionx.h),
   // where i indexes the processes.

B meanFieldAct;
   // initialized as false, which is OK

V<Spc> pointList;
   // a list of points determined by partition and needed
   // for defining meanFieldPar

//V<Z> nextPointIndexes(const Spc& x, R range);
set<Z> nextPointIndexes(const Spc& x, R range);
   // returns the list of all i for which
   // |pointList[i]-x|<=range. If such points don't
   // exist, we return the void list.

VV<Spc> repPointList;
    // an augmentation of pointList taking into account toroidal
    // topology

VectorField meanFieldPar;
   // smooth average field created by particles not listed in
   // nearList according to my 'self-consistent far field method'.
   // Besides this, it contains contributions
   // from virtual periodic replications (if partition implies
   // toroidal topology) and contribution from course grained
   // electro-magnetic sources from the neighbor processes.
   // See function prepareStep()

ScalarField meanPotPar;
   // electrical potential corresponding to meanFieldPar

void redistribute(void);
   // Let be any state of par over the various processes.
   // After call of this function, par for each process is in
   // a form that each process has just the particles that he owns
```

```
      // and those that it has to use.

      // Till 01-11-08 I had a hard time to understand the deadlock
      // happening in earlier versions. The final solution follows the
      // suggestion 'different ordering of calls between tasks' in
      // Section 2 of
      // http://www.tc.cornell.edu/Services/Edu/Topics/MPI/Pt2pt2/

   void setTimesFromExtSys();
      // make all times of system identical and equal to the mean
      // value of the times defined by the external systems
      // (which all have a get_t() function). This is important for
      // full reconstruction of a state from the information
      // written on movie-files
public:
   void prepareStep(const Camera& cam=Camera());
      // sets the preconditions for function step() to work
      // properly. After this function did its work, the function step
      // runs 'autonomously' without communicating with other processes
      // till it calls redistribute(). Thus the responsibility for calling
      // step(dt) with a reasonable value of dt is with the user of step.
      // The idea is that all influences which a particle feels from
      // particles outside the near zone of radius nearLength should be
      // effectively constant during dt.
protected:
   virtual void steplet(R t0, R dt, const Camera& cam);
      // time evolution over a fixed span dt of time. Implements my
      // direct mid-point method of time-discrete dynamics.
      // After function call, the whole system is in a state that is the
      // best guess for a system state at time t0+dt (based on the
      // information contained in *this at the point of function call).
      // In particular, for all particles par[i] we have after function
      // call par[i].get_t()==t0+dt. In parallel computing the various
      // processes don't interact during executing this function.

// constructors. The data elements tPar, xPar, nearList, filValList
// will not become initialized by the constructors, but only in preparing
// the execution of a time step.
public:
   CPM_CN(V<Par>  par_, Vp<Ext> ext_,
      Partition partition_=Partition()):
      par(par_),ext(ext_),np(par.dim()),ne(ext.dim()),
      partition(partition_), showColor(){}

//public:

   CPM_CN(V<Par>  par_, Z ne_):
      par(par_),ext(Vp<Ext>(ne_)),
      np(par.dim()),ne(ext.dim()),showColor()
      { }
```

```
      // simple constructor for filling in the external systems by
      // calling
      // void setExtComp(Z i, const X& x);
      // this is sufficient for enabling the visualization functionality
      // of the class.
      // For parallel computing also a proper setting of partition is
      // required.

   CPM_CN():par(V<Par>(0)),np(0),ne(0),showColor(){}
         // notice that the default for V<Par> would bring in one
         // particle

// I/O and documentation

   typedef CPM_CN Type;
   typedef CPM_CN CloneBase;
   CPM_IO_V // stream I/O, important that this is virtual
      // these are the functions scanFrom and prnOn which deserve some
      // clarifications.
      // Precondition of prnOn(str) is only that
      // str in valid. Then a complete data set for the system state gets
      // written on the stream, restricted only by the number of digits
      // chosen for data to ASCII conversion (set in cpmdefinitions.h
      // #define CPM_WRITE_PRECISION). By definition, the data set is
      // complete if it - together with the data of SysControl
      // determines the behavior of function step().
      // All 'bookkeeping quantities' that get initialized
      // anyway within function step() are not written
      // to str. These, however are copied by the copy constructor and
      // by the assignment operator. Unfortunately there is even a third
      // copying method in the game: the way in which states are filed
      // (movie-file production) Since these files tend to grow very
      // large, it is essential to have this process very data-efficient.
      // See comment to function file(...).
      // Precondition of scanFrom(str) is that str is valid and that the
      // data on str are such as if they had been written by
      // pardyn.prnOn(str) where pardyn is an instance of the present
      // template class with the same template arguments as *this
      // and with the  s a m e  n u m b e r  and  e x a c t  t y p e s
      // (derived from ExtSys) as this.
      // Postcondition is that *this is changed exactly into pardyn.
      // (By definition, a postcondition is met after function call if
      // the precondition held before function call; this seems to be a
      // useful notion, see e.g. Paul Taylor: Practical Foundations of
      // Mathematics, Cambridge 1999, p. 199)

   virtual Word nameOf()const{ return Word("ParDynSys<...>");}

   virtual CPM_FCN* clone()const{ return new CPM_CN(*this);}
```

```
   void done(R dt_, R dis_){ stpCon.done(dt_,dis_);}
      // if a time step dt was effective in changing the dynamical
      // variables of the system, this is the change to the 'book-keeping
      // quantities'

// obvious 'read'-access functions
   R get_t()const{ return stpCon.t;}
   R get_dt()const{ return stpCon.dtMean;}
   R get_tLimit()const{ return stpCon.tLimit;}
   bool getRelaxFlag()const{ return stpCon.relaxFlag;}

   const Par& getPar(Z i)const{ return par[i];}
   V<Par> getPar()const{ return par;}
   Vp<Ext> const& getExt()const{ return ext;}
   Vp<Ext>& getExt(){ return ext;}

   template <class X>
   X getExtComp(Z i, const X& x)const{ return ext(i,x);}
      // template member function works!!! fine
      // Here the value of x is unimportant, it is needed only to tell
      // the function the precise type of the returned quantity

   Z get_np()const{ return np;}
   Z get_ne()const{ return ne;}
   const Partition& getPartition()const{ return partition;}
   void setPartition(const Partition& pa){ partition=pa;}

   virtual Z get_nTot()const{ return np;}
      // in a derived class that support parallel computing
      // this has to become re-defined to mean the total number
      // of particles over all processes

   Spc cenOfMas(const V<bool>& vb)const;
      // returns the center-of-mass of those particles par[i]
      // for which vb[i]==true (actually we need the center-of-mass of
      // all carrier particles, but being a carrier particle is too
      // special a concept to be placed here. The default argument is set
      // such that the center-of-mass of all particles is calculated if
      // no vb-argument is given

   R get_m(const V<bool>& vb)const;
      // returns the mass of those particles par[i]
      // for which vb[i]==true

   R kinEnr(const V<bool>& vb)const;
      // returns the kinetic energy of those particles par[i]
      // for which vb[i]==true

   R potEnr(const V<bool>& vb)const;
      // returns the potential pair energy of those particles par[i]
```

```
      // for which vb[i]==true
      // Included are only Coulomb forces of point charges and
      // gravitational forces of mass points (added 2004-05-03)
      // Useful for testing energy conversation in pure Coulomb
      // systems or gravitational mass point systems.

   R kinEnr()const;
      // returns the kinetic energy of all particles owned by rank

   R potEnr()const;
      // returns the potential pair energy of all particles owned by rank
      //
      // by calling  res+=par[i].pot(par[j]); within a suitable loop.
      // So, the pot memberfunction of class Par decides what
      // contributions are employed here.
      // Notice that this has np^2 complexity and thus should be called
      // with caution.
      // Useful for testing energy conversation in pure Coulomb
      // systems or gravitational mass point systems.

   R potElasticExt()const;
      // potential energy of all particles resulting from elastic
      // interaction with the external systems

   R potHomFil(FieldSpace const& fs)const
   {
      R res=0;
      for (Z i=1;i<=np;++i){
         if (par[i].getOwner()==rank) res+=par[i].pot(fs);
      }
      return res;
   }
      // Total potential energy of the particles in the homogeneous field
      // of (constant) strength fs

   R totEnr()const;
      // an expression for parts of the energy which is the total energy
      // in simple conservative test systems

   R get_m()const;
      // returns the mass of all particles

// 'write'-access to the most important substructure
   template <class X>
   void setExtComp(Z i, const X& x){ext.set(i,x);}
      // template member function works!!! fine. Class X has to be a
      // derivation of Ext.

   void setPar(const V<Par>& par_){ par=par_; np=par.dim();}
      // a combination of get and set allows to perform modifications of
```

```
      // *this for which there are no member functions provided in
      // ParDynSys<>. For instance, a list of np instances of Group can
      // be used to transform the particles, each particle by its own
      // transformation:
      // V<Par> vp=getPar(); // extract the particles for modification
      // V<Group> vg(np);
      // Z i;
      // for (i=1;i<=np;i++) vg[i]=...; // define the Group-instances
      // for (i=1;i<=np;i++) vp[i]*=vg[i]; // transform the extracted
      // particles
      // setPar(vp); // Put the particles back. Now *this houses as par
      // the transformed list of particles

   void selectParticles(const V<bool>& vb)
   { par=par.select(vb); np=par.dim();}
      // allows to remove particles from the system under control of a
      // boolean array. Paricles par[i] for vb[i]==true are selected to
      // stay, not to get removed!
      // has to be redefined in a derived class

   void setActivityOfExternalSystems(Z iext, Z act)
      { ext(iext).setActivity(act); }
      // allows to disable the dynamical anf graphical activity of
      // selected external systems.

   void setVelocityOfExternalSystems(Z iext, const Vec& v)
      { ext(iext).set_v(v); }

   void setCol(V<Palette> const& c)
      {for (Z i=1;i<=np;i++) par[i].setCol(c[i]); }
         // setting the visualization color

   void setShowColor(Z i, Color c){ showColor[i]=c;}
      // setting the special Color (not only Palette). For the particle
      // with ID-number equal to i, color c is available for
      // visualization.

// 'write' access to simple control quantities
   void set_t(R t0);
      // make all times of system identical to the input
      // value t0;
   void set_tLimit(R dtL){ stpCon.tLimit=dtL;}
   void set_tSmall(R ts){ SysControl::tSmall=ts;}
      // see attribute tSmall
 //  void clear_dtLimit(void){stpCon.dtLimit=hugeNumber;}
   void clear()const{ VectorField::newScl(); }

   void setRelaxFlag(bool rf)
      // switching between 'relaxed dynamics' (a 'dynamics' which sets
      // all velocities to zero after each time step) and normal dynamics
```

```
      {
         stpCon.relaxFlag=rf;
         if (rf) for (Z i=1;i<=np;i++) par[i].stop();
      }

   void showParticleFields(bool b){SysControl::showPartFields=b;}

// kinematical transformations
   void boost(Vec const& v){ for (Z i=1;i<=np;i++) par[i].boost(v);}

   void ranBoost(Vec const& v, R dv, R angleRange, Z j=0)
   { for (Z i=1;i<=np;i++) par[i].ranBoost(v,dv,angleRange,++j);}

   void ranRotPar(R angleRange=180, Z j=0);
      // randomly rotate particles

   virtual void operator*=(const Group&);
      // geometrical transform of the whole system
      // if one adds data in derived classes, these have to be
      // transformed too; thus the function should be virtual

   void transform(Z i, const Group& g){ par[i]*=g;}
      // geometrical transform of particle i

   void transformExternalSystems(Z i, const Group& g){ ext(i)*=g;}
      // geometrical transform of external system i

   For force(const Par& par1, const Par& par2)const;
      // force which particle par1 feels from particle par2 and all the
      // mirror charges which get generated by par2 (not by par1 !)
      // Mirror charges come from the external systems individually
      // (e.g. from the Shell and the Target) and are defined in the
      // function influence() of the respective classes.
      // For force(par1,par1) returns the force felt by its own
      // mirror charge without an non-sensical direct self-force
      // since Clust::forces(const Clust&) takes care of this.

   FieldSpace field(const Spc& sp, const Par& par1,
      const bool& farAway=false)const;
      // field strength that particle par1 creates in point sp together
      // with all the mirror charges created in external systems. These
      // are defined by the same functions influence() called in the
      // previous function force().
      // If the last argument is true, the various mirror charges
      // created from the same external devise get collected into
      // one charge for computational efficiency.

   R potential(Spc const & sp, Par const& par1,
      bool const& farAway=false)const;
      // potential belonging to interaction type it that particle par1
```

```
      // creates in point sp ( together with all the mirror charges
      // created in external systems, if it==1)
      // it: interaction type
      // it=1: electric, it=2: magnetic, it=3 gravitational
      // Notice that the more important vector fields unify the
      // interaction types by using the return type FieldSpace
      // which unites electric, magnetic, and gravitational field.
      // The analog method here would be to use a return type R3.
      // However, too much of the existing code is based on
      // type ScalarField which would not harmonize with such a
      // choice.

// lambda abstractions of the previous

   VectorField fieldPar(const Par& par1, bool farAway=false)const;
    //  { return F2<Spc,Par,bool,FieldSpace>(par1,farAway)(field);}
    // This does not work since field is a member function and not a
    // plain function of signature Spc x Par x bool -->FieldSpace
    // However, the working version given below (not inline) is
    // also not much more complicated. It is a triumph of my lambda
    // abstraction mechanism

   ScalarField potentialPar(Par const& par1, bool farAway=false)const;

   void sizeAndRank()const{
      size=Cpm_com.getSize();
      rank=Cpm_com.getRank();
   }

   void step(R dt,const Camera& cam);
      // evolving the system over exactly the time step dt.
      // After call all subsystems refer exactly to the same time. The
      // working of the function will not be completed before it is
      // completed in each of all parallel processes. This function is
      // intended to be called for dt' s large enough that the change
      // of system state conveys information to the user. Since,
      // however, the particles which show a distance larger than
      // nearLength interact in a course-grained and constant manner
      // during dt (especially no inter-process communication takes place
      // during this span of time) dt has also not to be taken too large.
      // My present usage is to have dt=(time span between adjacent movie
      // frames)/5=1.88e-6 s. This time step is divided into
      // SysControl::fixedStep=20 steplets which implement the direct
      // mid-point method of integration.

   bool run(R dtStep, Z nSteps,
         Z filingPeriod, Z docuPeriod,
         Z viewingPeriod, Z stoppingPeriod, const Word& movieName,
         Camera& cam, const Timer& tm, const F<R,Z>& f,
         OFileStream& ofs);
```

```
    // Calling function step(dtStep) nSteps times in sequence
    // and writing to file (under control by movieName) and visualize
    // on screen (under control by cam) with a period (in number of
    // calls of step(R)) of filingPeriod and viewingPeriod
    // respectively.
    // Exceptions: nSteps<0 or dtStep<0 renders the action of the
    // function trivial.
    // filingPeriod<1 disables all filing
    // viewingPeriod<1 disables all display activity
    // Thus for nSteps==0 steps(R) will not be called but *this
    // will be filed unless filingPeriod<1 and *this will be displayed
    // on screen unless viewingPeriod<1.
    // The last argument manipulates as a function of t the parameters
    // in SysControl.
    // The argument tm allows to set a limit to the computation time.
    // After a filing action tm gets asked whether the timeOut
    // condition holds and if this is the case the program is stopped
    // after the last state has been written to file completely. After
    // restarting the run from file one needs not to adjust nSteps
    // according to the computational progress already achieved.
    // The new functionality governed by tm is not implemented in the
    // present class but only in the derived class Station by suitable
    // re-definition of virtual function file.
    // return value is true iff function was stopped early (earlyStop)

R_Vector anlOrd(R dtStep, Z nSteps, Z nFrames, Z fac, Word const& tag,
    Camera& cam, Iv const& iv, Z repLog, R repOrderD, R repOrderE);
    // documents the behaviour of a consecution of nFrames (>=2) runs
    // where the number of steplet calls within a step is
    // given as SysControl::fixedStep for the first run and increases
    // by the factor fac in each following run.
    // Both the trajectory and the total energy totEnr() along
    // trajectories is displayed on screen and the corresponding data
    // written to various files, the name of which always contains tag.
    // For the determination of the order in dt only the fraction
    // of the trajectory given by iv is taken (Iv iv(0,1) corresponds
    // to the whole trajectory. If iv is void extracting the order from
    // data is disabled. The same is true for the orbit order if
    // repOrderD>0 and for the energy order if repOrderE>0.
    // If iv is void and repOrderD<=0, then order is taken as 0, so
    // that no scaling of data according to an order takes place. The
    // same is true for repOrderE<=0.
    // If repLog!=0, the screen graphics is shown with y replaced by
    // log10(cpmabs(y)+1e-16). Data files tag&X&".txt" are created for
    // X = D,DL,E,EL in a format that facilitates the creation of
    // order-collapsed families of curves for path variation and energy
    // variation in either normal (X=D,E) or logarithmic (X=DL,EL)
    // form.
    // Two movies fsMovie&tag&".txt" and midMovie&tag&".txt" are made:
    // The first gives the final states of the runs in direction of
```

```
      // largest steplet-step to smallest steplet-step
      // and the last one gives the whole trajectory for a run in the
      // middle (the larger if nSteps+1 is even !)

   Vp<CPM_FCN> trj(R dtStep, Z nSteps, Camera& cam, Z viewingPeriod=0);
      // trajectory
      // returns the nSteps+1 states: *this (initial state) and nSteps
      // states resulting from applying step(dtStep,cam) nSteps times.
      // Visualization with cam can be enabled by giving the last
      // argument a value >0. The documentation channel OFileStream& ofs
      // would be redundant in this case since any application dependent
      // analysis of the state can be done with the result in a
      // transparent manner.
      // Notice that this does not work usefully with parallel execution
      // since one probably would need the array of total system states
      // in this case.
      // Due to the usage of the polymorhic array one gets the total
      // information of the states even when calling this function for a
      // suitably derived class. Example of usage:
/*
         Vp<Sys2Base> vsb=sys.trj(dtStep,nSteps,cam);
         Z dv=vsb.dim();
         for (Z i=1;i<=dv;++i){
            cam.clear();
            vsb(i).mark(cam,true);
            cpmwait(SysControl::tWait);
         }
*/
      // For long runs of large systems this needs a lot of memory,
      // wheras function run() is much less demanding with respect to this
      // point, it has a much more complicated interface, though.

   void movieFromTrj(Vp<CPM_FCN> const& trj, Word const& fileName,
      Camera& cam, Z viewingPeriod=1);
      //

   void stopParticles()
      // sets the velocities of the particles to zero
   {
      for (Z j=1;j<=np; j++){
         par[j].stop();
#if !defined(CPM_POINT_PARTICLES)
         par[j].deElectrize();
#endif
      }
   }

   virtual void setVel(){}
      // allows for derived classes to implement methods that give the
      // particles suitable velocities e.g. as in a gas of some given
```

```
      // temperature. Since all velocities are initialized as zero, it is
      // appropriate to define this action as addition of velocities to
      // the existing one.

// descriptors

   VectorField getFieldExt()const;
      // returns the field got by adding all contributions from Env and
      // all external systems

   VectorField getFieldPar(bool mean=false)const;
      // returns the field got by adding the contributions of all
      // particles and the mirror charges associated with them
      // (mirror charges where not included here till 03-01-13)
      // Forming the sum is done on the level of fields so that
      // much stress is put on the stack.
      // If the argument mean is true, the field meanFieldPar
      // is returned. This may differ from the field for
      // mean==false due to the periodic replicas.

   VectorField getFieldTot(bool mean=false)const;
      // return the sum of the previously considered fields

   void markPot(const Camera& cam)const;
      // shows on sceen (or only in cam's image memory; and after
      // calling cam.display() on screen) the potential
      // distribution in cam's image plane. What subsystems are included
      // in the potential is controled by
      // SysControl::showPartFields
      // SysControl::showMeanFil
      // SysControl::ignoreExternalSystems
      // SysControl::noInteractionChargeToWall

   void markField(const Camera& cam)const;
      // same logic as markPot(Camera)

   void markPotTot(const Camera& cam)const;

   void markPotExt(const Camera& cam)const;
      // same data types used as in previous function. For uniformity
      // with previous function

   void markPotPar(const Camera& cam)const;
      // before marking on screen, the particle fields will
      // be added as instances of DisField (a discrete version
      // of ScalarField) so that the number of fields that can be
      // is not limited by the stack.
      // The discrete set of points for the definition of DisField
      // is provided by the argument cam
```

```
void markCharge(const Camera& cam)const;
   // shows the charge density derived from charges sorted
   // into a rectangular grid on the field of view of the camera

void markScalarValue(const Word& w, const Camera& sc)const;
   // similar to charge density, for other properties indivated by w

ScalarField getPotExt()const;
   // sum of ext(j).getU()

ScalarField getPotPar(bool mean=false)const;
   // sum of the fields potentialPar(par[i]) Notice that this includes
   // the  contributions of mirror charges.
   // For mean=true, this is the
   // potential generated by particles in the form as
   // the actual dynamics (in function steplet) sees it
   // (actually, the dynamics only sees vector field
   // meanFieldPar. However, meanPotPar is built closely
   // parallel to meanFieldPar that it is obvious that it
   // actually is the potential of meanFieldPar.
   // Notice that in the present version of the program
   // gravitation is included and that the ScalarField returned
   // from this function is the gravitation potential in case
   // SysControl::noInteractionMassToMass==0.

ScalarField getPotTot(bool mean=false)const;
   // sum of getPotExt() and getPotPar()

R disVal(const Type& )const;
   // comparison function on which the step adjustment in step() is
   // based. This is not limited by 1 as in my numerical classes. See
   // the defining code for the meaning.

V<R> dist(const Type& )const;
   // returns distance values as individual values for the the
   // subsystems and not only the maximum over all

virtual void mark(const Camera& cam, bool display,
   const Word& infoText=Word())const;
   // visual representation of the configuration on which movie
   // production is based. What is shown (e.g. vector fields or
   // potentials) is controlled by
   // SysControl::bitmapToShow
   // SysControl::showParticles
   // SysControl::showMeanFil
   // SysControl::ignoreExternalSystems
   // SysControl::showPartition.
   // for details see implementation code

virtual bool file(const Word& fn, const Timer& tm)const
```

```
      {
         cpmwarning("Trivial version CPM_FCN::file(...) called");
         fn; tm;
         return false;
      }
      // Intended to writes the state into a suitable file, which should
      // be made visible by a suitable viewer program on a
      // single computer. Since the number of particles to be
      // considered is huge, it is rather evident that one should
      // file not too much redundant data. This suggests to divide
      // the data of an object in constant ones and those that change
      // with time. This is presently implemented in a non-trivial manner
      // only in a derived class Station.

   virtual X2<Z,Word> docuStep(OFileStream&)const
      { return X2<Z,Word>(0,"not useful");}
      // a function to be called in function run(....) just after step()
      // in order to allow problem dependent information gathering
      // In a useful implementation, the return value gives the number of
      // R's that are written to file in sequence in one call of the
      // function.
      // If active==false, the function will return this number without
      // acting on the argument stream.
};

CPM_TAL
void CPM_FCN::setTimesFromExtSys()
{
   if (ne==0) return;
   R sum=0;
   Z i;
   for (i=1;i<=ne; i++) sum+=ext(i).get_t();
   R tMean=sum/ne;
   for (i=1;i<=ne; i++) ext(i).set_t(tMean);
   stpCon.t=tMean;
}

CPM_TAL
void CPM_FCN::set_t(R t0)
{
   stpCon.t=t0;
   Z i;
   for (i=1;i<=ne; i++) ext(i).set_t(t0);
   for (i=1;i<=np; i++) par[i].set_t(t0);
      // par[i]-loop was missing till 02-06-28 and spoiled the
      // start of ctc7c run3
}

CPM_TAL
bool CPM_FCN::prnOn(ostream& str)const
```

```
{
   Z mL=1;
   static Word loc("CPM_FCN::prnOn(...)");
   CPM_MA
   cpmwt("ParDynSys<>");
   cpmp(stpCon);
   cpmp(np);
   cpmp(ne);
   Z i;
   R npInv=cpminv((R)np);
   Z count=0;
   Z countMax=50;
   for (i=1;i<=np;i++){
      count++;
      if (count==countMax){
         count=0;
         cpmmessage(mL,"fraction done: "&cpm(i*npInv));
      }
      cpmp(par[i]);
   }
   V<Word> tL(ne);
   for (i=1;i<=ne;i++){
      Word ti=ext(i).nameOf();
      tL[i]=ti;
      cpmmessage("type of ext("&cpm(i)&") is "&ti);
   }
   for (i=1;i<=ne;i++) cpmp(ext(i));
   cpmp(partition);
   cpmwt("ParDynSys<> end");
   CPM_MZ
   return true;
}

CPM_TAL
bool CPM_FCN::scanFrom(istream& str)
{
   Z mL=1;
   static Word loc("CPM_FCN::scanFrom(...)");
   CPM_MA
   cpms(stpCon);
   cpms(np);
   cpms(ne);
   Z i;
   par=V<Par>(np);
   // needed since the calling instance is not assumed to have the
   // correct number of particles already.
   for (i=1;i<=np;i++) cpms(par[i]);
   cpmmessage(mL,"reading "&cpm(np)&" particles ready");

   // ext=Vp<Ext>(ne); is missing since we assume that the calling
```

```
   // instance already has right number and types of ext(i).
   // Introducing the above statement would even be wrong since it would
   // give the external system the type ExtSys and thus would spoil the
   // additional attributes of derived classes.

   cpmmessage(mL,"trying to read "&cpm(ne)&" external systems");
   for (i=1;i<=ne;i++){
      cpmmessage(mL,"reading external system "&cpm(i)&" ahead");
      cpms(ext(i));
      cpmmessage(mL,"reading external system "&cpm(i)&" ready");
   }
   cpmmessage(mL,"reading external systems ready");
   cpms(partition);
   cpmmessage(mL,"reading partition ready");
   CPM_MZ
   return true;
}

CPM_TAL
void CPM_FCN::operator*=(const Group& g)
{
   Z i;
   for (i=1;i<=np;i++) par[i]*=g;
   for (i=1;i<=ne;i++) ext(i)*=g;
}

CPM_TAL
void CPM_FCN::ranRotPar(R angleRange, Z j)
{
   Group g(Angle(angleRange,DEG));
   for (Z i=1;i<=np;i++){
      Spc xi=par[i].get_x();
      Group gi=g.ranVal(xi,++j);
      par[i]*=gi;
   }
}

CPM_TAL
R CPM_FCN::disVal(const Type& hs)const
{
   Z i;
   R_Vector vp(np);
   for (i=1;i<=np;i++)
      vp[i]=par[i].disVal((hs.par)[i]);
   R res_p=vp.absVal();

   R_Vector ve(ne,0);
   for (i=1;i<=ne;i++)
      if (ext(i).isActive()) ve[i]=ext(i).disVal((hs.ext)(i));
   R res_e=ve.absVal();
```

```
   R2 res2(res_p,res_e);
   R res=res2.absVal();
   if (res==0.){ cpmwarning(
      "ParDynSys<>::disVal=0; parabolic motion or numerical problem");
   }
   return res;
}

CPM_TAL
V<R> CPM_FCN::dist(const Type& hs)const
{
   V<R> res(np,0);
   for (Z i=1;i<=np;i++) res[i]=par[i].disVal((hs.par)[i]);
   return res;
}

CPM_TAL
R CPM_FCN::get_m(const V<bool>& vb)const
{

   R res=0;
   for (Z i=1;i<=np;i++){
      if (vb[i]) res+=par[i].get_m();
   }
   return res;
}

CPM_TAL
R CPM_FCN::get_m()const
{
   R res=0;
   for (Z i=1;i<=np;i++) res+=par[i].get_m();
   return res;
}

CPM_TAL
Spc CPM_FCN::cenOfMas(const V<bool>& vb)const
{
   V<Spc> x(np);
   V<R>  w(np);
   for (Z i=1;i<=np;i++){
      x[i]=par[i].get_x();
      w[i]= vb[i] ? par[i].get_m() : 0. ;
   }
   Spc res;
   res.convexAsMember(x,w,true);
   return res;
}
```

```
CPM_TAL
R CPM_FCN::kinEnr(const V<bool>& vb)const
{
   R res=0;
   for (Z i=1;i<=np;i++){
      if (vb[i]) res+=par[i].kinEnr();
   }
   return res;
}

CPM_TAL
R CPM_FCN::potEnr(const V<bool>& vb)const
{
   R res=0;
   Z i,j;
   for (i=1;i<=np;i++){
      if (!vb[i]) continue;
      for (j=1;j<i;j++){
         if (!vb[i]) continue;
         res+=par[i].pot(par[j]);
      }
   }
   return res;
}

CPM_TAL
R CPM_FCN::kinEnr()const
{
   R res=0;
   for (Z i=1;i<=np;i++){
      if (par[i].getOwner()==rank) res+=par[i].kinEnr();
   }
   return res;
}

CPM_TAL
R CPM_FCN::potEnr()const
{
   R res=0;
   Z i,j;
   for (i=1;i<=np;i++){
      bool bi=(par[i].getOwner()==rank);
      if (bi){
         for (j=1;j<i;j++) res+=par[i].pot(par[j]);
      }
   }
   return res;
}

CPM_TAL
```

```
R CPM_FCN::potElasticExt()const
{
   R res=0;
   Z i,j;
   for (i=1;i<=np;i++){
      bool bi=(par[i].getOwner()==rank);
      if (bi){
         for (j=1;j<=ne;j++) res+=ext(j).potElastic(par[i]);
      }
   }
   return res;
}

CPM_TAL
R CPM_FCN::totEnr()const
{
   R res=kinEnr()+potEnr()+potElasticExt();
   for (Z i=1;i<=np;++i) res+=getExt()(1).potField(getPar(i));
   return res;
}

CPM_TAL
For CPM_FCN::force(const Par& par1, const Par& par2)const
{
   For res=par1.forces(par2);
      // direct force, vanishes for par1==par2
      // Since 2004-05-09 the dcision of equaliy is based on
      // id rather than on distance
   if (!SysControl::noInteractionChargeToWall){
      PointCharge q1=par1.getPointCharge();
         // feels the force
      PointCharge q2=par2.getPointCharge();
         // generates mirror charges
      for (Z j=1;j<=ne;j++){
         bool bj=ext(j).isActive();
         if (!bj) continue;
         R dj=ext(j).dist(q2.get_x());
         if (dj<=nearLength){
            V<PointCharge> extinfl=ext(j).influence(q2);
            res.addForce(q1.force(extinfl));
         }
      }
   }
   return res;
}

CPM_TAL
FieldSpace CPM_FCN::field(const Spc& sp, const Par& par1,
   const bool& farAway)const
{
```

```
   bool nogravity=(SysControl::noInteractionMassToMass!=0);
      // we don't want the particles to create gravitational fields
      // if masses don't interact

   FieldSpace res=(par1.getField(nogravity))(sp);
   if (!SysControl::noInteractionChargeToWall){
      PointCharge q=par1.getPointCharge();
      for (Z j=1;j<=ne;j++){
         V<PointCharge> extinfl=ext(j).influence(q);
         Z eid=extinfl.dim();
         if (eid==0) continue;
         else if(farAway){ // treat the mirror charges from one source
            // as placed at the same position of the first one
            PointCharge pc=extinfl[1];
            for (Z k=2;k<=eid;k++) pc.add(extinfl[k]);
            res.c1()+=PointCharge::electrostaticField(sp,pc);
         }
         else{
            res.c1()+=PointCharge::electrostaticField(sp,extinfl);
         }
      }
   }
   return res;
}

CPM_TAL
R CPM_FCN::potential( Spc const& sp, Par const& par1,
   bool const& farAway)const
{
   R res=par1.getU()(sp);
   if (SysControl::potentialToShow==1 &&
      !SysControl::noInteractionChargeToWall){
      PointCharge q=par1.getPointCharge();
      for (Z j=1;j<=ne;j++){
         V<PointCharge> extinfl=ext(j).influence(q);
         Z eid=extinfl.dim();
         if (eid==0) continue;
         else if(farAway){ // treat the mirror charges from one source
            // as placed at the same position of the first one
            PointCharge pc=extinfl[1];
            for (Z k=2;k<=eid;k++) pc.add(extinfl[k]);
            res+=PointCharge::electrostaticPotential(sp,pc);
         }
         else{
            res+=PointCharge::electrostaticPotential(sp,extinfl);
         }
      }
   }
   return res;
}
```

```
//////////////////////////// dynamics stuff ////////////////////////////////

CPM_TAL
set<Z> CPM_FCN::nextPointIndexes(const Spc& x, R range)
{
   set<Z> res;
   Z pld=pointList.dim();
   for (Z j=1;j<=pld;++j){
      Spc xj=pointList[j];
      R dj=(x-xj).absVal();
      if (dj<=range){
         res.insert(j);
      }
   }
   return res;
}

CPM_TAL
void CPM_FCN::prepareStep(const Camera& sc)
// to be called after redistribute, so np is the
// number of particles which are owned or used
{
   const Z mL=2;
      // normal message limit, proposal 2

   const Z mL2=4;
      // message limit for graphical activity, proposal 4

   static Word loc("CPM_FCN::prepareStep()");

   if (SysControl::noMeanFieldPar) return;
      // then all forces come directly from particles and
      // external systems.

   cpmmessage(mL,loc&" started");

   sizeAndRank();
   pointList=partition.getPoints(rank);
   Z pld=pointList.dim();
   Z i,j,k,l;
   xPar=V<Spc>(np);
   for (i=1;i<=np;i++) xPar[i]=par[i].get_x();
   nearLength=partition.getRes(rank);
   cpmmessage(mL,loc&": nearLength="&cpm(nearLength),-1);
   nearLength*=SysControl::nearFactor;
   cpmmessage(mL,loc&": nearLength after modification="
      &cpm(nearLength),-1);

   R covRad=partition.getCovRad(rank);
```

```
   R rAct=covRad+nearLength;
   vector< set<Z> > parListNeg(pld);
      // parListNeg[i] is a set {l[1],...l[n]} of particle
      // indexes such that
      // Par[l[k]], k=1,...n are the particles such that
      // | Par[l[k]].get_x()- poinList[i]|<= rAct.
      // The intention behind the choice of rAct is to implement
      // my algorithm (of 2004-05-07) that finds for each
      // x \in X all particles p for which
      // |p.get_x()-x|<= nearLength
   V< Z > nearPoint(np);
   for (i=1;i<=np;++i){
      Spc xi=xPar[i];
      R dMin=SysControl::intactRange;
      Z jMin=0;
      for (j=1;j<=pld;j++){ // save iteration
         // allows to use unchecked indexing for parListNeg
         Spc xj=pointList[j];
         R dij=(xi-xj).absVal();
         if (dij<dMin){
            jMin=j;
            dMin=dij;
         }
         if (dij<=rAct) parListNeg[j-1].insert(i);
      }
      nearPoint[i]=jMin;
   }
   cpmmessage(mL,"particle loop for parListNeg done");

   R npInv=(np==0 ? 0 : 1./np);
   nearList=vector< set<Z> >(np);
   for (i=1;i<=np;++i){
      if (cpmverbose>=mL2) cpmprogress("nearList",(i-1)*npInv);
      Spc xi=xPar[i];
      set<Z> neg=nextPointIndexes(xi,covRad);
      set<Z>::const_iterator k,l;
      for (k=neg.begin();k!=neg.end();++k){
         set<Z> negk=parListNeg[*k-1];
         for (l=negk.begin();l!=negk.end();++l){
            Spc xl=xPar[*l];
            R dil=(xi-xl).absVal();
            if (dil<=nearLength) nearList[i-1].insert(*l);
         }
      }
   }
   cpmmessage(mL,"particle loop for nearList done");

// We represent the electro-magnetic field of our np particles
// by computing suitable sources located at the few points of pointList.
// Aha! (2003-07-08) only owned particles should contribute to
```

```
// meanFieldPar, since the other ones are taken into account by other
// processes so that double-counting would result.

   Z npOwned=0;
   for (i=1;i<=np;i++) if (par[i].getOwner()==rank) ++npOwned;
   V<SourceOfVectorField> vsvf(npOwned);
   V<Z> whereToGo(npOwned);
   Z iOwned=0;
   for (i=1;i<=np;i++){
      if (par[i].getOwner()==rank){
         ++iOwned;
         vsvf[iOwned]=par[i].source();
         whereToGo[iOwned]=nearPoint[i];
      }
   }
   vsvf=SourceOfVectorField::redistribute2(vsvf,pointList,whereToGo);
      // this function does not neglect contributions of particles.
      // since each source is taken and only shifted.
      // See the declaration of
      // SourceOfVectorField::redistribute(...) in file cpmforcesx.h
      // for the meaning of this function call.
      // now we have a much smaller list sources than before since
      // typically pointList.dim() is much smaller than np
   Z nk=vsvf.dim();
   if (cpmverbose>=mL2){ // to see what we got
      // no longer very useful since the nean field can now be
      // represented
      // more conveniently via function mark(...)
      for (i=1;i<=nk;i++){
         vsvf[i].field().mark(sc);
         sc.display();
         sc.clear();
      }
   }
// it is economical to redistribute the actual sources before computing
// the mirror-charges. Notice that the mirrorcharges will never be
// redistributed so that we don't need to extend the grid into the area
// the mirror charges are located.

   if (!SysControl::noInteractionChargeToWall){
      // The point charges in vsvf should be augmented by the charges
      // which they influence in the external systems.
      for (j=1;j<=pld;j++){
         PointCharge pc=vsvf[j].getCharge();
         if (pc.get_q()==0.) continue;
            // this happens if there are points in the point list in the
            // neighborhood of which there are no particles. If these
            // charges are 0, we have not to care about mirror charges
         for (k=1;k<=ne;k++){
            V<PointCharge> pck=ext(k).influence(pc);
```

```
            Z pckd=pck.dim();
            if (pckd>0){
               PointCharge pcks=pck[1];
               for (l=2;l<=pckd;l++) pcks.add(pck[l]);
                  // it is certaily allowed to pack the induced
                  // charges (from same source charge and same external
                  // system !) together
               vsvf&=SourceOfVectorField(pcks); // array may become
               // longer!
            }
         } // k loop
      } // j loop
   }
   // now we may have much more sources than those at points of
   // pointList

   nk=vsvf.dim();

// Now we pack the information on sources in a manner
// suitable to sending to other processes.
   typedef  V<SourceOfVectorField> mesType;
   mesType valMy(vsvf);
// We should be prepared to receive quantities of type mesType from all
// other processes. So we have to have an array to receive these
// objects.
   V<mesType> valAll(size,mesType(V<SourceOfVectorField>(0)));
      // do not initialize vectors by default constructor
      // since these then have already one component which is
      // not what we want; was wrong till 02-04-04
   Cpm_com.sendAndRec(valMy,valAll);
      // responsibility for a consistent pairing of sends and
      // receives is now with class Com.
      // Now the present process has available the sources that create
      // the far field in the neighbor areas. Unlike the situation
      // for particles, these far field sources do not correspond only
      // to the physics happening in the overlap zone between the present
      // process and the neighbors but in the whole neighbor area.

   cpmmessage(mL,"communication done");

// initializing repPointList if not yet done

   if (!meanFieldAct){
      Z n2=partition.repNegDim();
      repPointList=VV<Spc>(pld,n2);
      for (i=1;i<=pld;i++){
         Spc pi=pointList[i];
         V<Spc> ppi=partition.repNeg(pi); // Note ppi[1]==pi
         for (j=1;j<=n2;j++) repPointList[i][j]=ppi[j];
      }
```

```
   }

   cpmmessage(mL,"defining meanField");

// now we have evaluate the contributions of all known sources
// not only those centered at some pointList[i] on the grid
// given by pointList.

   V<FieldSpace> filValList(pld);
   V<R> potValList(pld);

   for (i=1;i<=pld;i++){ // loop over the points where the
         // field values have to be defined
      FieldSpace vali; // starting with 0
      R valUi=0;
      Z n2=repPointList.dim2();
      for (j=1;j<=n2;j++){
         Spc xij=repPointList[i][j];
         // the virtual repetition copies for toroidal topology are taken
         // here
         for (k=1;k<=nk;k++){
            if (k!=i || j!=1){ // why j!=1
               vali+=vsvf[k].eval(xij);
               if (SysControl::showPartFields) valUi+=vsvf[k].evalU(xij);
            }
            // this excludes only the direct source at xi1
            // mirror charges from there are included, since these
            // are of the form vsvf[k] for k>pld.
            // adding fields from all sources (in rank) appart
            // the ones at xi1. Notice that it is natural to neglect
            // contributions from some sources since the nearest
            // neighbors of a particle are taken into account as direct
            // interaction partners and thus would be countet twice if
            // they would enter meanFieldPar. A reasonable definition of
            // 'nearLength' has made in the beginning aiming at making the
            // volume where sources are neglected equal to the volume in
            // with particles interact directly.
            // This is a canonical (though not trivial)
            // method of separating near field and far field.
            // This is what I call the 'self-consistent far field method'
            // In short the logic is as follows:
            // To extract a everywhere smooth field from a cloud of
            // point sources:
            // (i): chooce a lattice of points that will determine
            //    the resolution of the smooth field to be created
            // (ii) collect all sources at the lattice points by
            //    moving each source to the nearest lattice point
            // (iii) Compute the field strength in all lattice points
            //    by neglecting in each point just the contribution
            //    of the sources in just that point (which would be
```

```
            //    singular).
            // (iv) continue the field (not the sources!) from the
            //    lattice points to the whole space by whatever
            // reasonable interpolation method.
         }
      }
      filValList[i]=vali;
      if (SysControl::showPartFields) potValList[i]=valUi;
         // does neither know of other processes nor of
         // repetition
   }
   cpmmessage(mL,"defining meanField: first loop done");
// now the contribution from neighbor processes.
// No repetition copies here!

   for (i=1;i<=pld;i++){ // loop over the points where the
         // field values have to be defined
      FieldSpace exVali; // starting with 0
      R exValUi=0; // starting with 0
      Spc xi=pointList[i];
         // position where to compute the field values
         // no sum over repetition points
      for (j=1;j<=size;j++){ // loop over the contributing
            // neigbor processes
         if (j==rank) continue;
         Z nj=valAll[j].dim();
         for (k=1;k<=nj;k++){
            exVali+=valAll[j][k].eval(xi);
            if (SysControl::showPartFields)
               exValUi+=valAll[j][k].evalU(xi);
         }
      }
      filValList[i]+=exVali;
      if (SysControl::showPartFields) potValList[i]+=exValUi;
   }
   cpmmessage(mL,"defining meanField: second loop done");
   meanFieldPar=VectorField(pointList,filValList);
      // this is an interpolation constructor

   if (SysControl::showPartFields)
      meanPotPar=ScalarField(pointList,potValList);
   // meanPotPar is only needed for visualization and debugging,
   // so it should not be computed in serious production runs
   // These have to be done with
   // SysControl::showPartFields==false
   meanFieldAct=true;
   cpmmessage(mL,loc&" done");
}

//////////////////////// main dynamics stuff ////////////////////////
```

```
CPM_TAL
void CPM_FCN::steplet(R t0, R dt, const Camera& cam)
{
   const Z mL=3; // 2 usual
   const Z mL2=3;
   static Word loc("CPM_FCN::steplet(...)");
   static R dtMem=0;
   static Z npMem=0;
   cam; // presently not used
   if (dt==0.){ // this should be efficient
      cpmmessage(mL,loc&" trivially done");
      return;
   }
   bool b=false;
   if (dt!=dtMem){
      b=true;
      dtMem=dt;
   }
   if (np!=npMem){
      b=true;
      npMem=np;
   }
   if (b) cpmvalues2("steplet: dt, np",dt,np,2);

   cpmmessage(mL,"steplet(...), dt="&cpm(dt)&" np="&cpm(np));

   bool parValid=Par::isVal(par);
      // soundness check for particle array

   cpmassert(parValid==true,loc);
      // allows to stop by setting cpmdbg=2 (cpmdbg=1 is normal)

// linear evolution to the midpoint in time: direct midpoint method
// Linear means here that the velocities (linear velocity and angular
// velocity) are constant. Notice that constant angular velocity for
// a rigid body normally does not mean free motion.
   Z i,j;
   R tau=0.5*dt;
   R tc=t0+tau;
      // t center
   R tf=t0+dt;

   cpmassert(par.dim()==np,loc);

   for (i=1;i<=np;i++){
     par[i].freeMot(tc);
     par[i].phase1(tau);
   }
```

```
   V<R> te(ne,0);
   for (j=1;j<=ne;j++){
      te[j]=ext(j).get_t();
      ext(j).clear();
      ext(j).freeMot(tc);
         // see comment to ext(j).freeMot(tf)
   }

   R npInv=(np>0 ? 1./np : 0);

// collecting the interaction terms at the mid-point in time
// don't move particles yet, otherwise dynamics would depend on
// numbering of particles

   V<For> forces(np);
      // forces[i]=force acting on particle par[i]
   V<FieldSpace> fValues(np);
   //fValues[i]=field values at place of particle par[i]

   for (i=1;i<=np;i++){ // first particle loop
      if (cpmverbose>=mL2) cpmprogress("force loop",(i-1)*npInv);
      Par pi=par[i];
      For fi;
// forces by general fields
      if (SysControl::eta!=0 && ne>0 && !SysControl::noFriction)
       fi+=pi.forces(ext(1));
         // 'air friction' in the reference system of ext[1].
         // Here ext(i) enters through the base class RigidObject
      if (!SysControl::ignoreExternalSystems){
         for (j=1;j<=ne;j++){
            if (ext(j).isActive()){
               fi+=pi.forces(ext(j).getField());
            }
          }
      }
// forces by particle-generated fields
      FieldSpace fvi; // zero field
      Spc xi=pi.get_x();
      if (!SysControl::noMeanFieldPar){
         fvi=meanFieldPar(xi);
         fi+=pi.forces(fvi); //(***)
         // all influence of fvi on velocities are thus taken into
         // account; we need fvi later also as a quantity that
         // causes the electric polarization of pi to change
         // (see call of function react).
      }
// forces by interacting with particles directly
// including self-interaction:
// force on pi created by the mirror charges created by
// pi itself in external systems. Proper dependence on
```

```
// SysControl::noInteractionChargeToWall and ext(j).isActive()
// is assured by the implementation of function force()

      if (SysControl::noMeanFieldPar){ // forces from all particles
         for (j=1;j<=np;++j){
            fi+=force(pi,par[j]);
         }
      }
      else{ // forces from near particles

         set<Z>::const_iterator js;
         for (js=nearList[i-1].begin();js!=nearList[i-1].end();++js){
            fi+=force(pi,par[*js]);
            // includes forces from mirror charges, member of CPM_TAL, not
            // of Par. According to the new logic we have to include
            // mirror charges here only if we are near to a wall since
            // filValList includes all mirrorcharges on the average
         }
      }
      if (!SysControl::ignoreExternalSystems){
         for (j=1;j<=ne;j++){
            if (ext(j).fieldOnly()) continue;
            fi+=ext(j).interaction(pi,dt);
         }
      }
      forces[i]=fi;
      fValues[i]=fvi;
   } // end of first particle loop

// moving the particles according to interaction

   for (i=1;i<=np;i++){ // second particle loop
      par[i].intAct(dt,forces[i],fValues[i]);
         // Changes velocities according to forces.
         // For Par=Clust (as in class Station) this call includes
         // building up polarization of the particle through the
         // electric component of fValues[i]. See
         // Clust::intAct(R, const ForTor&, const FieldSpace&)
         // and void Clust::setElcDipole(R dt, const VecE& E);
      par[i].freeMot(tf);
         // executes the shift in position and angular orientation
         // according to the full acceleration and angular acceleration
         // that the particle feels (no regularization) at this place.
      par[i].reg();
         // regularization of the motion by reducing the velocity
         // if this is much larger than SysControl::vCrit
   } // end of second particle loop

   if (!SysControl::ignoreExternalSystems){
      for (j=1;j<=ne;j++) ext(j).freeMot(tf);
```

```
        // linear evolution at the end. Also
        // inactive systems should do this since
        // they should be in the right place if reactivated
   }

   cpmmessage(mL,loc&" done");
}

CPM_TAL
void CPM_FCN::step(R dt, const Camera& cam)
{
   const Z mL=2;
   Z fS=SysControl::fixedStep;
   SysControl::tStep=dt;

   static Z stepCount=0;
   R tcStart,tcPrepareDone,tcFinal;
      // tc : computation time
   R tmStart=get_t(),tmFinal;
     // tm : modeling time = image of real time in our model
   Z nStart=get_np(), nFinal;

   if (cpmverbose>0){
      tcStart=cpmtime();
   }
   cpmmessage(mL,"CPM_FCN::step("&cpm(dt)&") started");
   redistribute();
   prepareStep(cam); // was missing till 02-06-14
      // we need to take into account that interaction with other
      // processes at the end of step changed everything in the present
      // process (e.g. the particle number) in a manner that could not be
      // predicted within the given process. We also would not rely on
      // a constructor to ensure the preconditions for function step().

   if (cpmverbose>0){
      tcPrepareDone=cpmtime();
   }

   if (nStart==0){
      cpmmessage("step() started with no particles");
      steplet(tmStart,dt,cam);
      done(dt,0);
   }
   else{
      R dt_fS=dt/fS;
      for (Z j=1;j<=fS;j++){
         if (fS>0 && cpmverbose>1)
            cpmvalue("frac",(j-R(1.))/(fS-R(1.)),4);
         R t=get_t();
         R tf=t+dt_fS;
```

```
         steplet(t,dt_fS,cam);
         done(dt_fS,0); // faster than done(0)
      }
      if (getRelaxFlag()) stopParticles();
   }

   tcFinal=cpmtime();
   tmFinal=get_t();
   set_t(tmFinal); // to be sure that all internal times are equal
   if (cpmverbose>1){ // limit was 0 till 2005-09-19, however under Linux
      // writing long messages to file causes irregular pauses in the
      // video output which now can be avoided by setting cpmverbose to 1
      stepCount++;
      tcFinal=cpmtime();
      nFinal=get_np();
      ostringstream ost;
      ost<<endl<<"Performance data for step # "<<stepCount<<endl;
      ost<<"n: particle number: prior to step "
         <<nStart<<", after step "<<nFinal<<endl;
      R tcTotal=tcFinal-tcStart;
      R tcTotalInv=tcTotal>0 ? 1./tcTotal : 0.;
         // due to the limited resolution of the internal clock
         // tcTotal may well come out as 0, this is not to be
         // considered an situation to warn
      R tcPre=tcPrepareDone-tcStart;
      R percentPrepare=100*tcPre*tcTotalInv;
      R tmTotal=tmFinal-tmStart;
      ost<<"Point in model time at beginning of step: "<<tmStart<<endl;
      ost<<"tc: computation time/s for step: "<<tcTotal<<endl;
      ost<<"actual model time/s for step: "<<tmTotal<<endl;
      ost<<"tm: scheduled model time/s for step: "<<dt<<endl;

      if (tcTotal>0){
         ost<<"preparation phase in percent: "<<percentPrepare<<endl;
         R eff=(tmTotal*nStart)*tcTotalInv;
         ost<<"efficiency=(tm*n)/tc= "<<eff;
      }
      else{
         ost<<"preparation phase in percent undefined"<<endl;
         ost<<"efficiency undefined since computation time"
            <<" unmeasurably small";
      }
      cpmmessage(1,Word(ost.str()),-1); // don't write to the status bar
   }
   cpmmessage(mL,"CPM_FCN::step("&cpm(dt)&") done");
}

#if !defined(CPM_GUI)

CPM_TAL
```

```
bool CPM_FCN::run(R dtStep, Z nSteps,
   Z filingPeriod, Z docuPeriod, Z viewingPeriod, Z stoppingPeriod,
   const Word& movieName, Camera& cam, const Timer& tm,
   const F<R,Z>& f, OFileStream& ofs)
      // was const F<R,V<R> >& f till 02-06-03
      // return value not needed (void does not work with my function
      // objects since 'unknown type' for the MS compiler
      // Presently f does some manipulations of SysControl
      // as a function of time.
{
   static SysControlState sc;
   static Word loc("CPM_FCN::run(...)");
   const Z mL=2;
   CPM_MA
   const Z mL2=3;
   bool earlyStop=false;
   if (nSteps<0 || dtStep<0){
      cpmmessage(mL,loc&" trivially done");
      return earlyStop;
   }
   sc.mem(); // memorizing the state of SysControl since
      // SysControl may be changed by calling f
   CyclicAlarm ca1(filingPeriod,viewingPeriod);
   CyclicAlarm ca2(viewingPeriod,viewingPeriod);
   CyclicAlarm ca3(stoppingPeriod,stoppingPeriod);
   CyclicAlarm ca4(docuPeriod,docuPeriod); // introduced 2005-03-24
      // so the next call to operator() will give true
      // unless ...Period<1
   Z count=0;

   if (ca1()){
      earlyStop=file(movieName,tm);
      cpmmessage("frame written at t="&cpm(get_t()));
   }
   if (ca2()){ // viewing is requested
      if (!meanFieldAct) prepareStep(cam);
         // otherwise the fields meanFieldPar and meanPotPar
         // would have zero-values in the first viewing frame that
         // preceeds all calls to step(). Notice that it is extremely
         // useful to get a display even for nSteps==0 since one
         // can so inspect particle-rich configurations for which a
         // step would take a prohibitively long computation time.
      mark(cam,true);
      cpmwait(SysControl::tWait);
   }
   if (nSteps==0 ){
      cpmmessage(mL,loc&" trivially done due to nSteps==0");
      return earlyStop;
   }
   if (earlyStop){
```

```
      cpmmessage(mL,loc&" trivially done due to earlyStop");
      return earlyStop;
   }
   if (!meanFieldAct) prepareStep(cam);
   R tFinal=dtStep*nSteps;
   R tFinalInv=1./tFinal;
   R nStepsInv=1./nSteps;
   R tAct=get_t();
   for (Z is=1;is<=nSteps;++is){
      R glob_prog=tAct*tFinalInv;
      cpmprogress("Progress",glob_prog,3,3);
      if (cpmverbose>0)
         cpmcerr<<endl<<loc<<" global progress ="<<glob_prog;
         // very important since tasks can be stopped early via time-out
         // should not interfere on the status bar with cpmprogress
      f(tAct);
      cpmmessage(mL2,"vCrit="&cpm(SysControl::vCrit));
      cpmmessage(mL2,"aCrit="&cpm(SysControl::aCrit));
      step(dtStep,cam);
      tAct=get_t();
      count++;
      if (ca3()) stopParticles();
         // this is considered a part of the intended dynamics
         // thus it should preced documentation
      if (ca4()) docuStep(ofs);
      if (ca1()){
         earlyStop=file(movieName,tm);
         cpmmessage(1,"**** frame written at t="&
            cpm(get_t())&" count="&cpm(count));
      }
      if (ca2()){
         mark(cam,true);
         cpmwait(SysControl::tWait);
      }

      if (earlyStop) break;
   }
   sc.act();
      // restoring the state of SysControl which may have been
      // changed by the action of f
   cpmmessage(mL,loc
      &" done, earlyStop="&cpm(earlyStop)
      &", tAct="&cpm(tAct)
      &", tFinal="&cpm(tFinal)
      &", count="&cpm(count)
   );
   return earlyStop;
}

#else // CPM_GUI
```

```
// not yet tested

CPM_TAL
bool CPM_FCN::run(R dtStep, Z nSteps,
   Z filingPeriod, Z docuPeriod, Z viewingPeriod, Z stoppingPeriod,
   const Word& movieName, Camera& cam, const Timer& tm,
   const F<R,Z>& f, OFileStream& ofs)
// made by some obvious modifications from the (once working) guipadyan
// program.
// Not yet tested in this form.
// It is importanr that the function is done after having called step
// once. Then the program is free to react to, say, keys.
 {
   static Word loc("CPM_FCN::run(...)");
   static bool firstrun=true;
   static R tFinal;
   static CyclicAlarm ca1,ca2,ca3,ca4;
   static Z count=0;
   static bool earlyStop=false;

   const Z mL=2;
   const Z mL2=2;

   if (firstrun){
      set_dt(dt);
      tFinal=dt*nSteps;
      ca1=CyclicAlarm(filingPeriod,filingBurstPeriod,filingBurstLength);
      ca2=CyclicAlarm(viewingPeriod,viewingPeriod);
      ca3=CyclicAlarm(stoppingPeriod,stoppingPeriod);
      ca4=CyclicAlarm(docuPeriod,docuPeriod);
      if (ca4()) docuStep(ofs);
      if (ca1()){
         earlyStop=file(movieName,tm);
         cpmmessage("frame written at t="&cpmwrite(get_t()));
      }
      if (ca2()){
         if (!meanFieldAct) prepareStep(cam);
         // otherwise the fields meanFieldPar and meanPotPar
         // would have zero-values in the first viewing frame that
         // preceeds all calls to step(). Notice that it is extremely
         // useful to get a display even for nSteps==0 since one
         // can so inspect particle-rich configurations for which a
         // step would take a prohibitively long computation time.
         mark(cam);
      }
      firstrun=false;
   } // firstrun
   R dtstep=get_dt();
   cpmmessage(mL,loc&" started");
   if (nSteps<0 || dtstep<0){
```

```
      cpmmessage(mL,loc&" trivially done");
      return earlyStop;
   }
   if (nSteps==0 ){
      cpmmessage(mL,loc&" trivially done due to nSteps==0");
      return earlyStop;
   }
   if (earlyStop){
      cpmmessage(mL,loc&" trivially done due to earlyStop");
      return earlyStop;
   }
   if (!meanFieldAct) prepareStep(cam);
   R tAct=get_t();
   R glob_prog=1;
   if (tAct>=tFinal) goto Label;
   glob_prog=tAct/tFinal;
   cpmprogress(loc,glob_prog);
   if (cpmverbose>0)
      cpmcerr<<endl<<loc<<" global progress = "<<glob_prog;
   step(dtStep,cam);
   count++;
   if (ca4()) docuStep(ofs);
   if (ca1()){
         earlyStop=file(movieName,tm);
         cpmmessage(1,"**** frame written at t="&
            cpmwrite(get_t())&" count="&cpmwrite(count));
   }
   if (ca2()) mark(cam);
   if (ca3()) stopParticles();
   if (earlyStop) goto Label;
   return false; // interactive experiment
Label:
   cpmmessage(mL,loc
      &" done, earlyStop="&cpmwrite(earlyStop)
      &", tAct="&cpmwrite(tAct)
      &", tFinal="&cpmwrite(tFinal)
      &", count="&cpmwrite(count)
   );
   return earlyStop;
}

#endif

CPM_TAL
Vp<CPM_FCN> CPM_FCN::trj(R dtStep, Z nSteps, Camera& cam,
   Z viewingPeriod)
{
   static Word loc("CPM_FCN::trj(...)");
   const Z mL=2;
   CPM_MA
```

```
   bool trivial=nSteps<0 || dtStep<0;
   Z resDim=trivial ? 0 : nSteps+1;
   Vp<CPM_FCN> res(resDim);
   if (trivial){
      cpmmessage(mL,loc&" trivially done");
      return res;
   }
   CyclicAlarm ca(viewingPeriod,viewingPeriod);
   if (ca()){ // viewing is requested
      if (!meanFieldAct) prepareStep(cam);
         // otherwise the fields meanFieldPar and meanPotPar
         // would have zero-values in the first viewing frame that
         // preceeds all calls to step(). Notice that it is extremely
         // useful to get a display even for nSteps==0 since one
         // can so inspect particle-rich configurations for which a
         // step would take a prohibitively long computation time.
      mark(cam,true);
      cpmwait(SysControl::tWait);
   }
   if (nSteps==0 ){
      cpmmessage(mL,loc&" trivially done due to nSteps==0");
      return res;
   }
   if (!meanFieldAct) prepareStep(cam);
   R prog=0;
   cpmprogress("Progress",prog,3,3);
   R dprog=1./nSteps;
   res[1]=*this;
   for (Z i=1;i<=nSteps;++i){
      step(dtStep,cam);
      res[i+1]=*this;
      prog+=dprog;
      cpmprogress("Progress",prog,3,3);
      if (ca()){
         mark(cam,true);
         cpmwait(SysControl::tWait);
      }
   }
   CPM_MZ
   return res;
}

CPM_TAL
void CPM_FCN::movieFromTrj(Vp<CPM_FCN> const& trj, Word const& fileName,
   Camera& cam, Z viewingPeriod)
{
   Timer tm;
   CyclicAlarm ca(viewingPeriod,viewingPeriod);
   Z i,n=trj.dim();
   Pp<CPM_FCN> sMem(*this);
```

```
   Word header("Progress");
   Z seg=3;
   Z acc=3;
   R prog=0;
   cpmprogress(header,prog,seg,acc);
   R incr=1./n;
   for (i=1;i<=n;++i){
      *this=trj(i);
      file(fileName,tm);
         // I presently don't understand why
         // trj(i).file(fileName,tm);
         // does not work; why the detour over *this is needed
         // (notice that only derived classes implement function
         // file(...)
         // non-trivially
      prog+=incr;
      cpmprogress(header,prog,seg,acc);
      if (ca()){
         cam.clear();
         mark(cam,true);
         cpmwait(SysControl::tWait);
      }
   }
   *this=sMem();
}

CPM_TAL
R_Vector CPM_FCN::anlOrd(R dtStep, Z nSteps, Z nFrames, Z fac,
   Word const& runId, Camera& cam, Iv const& iv, Z repLog,
   R repOrderD, R repOrderE)
{
   Z mL=2;
   Word loc("CPM_FCN::anlOrd(...)");
   CPM_MA
   cpmassert(dtStep>0,loc);
   cpmassert(nFrames>=2,loc);
   cpmassert(nSteps>=1,loc);
   Z fsMem=SysControl::fixedStep;
   cpmassert(fsMem>=1,loc);

   Vp<CPM_FCN> finalStates(nFrames);
   Pp<CPM_FCN> sMem(*this);
   Vp<CPM_FCN> vs0=trj(dtStep,nSteps,cam);
   Word header("Total");
   Z seg=4;
   Z acc=2;
   R eps=1e-16;
   R incr=1./nFrames;
   R glob_prog=incr;
   cpmprogress(header,glob_prog,seg,acc);
```

```
   Z vsd=vs0.dim();
   finalStates[1]=vs0[vsd];
      // dimension is nSteps+1 since initial state is vs0(1)
   R eTot0=vs0(1).totEnr();
      // total energy of initial state
   cpmassert(eTot0!=0,loc);
   R eTot0Inv=1./eTot0;
   Z i,j;
   R_Vector dt(nFrames); // holds the time step for each trajectory
      // which is needed for the computation of the order
   dt[1]=dtStep/fsMem;
   R_Matrix dd(nFrames,nSteps);
      // for each of the nFrames-1 differences between trajectories and
      // one
      // dimension
      // more for the x-values
   R_Matrix enr(nFrames+1,nSteps);
      // for each trajectory and one dimension more for the x-values
   for (j=1;j<=nSteps;++j){
      R tj=vs0(j).get_t();
      dd[1][j]=tj;
      enr[1][j]=tj;
      R e2j=(vs0(j+1).totEnr()-eTot0)*eTot0Inv;
      enr[2][j]=e2j;
   }
   R facInv=1./fac;
// main loop that successively increases the number of steplet calls
// in a step
   Z iMid=cpmround((1.1+nFrames)*0.5); // the larger if nFrames is even
   Word fileMid("midMovie");
   Word graphMid("midGraph");
   fileMid&=runId;
   fileMid&=".txt";
   graphMid&=runId;
   graphMid&=".txt";
   Z viewingPeriod=4; //
   for (i=2;i<=nFrames;++i){
      *this=sMem();
      SysControl::fixedStep*=fac;
      dt[i]=dt[i-1]*facInv;
      Vp<CPM_FCN> vsi=trj(dtStep,nSteps,cam);
      if (i==iMid){
         movieFromTrj(vsi,fileMid,cam,viewingPeriod);
      }
      finalStates[i]=vsi[vsd];
      glob_prog+=incr;
      cpmprogress(header,glob_prog,seg,acc);
      for (j=1;j<=nSteps;++j){
         R xij=vs0(j).disVal(vsi(j));
         dd[i][j]=xij;
```

```
      R eij=(vsi(j+1).totEnr()-eTot0)*eTot0Inv;
      enr[i+1][j]=eij;
   }
   vs0=vsi;
}
cpmvalues2("accMem, newRigBodyInt",
   R(SysControl::accMem),R(SysControl::newRigBodyInt),4);

Word fileFS("fsMovie");
fileFS&=runId;
fileFS&=".txt";

viewingPeriod=1;
movieFromTrj(finalStates,fileFS,cam,viewingPeriod);
R orderE=0;
R orderD=0;
bool ivv=iv.isVoid();
if (!ivv){ // then we know from which piece of the trajectory to infer
   // an estimate for order
   if (repOrderD<=0){ // if there is no valid input, we have to
         // determine a reasonable order from data
      R_Vector xd(nFrames-1);
      R_Vector yd(nFrames-1);
      for (i=1;i<nFrames;++i){
         R a1=::log10(dt[i]);
         R a2=::log10(dt[i+1]);
         xd[i]=0.5*(a1+a2);// the difference
         // curve belongs to the mean value of step size logarithms
         yd[i]=::log10(eps+dd[i+1].cutCopy(iv).absVal());
      }
      R_Vector pd=CpmArrays::polyFit(xd,yd,1,0,1,false);
      orderD=pd[4];
   }
   if (repOrderE<=0){
      R_Vector xe(nFrames);
      R_Vector ye(nFrames);
      for (i=1;i<=nFrames;++i){
         xe[i]=::log10(dt[i]);
         ye[i]=::log10(eps+enr[i+1].cutCopy(iv).absVal());
      }
      R_Vector pe=CpmArrays::polyFit(xe,ye,1,0,1,false);
      // here order of polynomial=1, no autoadjustment so that the
      // order
      // is directly the highest coefficient of the polynomial
      orderE=pe[4];
   }
}
if (repOrderD>0) orderD=repOrderD;
if (repOrderE>0) orderE=repOrderE;
   // meaningful input values have the priority
```

```
   R_Vector facRepD(nFrames-1,1.);
   if (orderD>0){
      // holds factors which help to collapse a family of D curves into
      // a single one if the order of the integrator equals orderD
      R pD=::pow((R)fac,orderD);
      for (i=2;i<nFrames;++i) facRepD[i]=facRepD[i-1]*pD;
   }
   R_Vector facRepE(nFrames,1.);
      // holds factors which help to collapse a family of E curves into
      // a single one if the order of the integrator has order orderE
      // of the energy error
   if (orderE>0){
      R pE=::pow((R)fac,orderE);
      for (i=2;i<=nFrames;++i) facRepE[i]=facRepE[i-1]*pE;
   }
   R_Matrix ddLog(nFrames,nSteps);
   R_Matrix enrLog(nFrames+1,nSteps);
   for (j=1;j<=nSteps;++j){
      R ti=dd[1][j];
      ddLog[1][j]=ti;
      enrLog[1][j]=ti;
   }
   for (i=2;i<=nFrames;++i){
      R fi=facRepD[i-1];
      for (j=1;j<=nSteps;++j){
         dd[i][j]*=fi;
         ddLog[i][j]=::log10(cpmabs(dd[i][j])+eps);
      }
   }
   for (i=2;i<=nFrames+1;++i){
      R fi=facRepE[i-1];
      for (j=1;j<=nSteps;++j){
         enr[i][j]*=fi;
         enrLog[i][j]=::log10(cpmabs(enr[i][j])+eps);
      }
   }

// Writing data to file in a format suitable to make gnuplot
// diagrams

   Word fileD=runId&"D.txt";
   OFileStream ofsD(fileD);
   for (i=1;i<=nSteps;++i){
      ofsD()<<endl;
      for (j=1;j<=nFrames;++j)ofsD()<<" "<<dd[j][i];
   }

   Word fileDL=runId&"DL.txt";
   OFileStream ofsDL(fileDL);
   for (i=1;i<=nSteps;++i){
```

```
      ofsDL()<<endl;
      for (j=1;j<=nFrames;++j)ofsDL()<<" "<<ddLog[j][i];
   }

   Word fileE=runId&"E.txt";
   OFileStream ofsE(fileE);
   for (i=1;i<=nSteps;++i){
      ofsE()<<endl;
      for (j=1;j<=nFrames+1;++j)ofsE()<<" "<<enr[j][i];
   }

   Word fileEL=runId&"EL.txt";
   OFileStream ofsEL(fileEL);
   for (i=1;i<=nSteps;++i){
      ofsEL()<<endl;
      for (j=1;j<=nFrames+1;++j)ofsEL()<<" "<<enrLog[j][i];
   }

   Frame fra;
   fra.paint(Color(LIGHTBLUE));
   Frames frs(fra,Z(1),Z(2));
   VV<Frame> vvf=frs;
   V<Frame> vf=vvf[1];
   Graph g11(vf[1]);
   Graph g12(vf[2]);

   g11.addText("ordD="&cpm(orderD),0.5);
   if (repLog) g11.show(ddLog); else g11.show(dd);

   g12.addText("ordE="&cpm(orderE),0.5);
   if (repLog) g12.show(enrLog); else g12.show(enr);
   g12.display();

   SysControl::fixedStep=fsMem;
   *this=sMem();
   R_Vector res(2);
   res[1]=orderD;
   res[2]=orderE;
   CPM_MZ
   return res;
}

CPM_TAL
void  CPM_FCN::redistribute(void)
{
   Z mL=2;
   Z mL2=3;
   sizeAndRank();
   Word loc("CPM_FCN::redistribute()");
   bool isTorus=partition.torus();
```

```
if (size==1 && !isTorus) return; // nothing to redistribute then
// notice that toroidal topology needs some redistribution
// even for size==1
   // added 2004-05-11 and 2004-05-18
cpmmessage(mL,loc&" started, np="&cpm(np));
const R slowDownFactor=0.9;
Z i,j;
V< V<Par> > groupsToSend(size,V<Par>(0));
V< V<Par> > groupsToReceive(size,V<Par>(0));
for (i=1;i<=np;i++){ // each rank loops over the particles that are
   // visible to it
   Z oi=par[i].getOwner();
      // ownership by presently valid assignment
   if (oi==rank){ // we dont collect the 'virtual' particles
      // moreover we collect only those particles which we own
      Par pi=par[i];
      Spc xi=pi.get_x();
      X3<Z,Group,B> xzg=partition.owner(xi);
      oi=xzg.c1();
         // owner based on present position, which is the one that
         // matters
      Group gi=xzg.c2();
      bool slowDown=xzg.c3();
      pi*=gi;
         // so far this is OK for toroidal topology. But also in
         // the non-toroidal case one should have means to bring a
         // particle
         // back to X if it has left it by action of steplet(...)
         // No problem, in this case the wall forces have to do the
         // job!
      if (slowDown){
         bool selective=true; // influences only velocity in
         // 3-direction
         pi.stop(slowDownFactor,selective);
      }
      pi.setOwner(oi);
      groupsToSend[oi].push_back(pi);
         // for oi==rank this is also OK
   }
}
// Thus the assigned owner decides about the new ownership (i.e. the
// one after function call)

if (cpmverbose>=mL2){ // diagnosis concerning groupsToSend and
   //groupsToReceive
   Z dimS=groupsToSend.dim(), dimR=groupsToReceive.dim();
   Z dimSumS=0, dimSumR=0;
   for (i=1;i<=size;++i){
      dimSumS+=groupsToSend[i].dim();
      dimSumR+=groupsToReceive[i].dim();
```

```
      }
      cpmmessage(loc&": size="&cpm(size));
      cpmmessage(loc&": dimS="&cpm(dimS));
      cpmmessage(loc&": dimR="&cpm(dimR));
      cpmmessage(loc&": dimSumS="&cpm(dimSumS));
      cpmmessage(loc&": dimSumR="&cpm(dimSumR));
   }
   Cpm_com.exchange(groupsToSend,groupsToReceive);
   par=groupsToReceive[rank];
      // these particles have positions that where held by particles
      // of rank before
   for (i=1;i<=size;i++){
      if (i!=rank) par&=groupsToReceive[i];
   }
      // this adds particles that might overlap with existing particles
      // and could cause desaster.
   np=par.dim();
   for (i=1;i<=np;i++) cpmassert(par[i].getOwner()==rank,loc);
// now each process has all the particles which he owns. It now waits
// for those that he has to use only.

   V< V<Par> > groupsToSend2(size,V<Par>(0));
   V< V<Par> > groupsToReceive2(size,V<Par>(0));

   typedef X2<Z1,Group> ZG;

   for (i=1;i<=np;i++){
      Par pi=par[i];
      groupsToSend2[rank].push_back(pi);
         // needed in order for following statement (*) to work
      Spc pix=pi.get_x(); // only the position matters
      V<ZG> vzg=partition.toSend(rank,pix);
      Z vd=vzg.dim();
      for (j=1;j<=vd;j++){
         ZG zgj=vzg[j];
         Z ij=zgj.c1();
         Group gj=zgj.c2();
         Par pij=pi;
         pij*=gj;
         pij.setOwner(0);
         groupsToSend2[ij].push_back(pij);
      }
   }

   Cpm_com.exchange(groupsToSend2,groupsToReceive2);

   par=groupsToReceive2[rank]; // (*)
   for (i=1;i<=size;i++){
      if (i!=rank) par&=groupsToReceive2[i];
   }
```

```
      // again, this adds particles that might overlap with existing
      // particles and could cause desaster.
   np=par.dim();
   cpmmessage(mL,loc&" done, np="&cpm(np));
}

//////////////////////////// end of main dynamics stuff ////////////////////

// getting VectorField and ScalarField

// external fields

CPM_TAL
VectorField CPM_FCN::getFieldExt()const
{
   VectorField res;
   if (!SysControl::ignoreExternalSystems)
      for (Z i=1;i<=ne;i++) res+=ext(i).getField();
   return res;
}

CPM_TAL
ScalarField CPM_FCN::getPotExt()const
{
   ScalarField res;
   if (!SysControl::ignoreExternalSystems)
      for (Z i=1;i<=ne;i++) res+=ext(i).getU();
   return res;
}

// fields of particles. These have the mean field alternative.
// This alternative becomes active only for graphical
// actions. The internal dynamics does not use
// meanPotPar and uses meanFieldPar directly where needed.

CPM_TAL
VectorField CPM_FCN::getFieldPar(bool mean)const
{
   Z mL=3;
   Word loc("CPM_FCN::getFieldPar(mean)");
   CPM_MA
   VectorField res;
   if (mean){
      if (!meanFieldAct){
         cpmerror(loc&": mean field not initialized");
      }
      else
         res=meanFieldPar;
   }
   else{
```

```
      V<VectorField> vf(np);
      for (Z i=1;i<=np;i++) vf[i]=fieldPar(par[i]);
      res=VectorField::sum(vf);
   }
   CPM_MZ
   return res;
}

CPM_TAL
ScalarField CPM_FCN::getPotPar(bool mean)const
{
   Z mL=3;
   Word loc("CPM_FCN::getPotPar(bool)");
   CPM_MA
   ScalarField res;
   if (mean){
      if (!meanFieldAct){
         cpmerror(loc&": mean field not initialized");
      }
      else
         res=meanPotPar;
   }
   else{
      V<ScalarField> vf(np);
      for (Z i=1;i<=np;i++) vf[i]=potentialPar(par[i]);
      // for (Z i=1;i<=np;i++) vf[i]=par[i].getU(); is OK
      // if there are no mirror charges
      res=ScalarField::sum(vf);
   }
   CPM_MZ
   return res;
}

// combining external and particle fields

CPM_TAL
VectorField CPM_FCN::getFieldTot(bool mean)const
{
   return getFieldExt()+getFieldPar(mean);
}

CPM_TAL
ScalarField CPM_FCN::getPotTot(bool mean)const
{
   return getPotExt()+getPotPar(mean);
}

// marking fields

CPM_TAL
```

```
void CPM_FCN::markPotExt(const Camera& sc)const
{
   Z mL=3;
   Word loc("CPM_FCN::markPotExt(Camera)");
   CPM_MA
   getPotExt().mark(sc);
   CPM_MZ
}


CPM_TAL
void CPM_FCN::markPotPar(const Camera& sc)const
{
   Z mL=3;
   Word loc("CPM_FCN::markPotPar(Camera)");
   CPM_MA
   bool mean=SysControl::showMeanFil;
   getPotPar(mean).mark(sc);
   CPM_MZ
}

CPM_TAL
void CPM_FCN::markPotTot(const Camera& sc)const
{
   Z mL=3;
   Word loc("CPM_FCN::markPotPar(Camera)");
   CPM_MA
   bool mean=SysControl::showMeanFil;
   getPotTot(mean).mark(sc);
   CPM_MZ
}


CPM_TAL
void CPM_FCN::markPot(const Camera& sc)const
{
   Z mL=3;
   Word loc("CPM_FCN::markPot(Camera)");
   CPM_MA
   if(SysControl::showPartFields)
      getPotTot(SysControl::showMeanFil).mark(sc);
   else
      getPotExt().mark(sc);
   CPM_MZ
}


CPM_TAL
void CPM_FCN::markField(const Camera& sc)const
{
```

```
   Z mL=3;
   Word loc("CPM_FCN::markField(Camera)");
   CPM_MA
   if(SysControl::showPartFields)
      getFieldTot(SysControl::showMeanFil).mark(sc);
   else
      getFieldExt().mark(sc);
   CPM_MZ
}

CPM_TAL
void CPM_FCN::markScalarValue(const Word& w,const Camera& sc)const
{
   Z mL=3;
   Word loc("CPM_FCN::markScalarValue(Camera)");
   CPM_MA
   ParticleViewer<Par,Spc,Camera> pv(par);
   Z ny=SysControl::nyScalarFields;
   bool pixelized=false;
   pv.showScalarValue(w,sc,ny,pixelized);
   CPM_MZ
}

CPM_TAL
void CPM_FCN::markCharge(const Camera& sc)const
{
   Z mL=3;
   Word loc("CPM_FCN::markCharge(Camera)");
   CPM_MA
   ParticleViewer<Par,Spc,Camera> pv(par);
   Z ny=SysControl::nyScalarFields;
   bool pixelized=false;
   pv.showCharge(sc,ny,pixelized);
   CPM_MZ
}

// marking the whole system

CPM_TAL
void CPM_FCN::mark(const Camera& sc, bool display,
   const Word& text)const
{
   Z mL=3;
   cpmmessage(mL,"CPM_FCN::mark(Camera,bool,Word,Z) started, np="&
      cpm(np));
   Z i,id;
   Color cAux(WHITE);
   Z bms=SysControl::bitmapToShow;
   Z scd=showColor.dim();
```

```
// first graphical layer: area-filling bitmap for bms!=0
   if (bms==0) ; // no bitmap layer
   else if (bms==1) markPot(sc);
   else if (bms==3) markCharge(sc); // traditional
   else{
      Word w;
      if (bms==2) w="mass";
      else if (bms==4)  w="velocity";
      else if (bms==5)  w="angular_velocity";
      else if (bms==6)  w="kinetic_energy";
      else if (bms==7)  w="electric_dipole";
      else if (bms==8)  w="electric_polarization";
      else if (bms==9)  w="magnetic_dipole";
      else if (bms==10) w="volume";
      else if (bms==11) w="surface";
      else w="nonsense";
      markScalarValue(w,sc); // fails for 'nonsense'
   }
// second layer: vector graphics showing particles and field
// vectors
   if (SysControl::showParticles){ // showing the mean field
         // works better if the particles are not shown
      if (scd>0){
         for (i=1;i<=np;i++){
            id=par[i].get_id();
            Color ci=showColor(id) ;
            // If the particle manifold resulted from a larger one by
            // selection, id needs not to be in the range 1,...,np
            if (ci==Color()) par[i].mark(sc); else par[i].mark(sc,ci);
         }
      }
      else{
         for (i=1;i<=np;i++){
            Z ovi=par[i].getOwner();
            if (ovi<=0) par[i].mark(sc,cAux);
            else par[i].mark(sc);
         }
      }
   }

   if (SysControl::markElectricField>=0
         || SysControl::markMagneticField>=0
         || SysControl::markGravityField>=0 ) markField(sc);

   if (!SysControl::ignoreExternalSystems)
      for (i=1;i<=ne;i++) ext(i).mark(sc);

   if (SysControl::showPartition) partition.mark(sc);

   R xT=0, yT=1;
```

```
   Color colT(Z(200),Z(200),Z(200));
   if (display){
      sc.addText(text,xT,yT,colT);
      sc.display();
      // xT=0, yT=1 sets the text to the left upper corner (i.e. so that
      // the left upper corner of the text box coincides with the
      // left upper corner of the frame rectangle) which is presumably
      // the least obstructing position
   }

   cpmmessage(mL,"CPM_FCN::mark(Camera,...) done, np="&cpm(np));
}

// most advanced lambda abstraction within a template class:

//fieldPar(...)
CPM_TAL
FieldSpace fieldParFunc
(const Spc& x, const CPM_FCN &th, const Par& p, const B& b)
{ return th.field(x,p,b);}


CPM_TAL
VectorField CPM_FCN::fieldPar(const Par& par1, bool farAway)const
{
   return
   F3<Spc,CPM_FCN,Par,B,FieldSpace>(*this,par1,B(farAway))(fieldParFunc);
}

// potentialPar(...)
CPM_TAL
R potentialParFunc
(const Spc& x, const CPM_FCN &th, const Par& p,
   const B& b)
{ return th.potential(x,p,b);}

CPM_TAL
ScalarField CPM_FCN::potentialPar(const Par& par1, bool farAway)const
{
   return
   F3<Spc,CPM_FCN,Par,B,R>(*this,par1,B(farAway))
      (potentialParFunc);
}

#undef CPM_CN
#undef CPM_TAL
#undef CPM_FCN

} // namespace
```

```
#endif
```

134 cpmpardynsys.cpp

```
//? cpmpardynsys.cpp
//? Status of work 2008-10-25.
//?
#include <cpmpardynsys.h>

using namespace CpmRoot;

void CpmDynSys::aux::adjust_dt(R& dt, Z counter)
{
   const R incr1=2;
   const R incr2=sqrt(incr1);
   const R incr1Inv=1./incr1;
   const R incr2Inv=1./incr2;

   if (counter<=1) dt*=incr1;
   else if (counter==2) dt*=incr2;
   else if (counter<=4) ;
   else if (counter<=8) dt*=incr2Inv;
   else  dt*=incr1Inv;
}
```

135 cpmparens.h

```
//? cpmparens.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_PARENS_H_
#define CPM_PARENS_H_
/*
   Description:  see file cpmparensx.h
*/
#include <cpmcompar.h>

#if defined(CPM_DIMDEF2)

#define CPM_DIM CpmDim2
#define CPM_CAMERA CpmCamera2
#define CPM_PHYSICS CpmPhysics2
#define CPM_RIGID_BODY CpmRigidBody2

namespace CPM_RIGID_BODY {
   #include <cpmparensx.h>
}

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS
#undef CPM_RIGID_BODY

#endif

#if defined(CPM_DIMDEF3)

#define CPM_DIM CpmDim3
#define CPM_CAMERA CpmCamera3
#define CPM_PHYSICS CpmPhysics3
#define CPM_RIGID_BODY CpmRigidBody3

namespace CPM_RIGID_BODY {
   #include <cpmparensx.h>
}

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS
#undef CPM_RIGID_BODY

#endif

#endif // guard
```

136 cpmparensx.h

```
//? cpmparensx.h
//? Status of work 2008-10-25.
//?
/*
   Description: Declares a class ParEns for describing
      ensembles of particles.
*/

using namespace CpmRoot;

using CPM_DIM::Vec;
using CPM_DIM::AxVec;
using CPM_DIM::Spc;
using CPM_DIM::Group;
using CPM_DIM::SubsetOfSpace;
using CPM_DIM::Rdim;
using CPM_DIM::Rdim_;
using CPM_DIM::Box;

using CpmRootX::R1;
using CpmRootX::Z1;
using CpmRootX::B;
using CpmRootX::RecordHandler;
using CpmProperties::SysControl;
using CpmProperties::BodyProperties;
using CpmProperties::ContactProperties;
using CpmProperties::ParticleProperties;
using CpmProperties::ParEnsPrp;
using CpmProperties::DumbbellData;
using CpmProperties::NuggetData;
using CpmProperties::PairData;
using CpmLinAlg::R2;
using CpmArrays::V;
using CpmArrays::X2;
using CpmArrays::X4;
using CpmArrays::R_Vector;
using CpmArrays::R_Matrix;
using CpmFunctions::F;
using CpmGeo::Iv;

//doc namespace CpmRigidBodyx{
//////////////////////////// class ParEns ////////////////////////////////
   template <class ParT>
   class ParEns { // particle ensemble
      // This is still the version from electrophotography (toner and
      // carrier particles. But since the generating data type ParEnsPrp
      // is now universal (since derived from Recordable) it will be easy
```

```
      // to implement constructors for completely different ensembles,
      // such as a pile of sand.
      // An aggregate of at least one carrier particle and several toner
      // particles near to it such that the net charge is zero. The
      // constructors allow to set the constituents according to toner
      // and carrier data in a format which is common for such data. The
      // grain is electrically neutral as a whole and the charges are
      // adjusted to a charge/mass ratio of the toner material.
      static V<ParT> makeParticles(Z np, const ParEnsPrp& gp);
      static V<ParT> fillBox(const Box& box, ParEnsPrp gp,
         R facVol=1, Z np=0);

      V<ParT> rep;

   public:
      typedef ParEns Type;
      ParEns(ParEnsPrp const& pep, Z np):rep(makeParticles(np,pep)){}
         // This creates the particles near the origin and assumes that
         // proper positioning in space will be carried out by other
         // instances.

       ParEns(ParEnsPrp const& pep, Box const& box, R facVol=1, Z np=0):
         rep(fillBox(box,pep,facVol,np)){}
         // Creates a list of particles which are 'according to
         // specifications pep' and are placed randomly inside the box
         // (without provisions against overlap, since now a modification
         // of the dynamical law unbuilds overlap effectively). Adding
         // up the volumes of the particles gives facVol times the volume
         // of the box. Since making particles is expensive due to the
         // stochastic integration involved, it is possible by giving to
         // np a value > 0 to make only np particles and use them
         // repeatedly.

       CPM_IO_V

       V<ParT> operator()(void)const{ return rep;}

   }; // class ParEns

template <class ParT>
V<ParT> ParEns<ParT>::makeParticles(Z np, ParEnsPrp const& gp)
{
   Z mL=1;
   static Z npSmall=16;
   static Word loc("ParEns<>::makeParticles(Z,ParEnsPrp)");
   CPM_MA
   cpmmessage(mL,"np="&cpmwrite(np));
   Word sec("grain properties");
   R r0T=0.5*gp.getR(sec,"dT");
   R r0C=0.5*gp.getR(sec,"dC");
```

```
R sT=gp.getR(sec,"sT");
R sC=gp.getR(sec,"sC");
R TC=gp.getR(sec,"TC");
R qmT=gp.getR(sec,"qmT");
Word typeT=gp.getW(sec,"typeT");
Word typeC=gp.getW(sec,"typeC");
sec="tonerproperties";
R rhoT=gp.getR(sec,"rho");
sec="carrierproperties";
R rhoC=gp.getR(sec,"rho");

NuggetData nugDatT(gp.rc(),"nuggetdata_for_toner");
NuggetData nugDatC(gp.rc(),"nuggetdata_for_carrier");

DumbbellData dumDatT(gp.rc(),"dumbbelldata_for_toner");
DumbbellData dumDatC(gp.rc(),"dumbbelldata_for_carrier");

PairData pairDatT(gp.rc(),"pairdata_for_toner");
PairData pairDatC(gp.rc(),"pairdata_for_carrier");

V<R> radii(np);
V<R> radii2(np);
V<R> masses(np);
V<bool> toner(np);

R r,q,m;
Z i;
R tonerContent=0;
R tonerMass=0;
R carrierMass=0;
R totalMass=0;
R spc=0;
for (i=1;i<=np;i++){
   bool ton=(tonerContent<TC);
      // for TC>0, the first particle will be a toner particle
      // for TC==0, we will never get a toner particle
   if (ton){
      r=r0T;
      r*=exp(CpmAlgorithms::gaussRandomValue(0,sT));
      m=CpmDim3::Spc::vol(r)*rhoT; // particles are 3-dimensional !
      tonerMass+=m;
   }
   else{
      r=r0C;
      r*=exp(CpmAlgorithms::gaussRandomValue(0,sC));
      m=CpmDim3::Spc::vol(r)*rhoC;
      carrierMass+=m;
   }
   totalMass+=m;
   if (totalMass<=0)
```

```
      cpmerror("ParEns::makeParticles(Z,...): totalMass<=0");
   tonerContent=tonerMass/totalMass;
   radii[i]=r;
   radii2[i]=r*r;
   masses[i]=m;
   toner[i]=ton;
}
R toner_r2=0;
R carrier_r2=0; // the sums over radius square will help to asign
   // charges in proportion to the surfaces
for (i=1;i<=np;i++){
   if (toner[i]) toner_r2+=radii2[i]; else carrier_r2+=radii2[i];
}
R tonerCharge=tonerMass*qmT;
R carrierCharge=-tonerCharge;
R tonerFactor=0,carrierFactor=0;
if (toner_r2>0) tonerFactor=tonerCharge/toner_r2;
if (carrier_r2>0) carrierFactor=carrierCharge/carrier_r2;

V<ParT> res(np);
R npInv=1./np;
R qTot=0;
for (i=1;i<=np;i++){
   cpmprogress("Particle Factory",i*npInv);
   bool ton=toner[i];
   r=radii[i];
   q = ton ? radii2[i]*tonerFactor : radii2[i]*carrierFactor;
   R d=2*r;
   R surface=CpmDim3::Spc::sur(r);
   ParticleProperties pp;
   sec= ton ? "tonerproperties" : "carrierproperties";
   pp.sigma=gp.getR(sec,"sigma");
   pp.J=gp.getR(sec,"J");
   pp.rho=gp.getR(sec,"rho");
   pp.E_=gp.getR(sec,"effectiveE");
   pp.alphaAdhesion_=gp.getR(sec,"alphaAdhesion");
   pp.relativeRangeAdhesion_=gp.getR(sec,"relativeRangeAdhesion");

   if (np>1) pp.sigma=q/surface;
      // notice that pp.sigma is already initialized in gp.
      // If there is charge to mass,...  given also
      // it has to be adjusted to these data
   if (ton){
      if (typeT=="pair"){
         res[i]=ParT(d,pp,pairDatT);
      }
      else if (typeT=="dumbbell"){
         res[i]=ParT(d,pp,dumDatT);
      }
      else{
```

```
            res[i]=ParT(d,pp,nugDatT);
         }
      }
      else{
         if (typeC=="pair"){
            res[i]=ParT(d,pp,pairDatC);
         }
         else if (typeC=="dumbbell"){
            res[i]=ParT(d,pp,dumDatC);
         }
         else{
            res[i]=ParT(d,pp,nugDatC);
         }
      }
   }
   for (i=1;i<=np;i++){
      res[i].set_id(i); // added 2004-11-01
      qTot+=res[i].get_q();
   }
   if (np>npSmall){
      R atc=cpmabs(tonerCharge);
      if (atc>0){
         R vQuality=cpmabs(qTot)/atc;
         cpmmessage(mL,"vQuality="&cpmwrite(vQuality));
         cpmassert(vQuality<1e-4,loc);
      }
   }
   CPM_MZ
   return res;
}

template <class ParT>
V<ParT> ParEns<ParT>::fillBox(const Box& box, ParEnsPrp gp,
   R facVol, Z np)
{
   Word sec("grain properties");
   R gpdC=gp.getR(sec,"dC");
   R gpdT=gp.getR(sec,"dT");
   R rPart=0.5*(gpdC+gpdT);
   R volPart=Spc::vol(rPart);
   cpmassert(volPart>0,"V<ParT> ParEns::fillBox(...)");
   R volAim=box.vol()*facVol;
   Z n=Z(volAim/volPart);
   if (n<1) n=1;
   Z nAct;
   if (np<=0) nAct=n;
   else nAct=CpmRootX::inf<Z>(np,n);
   V<ParT> vc=ParEns::makeParticles(nAct,gp);
   V<Spc> vs=box.ran(n);
   V<ParT> vcRes(n);
```

```
   Spc ori;
   Z iAct=0;
   for (Z i=1;i<=n;i++){
      Vec tr=vs[i]-ori;
      if (iAct<nAct) iAct++;
      else iAct=1;
      ParT pi=vc[iAct];
      pi*=Group(tr);
      vcRes[i]=pi;
   }
   return vcRes;
}

template <class ParT>
bool ParEns<ParT>::prnOn(ostream& str)const
{
   cpmwt("ParEns<...> begin");
   cpmp(rep);
   cpmwt("ParEns<...> end");
   return true;
}

template <class ParT>
bool ParEns<ParT>::scanFrom(istream& str)
{
   cpms(rep);
   return true;
}
```

137 cpmparofspc.h

```
//? cpmparofspc.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_PAROFSPC_H_
#define CPM_PAROFSPC_H_
/*
    Description: Describing a partition of space into
    subsets in a way to help parallel computing, where
    a process is linked to one such subset.
*/
#include <cpmdim.h>
#include <cpmshapes.h>
#include <cpmangle.h>
#include <cpmlinalg.h>
#include <cpmfr.h>
#include <cpmrecord.h>
#include <cpmframe.h>
#include <cpmgroupgraph.h>
#include <cpmdimdef.h>
#include <cpmalgorithms.h>
#include <cpmmini.h>
#include <cpmvectormore.h>
#include <cpmforces.h>
#include <cpmimagingtools.h>

#if defined(CPM_DIMDEF2)
#define CPM_PHYSICS CpmPhysics2
#define CPM_CAMERA CpmCamera2
#define CPM_DIM CpmDim2

namespace CPM_DIM{
   #include <cpmparofspcx.h>
}

#undef CPM_PHYSICS
#undef CPM_CAMERA
#undef CPM_DIM

#endif

//////////////////////////////////////////////////////////////////////////

#if defined(CPM_DIMDEF3)
#define CPM_PHYSICS CpmPhysics3
#define CPM_CAMERA CpmCamera3
#define CPM_DIM CpmDim3
```

```
namespace CPM_DIM{
   #include <cpmparofspcx.h>
}

#undef CPM_PHYSICS
#undef CPM_CAMERA
#undef CPM_DIM

#endif

#endif // guard
```

138 cpmparofspcx.h

```
//? cpmparofspcx.h
//? Status of work 2008-10-25.
//?
/*
   Description: See cpmparofspc.h
*/

   using namespace CpmSystem;
   using namespace CpmStd;
   using CpmRoot::C;
   using CpmRoot::R;

   using CpmRootX::R1;
   using CpmRootX::Z1;
   using CpmRootX::B;
   using CpmRootX::Record;
   using CpmRootX::RecordHandler;

   using CpmLinAlg::Z2;
   using CpmLinAlg::Z3;

   using CpmArrays::V;
   using CpmArrays::Vo;
   using CpmArrays::X2;
   using CpmArrays::X3;
   using CpmArrays::R_Vector;

   using CpmFunctions::R_Func;
   using CpmFunctions::R2_Func;
   using CpmFunctions::F;
   using CpmFunctions::F5;

   using CpmAlgorithms::CyclicAlarm;
   using CpmAlgorithms::Mini;

   using CpmGraphics::Frame;
   using CpmGraphics::Graph;

   using CpmImaging::Color;
   using CpmImaging::Response;
   using CpmImaging::WHITE;

   using CpmGeo::Angle;
   using CpmGeo::Iv;
   using CpmGeo::GroupGraph;

   using CPM_PHYSICS::ScalarField;
```

```
   using CPM_CAMERA::Camera;

// belongs the namespace for which Spc,Vec,SubsetOfSpc are free
// for usage without qualification
   //doc namespace CpmDimx{
////////////////////// class ParOfSpc<> ////////////////////////////
   template <class Dat>
   class ParOfSpc{
      // partition of space for parallelization of particle dynamics
      // A construct for dividing a region in space (2D or 3D
      // according to the dimension set in cpmdimdef.h) in
      // regions in such a manner that particles living in
      // the same region can efficiently be handled by a single
      // processor in a parallel computing application.
      // The underlying abstraction is the following:
      // Let 1,...n enumerate the processes and let X be
      // the part of space in which 'the whole system lives'.
      // Let X(1),...X(n) be a partition of X. (i.e. the
      // X(i) are mutually disjoint and their union is X).
      // For each X(i), we consider a larger region Y(i) such
      // that the boundary of Y(i) has positive distance from
      // the boundary of X(i). A particle p with position x \in X(i)
      // is said to be owned by process i and used by all processes j
      // for which x \in Y(j). ('\in' is a notation borrowed from LaTeX
      // and denotes the 'is element of ...' relation symbol of set
      // theory). Or put differently: Process i owns all
      // particles with position \in X(i) and uses all particles with
      // position \in Y(i)\X(i). The idea behind this is: giving
      // a process for some small span of time control over all
      // the particles he owns and all the particles he uses
      // will allow it to take into account all the interactions that
      // control the fate of the owned particles. After
      // the time span has elapsed, the particles have to become
      // redistributed over the processes since they might have crossed
      // boundaries of the X(i) and Y(i), and thus have to be associated
      // with new owners and users for the next autonomous time step.
      // To provide the information on which this redistribution can
      // be efficiently based is the main functionality of this class.
      // For systems with long-ranging forces, like charged and
      // magnetized particles, it is not sufficient to take the particles
      // in Y(i) into account in addition to the 'native' ones of X(i).
      // Actually also the most distant particles must not be forgotten
      // since they may in their totality add significantly to,say,
      // the electric field in area X(i). The important point is,
      // however, that these distant particles need not enter the model as
      // individual particles but as more economic integrated quantities
      // with drastically less degrees of freedom and a lower rate of
      // change so that they can be considered constant during the time
      // span of the previous consideration.
      // The data nS1, nS2, nS3 of the present class will be used to
```

```
   // implement this "far field description" when constructing
   // class Station. For this construction the present template class
   // will be used with template argument class StationDat
   // (see cpmstationdatx.h) which specializes X to the region between
   // a plane, a cylinder, and two planes containing the cylinder
   // axis.

typedef X2<Z1,Group> ZG;

Dat dat;
   // data that determine the part of space to be partitioned
Z1 nL1;
   // number of subfields in 1-direction (x-direction)
Z1 nL2;
   // number of subfields in 2-direction (y-direction)
Z1 nL3;
   // number of subfields in 3-direction (z-direction)
R1 overlap1;
   // fractional overlap in 1-direction
R1 overlap2;
   // fractional overlap in 2-direction
R1 overlap3;
   // fractional overlap in 3-direction

R1 ov1;
   // overlap1/nL1, is absolute overlap between adjacent regions
   // Y(i) and X(++i) if we denote by ++i the index of a neighbor
   // process to process i. Here the overlap is given in the
   // normalized parameter space to which dat.map() maps.
R1 ov2;
   // overlap2/nL2
R1 ov3;
   // overlap3/nL3

Z1 nS1;
   // a number of points in 1-direction needed for course graining
   // the sources of the far field
Z1 nS2;
   // a number of points in 2-direction needed for course graining
   // the sources of the far field
Z1 nS3;
   // a number of points in 3-direction needed for course graining
   // the sources of the far field

B isTorus1;
   // True if toroidal topology in 1-direction should hold.
   // Toroidal topology is the present implementation of
   // 'periodic boundary conditions' as proposed by Eric Stelter.
B isTorus2;
   // true if toroidal topology in 2-direction should hold
```

```
B isTorus3;
   // true if toroidal topology in 3-direction should hold

Group g1;
   // Transformation that change a particle state near one
   // boundary of the box into the state which influences
   // the particles near the opposit boundary of the box.
   // Schematics of the situation:
   //
   // (......Y1.......)  (...... Y3......)  areas Y of usage
   // ..Y3..)  (.......Y2.......)  (..Y1..     with toroidal topology
   //  .s[.p)..(..][..)..(..][..)..(.r].q   overlap and
   //                                          particles p,q,r,s
   //  ..[...X1...][...X2...][...X3...]..   areas X of ownership
   //
   // Since p belongs to X1 and lives near the boundary to
   // the toroidal neighbor region X3, thus in the wrapped-arround
   // piece of Y3, p has to be messaged to process 3 as a transformed
   // object q=p*g1 which lives in the non-wrapped-arround part of
   // Y3\X3, where it is close to the particles owned by process 3.

   // In an completely analog manner:
   // Since r belongs to X3 and lives near the boundary to
   // the toroidal neighbor region X1,
   // r has to be messaged to process 1 as a transformed object
   // s=r*!g1 which lives in Y1\X1, the area used but not owned by
   // process 1. Here, !g1 is the group inverse of g1. This implies
   // in the 'graphics' above that the vector from p to q equals the
   // vector from s to r. Notice that r and s both have distance 1
   // from their respective boundary, wheras p and q have distance of
   // 2 units. I'm surprised that the schematic representation above
   // looks clearer to me than my pencil drawings!

Group g2;
   // only for uniformity. In the PaDyAn program it makes no sense
   // to consider the PC and the shell toroidal neighbors
Group g3;
   // toroidal geometry in 3-direction is even more natural than
   // in 1-direction since it reduces the surface effects from
   // missing neighbors (by providing neighbors)

Z nTot;
   // total number of space areas that make up the partition
   // nL1*nL2 for 2D and nL1*nL2*nL3 for 3D

V<Iv> xv1;
   // array of nL1 non-overlapping intervals for angle after map
V<Iv> xv2;
   // array of nL2 non-overlapping intervals for radius after map
V<Iv> xv3;
```

```
   // array of nL3 non-overlapping intervals for z after map

V<SubsetOfSpace> sub;
   // The pieces X(i) of space for which the various processes are
   // responsible

SubsetOfSpace subX;
   // The piece X of space under consideration (set union of the
   // X(i), but for reasons of efficiency not implemented as such

VV<Spc> points;
   // points[i] is
   // a set of nS1*nS2(*nS3) internal points of X(i) useful
   // for approximating fields and field sources. The primary
   // definition of these points is by giving values to their
   // images under map, so that mapInv has to be used to get
   // these points in Spc.

V<R> resolutions;
   // resolutions[i]=Spc::radFrmVol(volumes[i]/(nS1*nS2(*nS3)))
   // i.e. the radius of a sphere that has volume
   // volumes[i]/(nS1*nS2(*nS3))

V<R> coveringRadii;
   // r:=coveringRadii[i] is such that for each x \in X(i)
   // there is a p \in points[i] for which |x-p|<=r holds.
   // coveringRadii[i] and resolutions[i] may differ by
   // a factor of 2 but not much more.

R smallLength_;
   // a small fraction of the smallest of the resolutions

V<R> volumes;
   // volumes[i] is the volume of the X(i) \subset X.
   // The computation assumes a geometry close to
   // a parallelepiped for X(i)

Box box;
   // minimum enclosing box

V<Spc> corners;
   // corners of X in space

Spc center;
   // center of X in space

void iniPoints();
void iniResolutions();
void iniVolumes();
void iniSub();
```

```
   V<Spc> makePoints(Z)const;

   V<ZG> toSend1(Z i, R r)const;
      // as pointed out in the explanation to g1, the messaging
      // between toroidal neighbors involves transformations which
      // get consistently set by calling this function. Although
      // the Group element is the unit element in most cases and
      // then has no effect at all it is not advisable to make use
      // of this and to treat the transformations as an ad hoc or
      // exceptional addition to non-toroidal code. The present
      // implementation of Group and the pairing templates is
      // sufficiently efficient that the computational burden from
      // these additional data is far from being significant.
      // The function will be called for r=map(x)[1]
      // where x is the position of a particle.
      // There are two possibilities:
      // (i) the particle position x is in 1-direction (we are
      // considering function toSend1) 'interior' i.e. not near to a
      // boundary. In this we return V<ZG>(0).
      // (ii) the particle position x is not in the interior
      // so that it has to made known to other parts of the system
      // (which in most cases will be handled by a different process).
      // This transition is described by res[1] which is an
      // instance of X2oio(Z1,Group).

   V<ZG> toSend2(Z i, R r)const;
      // see toSend1
   V<ZG> toSend3(Z i, R r)const;
      // see toSend1
// private methods
   void adjust();
      // sets the values of the dependent quantities3
public:
   typedef ParOfSpc Type;
   CPM_IO
   ParOfSpc(const Dat& dat, Z nL1, Z nL2, Z nL3=1,
      R ov1=0, R ov2=0, R ov3=0, Z nS1=2, Z nS2=2, Z nS3=1, R eps=1e-2);
      // constructor from basic input
   ParOfSpc(){}
      // trivial, everything zero

   const Dat& getDat()const{ return dat;}
   SubsetOfSpace getX()const{ return subX;}
   SubsetOfSpace getX(Z i)const{ return sub[i];}
   V<Spc> getPoints(Z i)const{ return points[i];}
      // a set of nS1*nS2(*nS3) internal points of X(i) useful
      // for approximating fields and field sources. The primary
      // definition of these points is by giving values to their
      // images under map, so that mapInv has to be used to get
      // these points in Spc.
```

```
R getRes(Z i)const{ return resolutions[i];}
R getCovRad(Z i)const{ return coveringRadii[i];}
R getVol(Z i)const{ return volumes[i];}
   // returns the volume (area in 2D) of X(i) (in real space, not
   // after
   // mapping
   // into the unit cube/square)
R get_xL()const{ return dat.get_xL();}
R get_xU()const{ return dat.get_xU();}
R get_xLength()const{ return dat.get_xLength();}
R get_xLength2()const{ return dat.get_xLength2();}
R get_yLength()const{ return dat.get_yLength();}
R get_zLength()const{ return dat.get_zLength();}
V<Spc> testPoints(Z n)const;
   // test points along the PC for testing the values of
   // the electrical potential

bool torus1()const{ return isTorus1;}
bool torus2()const{ return isTorus2;}
bool torus3()const{ return isTorus3;}
bool torus()const{ return torus1()||torus2()||torus3();}

void setTorus(Z i, const Group& g);
   // In order not to overload the argument list of the constructor,
   // we set toroidal topology (if needed) by the present
   // function. It goes without saying that all dependent
   // quantities get adjusted accordingly.

Z card()const{ return nTot;}
   // number of subsets

R resolution(Z rank)const;
   // returns the smallest distance between the coarse graining
   // points in subset rank

R smallLength()const{ return smallLength_;}

Rdim map(const Spc& z)const;
   // For a point z in the region RegPar which is to be partitioned
   // the return value belongs to the square [0,1]x[0,1] or the cube
   // [0,1]x[0,1]x[0,1] of Rdim. So, a partition of the region can
   // be achieved by partitioning this square (or cube). This is
   // done in function index().
   // All points outside RegPar get mapped to points outside
   // the unit square or unit cube.
   // map is a bijective mapping Spc(Rdim\X) --> Rdim\X, where
   // X:={ x \in Rdim | x[1]==0 and x[2]<=-r}

Spc mapInv(const Rdim& z)const;
   // inverse of map: Rdim/X --> Spc(Rdim\X)
```

```
F<Spc,Rdim> toMap()const;
   // map as autonomous function object. Notice the following:
   // Let par be any instance of the present class. Then
   // f=par.toMap is a function that describes the part of space
   // covered by par. If we change par by e.g. par*=g, (not
   // implemented yet but clear what this would do) then f would still
   // describe the unchanged position (which is perfectly OK!).
   // Function is used to initialize data 'sub'.

F<Rdim,Spc> toMapInv()const;
   // analogous to the last function

Z1 linearCoding(const Zdim& z)const;
   // linear index made out of a Z2-or Z3-index.

Zdim linearCodingInv(Z z)const;
   // inverse of function linearCoding()

V<X2<Z1,Group> > toSend(Z rank, const Spc& pix)const;
   // This is the function providing the main functionality of the
   // class. It will only be called (only then the behavior is as
   // described) from process rank for values of pix for which
   // owned(rank,pix)==true. Let then res be the return value of
   // the function. Then rd:=res.dim() is the number of processes to
   // which process rank has to send a particle p with p.get_x()==pix.
   // ( since these processes have to use particle p, which is owned
   // by process rank). For i=1,...rd the receiving processes
   // have ranks res[i].c1() and need particle p transformed by
   // Group element res[i].c2(); see explantion to g1 for the meaning
   // these transformations.

bool owned(Z rank, const Spc& pix)const;
   // returns true if pix belongs to X(rank) and false else

R dist(Z i, const Spc& pix)const{ return sub[i](pix);}
   // the more negative this quantity the 'deeper inside X(i)'
   // ix point pix. For value 0, pix is on the boundary
   // and for positve values outside. Important tool for
   // achieving 'unique ownership' of particles in parallel
   // computing.

X3<Z,Group,B> owner(const Spc& pix)const;
   // Returns (i,Group(),false) where i is such that pix \in X(i).
   // If not all attributes isTorus... are false, there
   // is the possibilty that there is no i for which
   // pix \in X(i) is valid. Then we return (j,g,b) where
   // pix*=g \in X(j) and g is of the form g1^e1*g1^e2*g3^e3
   // where ei=0 for !isTorusi and \in {1,-1} else.
   // b=false is normal, b=true can only happen in 3D and then
```

```
   // indicates that pix is off of any X(i) in lateral
   // direction. Knowing this allows ParDynSys::redistribute(..)
   // to reduce the velocity of the particle in order to
   // damp lateral motion.

Z ownerSimple(const Spc& pix)const;

Z repNegDim()const;
   // dimension of the array to created by the next function

V<Spc> repNeg(const Spc& x)const;
   // repetition neighbors. Assume that the total space X is only a
   // small part of the real system space.
   // Then one may want to approximate the huge system by
   // considering X in detail and assume that the system outside
   // behaves just like the system in X, so that we can think of X
   // as surrounded by copies (repetitions) of X. These copies of
   // X are not assumed to be detailed copies but are only assumed
   // to agree with X with respect to coarse grained descriptors
   // e.g. of electromagnetic fields. Let s be a field source
   // located at x in X. Assuming the kind of periodic immersion
   // of X in the larger system, we have to assume that the same
   // source is located also in all points of a lattice generated
   // from x with translation vectors or - more generally - by
   // transformations as our g1,g2,g3. A reasonable approximation,
   // which will bring about the essential influence from the
   // neigborhood will be to consider only the directly adjacent
   // copies of X (presently we also take into account the diagonally
   // adjacent ones, which have only edges or points in common with
   // X).
   // The result of shifting the source can be immitated by
   // considering only one source, evaluating the field not simply in
   // a point y, but in all points which are components of repNeg(y)
   // and add the results.
   // This involves some approximation in systems
   // like the toning nip when we consider X over which the nip
   // height cannot be considered constant. It is, however, the
   // intention of this periodic immersion to avoid the need for
   // considering such large pieces of the system.
   // Short explanation: repNeg(x) is the list of points which are
   // equivalent to x with respect to the kind of periodicity
   // under consideration. See function
   // CPM_TAL
   // void CPM_FCN::prepareStep(...) in cpmpardynsys.h for the only
   // utilitisation of the concept made so far.
   // Since meanwhile I came to distrust this logic (erroneously
   // as it appears to me now) lets make it crystal clear. Formulas
   // have to be understood according to LATEX formatting:
   // Let V be the superposition of shifted terms
   // V(x):=\sum_{i=-n}^{n} V_{i}(x)
```

```
      // V_{i}(x):=V_{0}(x-i*s)
      // Thus
      // V(x)=\sum_{i=-n}^{n} V_{0}(x-i*s)
      // Thus we have in fact to sum up the values that a single
      // function V_{0} gives for a chain of points
      // \{ p=x-i*s | i=-n,...n \}.

   C toColor(const V<Z>& vz)const;
      // a peliminary color coding complex number intended to let the
      // mapping C-->C, z|--> part.toColor(neighbors(z)) be a function
      // which visualizes the Partition part as a pattern in space C.
      // when inserted as f in function
      // void CpmGraphics::Graph::show(const F<C,C>& f, R gamma=0.01);

   void mark(const Camera& gr)const;

   void show(const Frame& fr, bool undistorded=true)const;
      // displays the geometry of the partition on the graphics
      // window fr. If the parameter undistorted is set to 'true'
      // the width of the displayed rectangle is set such that
      // angles are represented correctly (no distortion),
      // otherwise the width of the partition as determined by
      // r, phiL, and phiU just fits the width of the graphics
      // window. The height of the displayed rectangle is
      // determined in relation to h by the parameter
      // const R hightExcess in the implementation of the function.

   void showScalarField(const Camera& ca, const ScalarField& f,
      Z nx, Z ny, const Response& rp)const;
      // not yet in use

   Record toRecord()const;
      // Record is a data type which allows to hold data of types
      // bool,Z,R, and Word in a convenient manner (if one has to add
      // a new data element of any of those types one does not have
      // to reorganize the yet existing data in any way. So putting
      // into a record is a RAM-based analogy to writing to external
      // storage via prnOn(). This will help ma lot in defining the
      // constructor of class Station more densly than before.
      // A present drawback compared to stream writing is that vectors
      // (arrays) can't (not yet) be transfered by a single statement.

//   GroupGraph<Group> groupgraph(const V<Spc>& vs )const;
      // see my experimental file cpmparallel.h for usage
      // and the simple implementation code as explanation.
      // Not yet proven by experience

   V< X2<Spc,Z> > epLatticePrel(const V<X2<R,bool> >& vx,
      Rdim xL, Rdim xU, R& facCell)const;
      // ep stands for electro photography
```

```
      // vx[i].c1() is a radius of a bounding sphere of a particle,
      // and vx[i].c2() is true if this particle is a carrier particle.
      // Let res be the return value of the function. For each
      // j<=res.dim(), res[j].c1() is a point in space, where one has
      // to place a particle with radius and particle type given by
      // vx[res[j].c2()]. The 'particle'-ensemble defined this way
      // fills, after having transformed positions via map into
      // the unit cube (square for dimVec=2), the subcube
      // xL.x1<=x[1]<=xU.x1, xL.x2<=x[2]<=xU.x2, ( xL.x3<=x[3]<=xU.x3 ).
      // The last factor controlls the density of the lattice points
      // so that for fast testing also 'dillute' systems can be created
      // (facCell >> 1 very dillute system). In the present version
      // res.dim is not an integer multiple of vx and thus it is not
      // guaranteed that each input 'particle' is used a fixed number of
      // times. As a result it is not guaranteed that the resulting
      // particles have net charge zero even when vx came from particles
      // of net charge zero. I observed in most cases that |net charge|
      // was arround 10% of the total positive charge.
      // This is more than I expected. Since it is difficult to build
      // charge neutrality reliably into the present code (without
      // leaving loosely bound particles at the end (which would probably
      // move away) I will use an explicit neutralization procedure
      // afterwards.
      // In the present context, where the particles are present only
      // in a very abstracted form, this can't be done.
      // For vx.dim()=1 we define res=V< X2<Spc,Z> >(1) and
      // res[1]=X2<Spc,Z>(xc,1), where xc is the mid-point of the cell
      // defined by xL and xU. This is helpful since it allows to
      // have a single, well defined particle as a special case of
      // a developer lump.
      // For vx.dim()=2 the lattice points are just those corresponding
      // to xL and xU.
      // For dimVec==3 and narrow space in 3-direction
      // (xU[3]-xL[3])<<(xU[1]-xL[1]) the relative arrangement of
      // toner and carrier is not yet ideal. For densely packed
      // systems as those we are interested in for PaDyAn, this is no
      // concern, however. Severe malfunction now results in
      // program stop, less severe ones create a warning.

V< X2<Spc,Z> > epLattice(const V<X2<R,bool> >& vx,
      Rdim xL, Rdim xU, R facCel, Z nPar=0)const;
      // Based on previous function. For nPar<=0 it is just the previous
      // function. For nPar>=1 the value of facCel is taken as initial
      // value for a simple and efficient strategy to let res.dim()
      // approximate nPar by changing facCel from trial to trial by a
      // fixed factor or divisor of 1.1. Even for 10000 particles and very
      // bad initial value for facCell, the execution time is no concern.

   void autoFrame1(Camera& ca, R autoFrameFactor=1.1);
      // sets the frame of cam such that *this fills the frame
```

```
      // with a 'security' determined by the second argument

   Z dim()const{ return Spc::dimension();}
};

namespace{

template <class Dat>
Rdim fmap(const Spc& x, const ParOfSpc<Dat>& ps){ return ps.map(x);}

template <class Dat>
Spc fmapInv(const Rdim& x, const ParOfSpc<Dat>& ps)
   { return ps.mapInv(x);}

}

template <class Dat>
F<Spc,Rdim> ParOfSpc<Dat>::toMap(void)const
{
   return F1<Spc,ParOfSpc<Dat>,Rdim>(*this)(fmap);
}

template <class Dat>
F<Rdim,Spc> ParOfSpc<Dat>::toMapInv(void)const
{
   return F1<Rdim,ParOfSpc<Dat>,Spc>(*this)(fmapInv);
}

template <class Dat>
void ParOfSpc<Dat>::setTorus(Z i, const Group& g)
{
   if (i==1){
      isTorus1=true;
      g1=g;
   }
   else if (i==2){
      isTorus2=true;
      g2=g;
   }
   else if (i==3){
      isTorus3=true;
      g3=g;
   }
   else{
      cpmerror("ParOfSpc<Dat>::setTorus(Z i,...) i not in {1,2,3}");
   }
   adjust();
}

template <class Dat>
```

```
void ParOfSpc<Dat>::adjust()
{
   const Z mL=3;
   cpmmessage(mL,"void ParOfSpc<Dat>::adjust() started");
   if (dimVec==2) nL3=1;
   nTot=nL1*nL2*nL3;
   ov1=overlap1/nL1;
   ov2=overlap2/nL2;
   ov3=overlap3/nL3;
   xv1=Iv(0,1).divide(nL1);
   xv2=Iv(0,1).divide(nL2);
   if (dimVec==3){
      xv3=Iv(0,1).divide(nL3);
   }
   iniPoints();
   iniVolumes();
   iniResolutions(); // needs volumes
   iniSub(); // needs resolutions
   cpmmessage(mL,
      "void ParOfSpc<Dat>::adjust() done, nTot="&cpmwrite(nTot));
}

template <class Dat>
ParOfSpc<Dat>::ParOfSpc(const Dat& dat_, Z n1, Z n2, Z n3,
   R ov1, R ov2, R ov3, Z nS1_, Z nS2_, Z nS3_, R eps_):
   dat(dat_),nL1(n1),nL2(n2),nL3(n3),
   overlap1(ov1), overlap2(ov2), overlap3(ov3),
   nS1(nS1_), nS2(nS2_), nS3(nS3_)
{
   eps_; // not used
   Z mL=2;
   Z mL2=10;
   Word loc("ParOfSpc<Dat>::ParOfSpc(...)");
   CPM_MA
   adjust(); // main work
   if (cpmverbose>=mL2){ // a visual test structure
      // to see whether the sub[i] where created
      // correctly
      V<Word> inifiles("","station.ini");
      RecordHandler rc(inifiles);
      Frame fr;
      Word sec="cameracontrol";
      Camera ca(rc,sec,fr);
      for (Z i=1;i<=nTot;i++){
         SubsetOfSpace sos=sub[i];
         if (cpmverbose>mL2){
           // sos.set_dr(5e-6);
            sos.set_r(2000e-6);
          //  sos.set_ds(10e-6); no longer needed
            sos.set_x(Rdim());
```

```
            ca.show(sos,Texture(40e-6));
         }
         else{
            ScalarField sf(sos);
            sf.mark(ca);
         }
         cpmwait(10);
      }
   }
   CPM_MZ
}

template <class Dat>
Rdim ParOfSpc<Dat>::map(const Spc& z)const
{
   return dat.f(z.toRdim());
}

template <class Dat>
Spc ParOfSpc<Dat>::mapInv(const Rdim& z)const
{
   return Spc(dat.fInv(z));
}

template <class Dat>
bool ParOfSpc<Dat>::owned(Z rank, const Spc& pix)const
{
   X3<Z,Group,B> zgb=owner(pix);
   return zgb.c1()==rank;
}

template <class Dat>
Z ParOfSpc<Dat>::ownerSimple(const Spc& pix)const
{
   Z mL=3;
   Word loc("ParOfSpc<Dat>::ownerSimple(...)");
   cpmmessage(mL,loc&" started");
   Rdim mz=map(pix);
   Zdim res0;
   Z i1=1,i2=1,i3=1;
   R p;
   p=mz[1];
   if (p<=0) i1=1;
   else if (p>=1) i1=nL1;
   else
      while (!xv1[i1].contains(p,false,true)) i1++;
   res0[1]=i1;

   p=mz[2];
   if (p<=0) i2=1;
```

```
   else if (p>=1) i2=nL2;
   else
      while (!xv2[i2].contains(p,false,true)) i2++;
   res0[2]=i2;

   if (dimVec==3){
      p=mz[3]; // was missing till 2003-08-21
      if (p<=0) i3=1;
      else if (p>=1) i3=nL3;
      else
         while (!xv3[i3].contains(p,false,true)) i3++;
      res0[3]=i3;
   }
   Z res=linearCoding(res0);
   cpmmessage(mL,loc&" done");
   return res;
}

template <class Dat>
X3<Z,Group,B> ParOfSpc<Dat>::owner(const Spc& pix)const
{
   Z mL=3;
   Word loc("ParOfSpc<Dat>::owner(...)");
   cpmmessage(mL,loc&" started");
   Rdim mz=map(pix);
   Z i1=1,i2=1,i3=1;
   R p;
   Group g;
   B slowDown; // false by default

   p=mz[1];
   if (p<0){
      if (isTorus1){
         i1=nL1;
         g*=g1;
      }
      else{
         i1=1;
      }
   }
   else if (p>=1){
      if (isTorus1){
         i1=1;
         g*=!g1;
      }
      else{
         i1=nL1;
      }
   }
   else{
```

```
   while (!xv1[i1].contains(p,false,true))
   { i1++; if (i1>nL1) cpmerror("pBad1="&cpmwrite(p));}
}

p=mz[2];
if (p<0){
   if (isTorus2){
      i2=nL2;
      g*=g2;
   }
   else{
      i2=1;
   }
}
else if (p>=1){
   if (isTorus2){
      i2=1;
      g*=!g2;
   }
   else{
      i2=nL2;
   }
}
else{
   while (!xv2[i2].contains(p,false,true))
   { i2++; if (i2>nL2) cpmerror("pBad2="&cpmwrite(p));}
}

Zdim res0;
res0[1]=i1;
res0[2]=i2;
if (dimVec==3){
   p=mz[3];
   if (p<0){
      if (isTorus3){
         i3=nL3;
         g*=g3;
         slowDown=true;
      }
      else{
         i3=1;
      }
   }
   else if (p>=1){ // was p>1 till 03-09-13, so that p==0
      // had to trigger cpmerror("pBad3="&cpmwrite(p))
      // this is what happened in Cornell in 03-09-12
      if (isTorus3){
         i3=1;
         g*=!g3;
         slowDown=true;
```

```
         }
         else{
            i3=nL3;
         }
      }
      else{
         while (!xv3[i3].contains(p,false,true))
         { i3++; if (i3>nL3) cpmerror("pBad3="&cpmwrite(p));}
      }
      res0[3]=i3;
   }
   Z res=linearCoding(res0);
   cpmmessage(mL,loc&" done");
   return X3<Z,Group,B>(res,g,slowDown);
}

#define zg(z,g) ZG(Z1(z),g)
#define vzg(z,g) V<ZG>(1,ZG(Z1(z),g))

namespace{

// method to avoid code replication in implementing functions
// toSend1, toSend2, toSend3

V<X2<Z1,Group> > toSendx(Z mL, Word loc, Z i, R r,
   Z nLx, Iv xvxi, Group gx, R ovx, bool isTorusx)
{
   typedef X2<Z1,Group> ZG;
      // would not be needed if we would give the present function
      // the status of a private static member function
   Group u; // unity
   if (nLx==1){
      cpmassert(i==1,loc);
      if (!isTorusx) return V<ZG>(0);
      else{ // added 03-06-04, also a single interval
            // should be allowed to act as a torus
         if (r<xvxi.inf()+ovx) return vzg(1,gx);
         if (r>xvxi.sup()-ovx) return vzg(1,!gx);
            // no neighborhood relation
      }
   }
   else if (i==1){ // notice nLx>=2
      if (r>xvxi.sup()-ovx) return vzg(2,u);

      if (isTorusx)
         if (r<xvxi.inf()+ovx) return vzg(nLx,gx);
   }
   else if (i==nLx){
      if (isTorusx)
         if (r>xvxi.sup()-ovx) return vzg(1,!gx);
```

```
      if (r<xvxi.inf()+ovx) return vzg(nLx-1,u);
   }
   else{
      if (r>xvxi.sup()-ovx) return vzg(i+1,u);
      if (r<xvxi.inf()+ovx) return vzg(i-1,u);
   }
   cpmmessage(mL,loc&" done, V<ZG>(0) returned");
   return V<ZG>(0);
}

} // anon namespace

template <class Dat>
V< X2<Z1,Group> > ParOfSpc<Dat>::toSend1(Z i, R r)const
{
   Z mL=3;
   Word loc("ParOfSpc<>::toSend1(...)");
   CPM_MA
   return toSendx(mL,loc,i,r,nL1,xv1[i],g1,ov1,isTorus1);
}

template <class Dat>
V< X2<Z1,Group> > ParOfSpc<Dat>::toSend2(Z i, R r)const
{
   Z mL=3;
   Word loc("ParOfSpc<>::toSend2(...)");
   CPM_MA
   return toSendx(mL,loc,i,r,nL2,xv2[i],g2,ov2,isTorus2);
}

template <class Dat>
V< X2<Z1,Group> > ParOfSpc<Dat>::toSend3(Z i, R r)const
{
   Z mL=3;
   Word loc("ParOfSpc<>::toSend3(...)");
   CPM_MA
   return toSendx(mL,loc,i,r,nL3,xv3[i],g3,ov3,isTorus3);
}

#undef vzg

template <class Dat>
V<X2<Z1,Group> > ParOfSpc<Dat>::toSend(Z rank, const Spc& pix)const
{
   Z mL=3;
   Word loc("ParOfSpc<>::sendTo(...)");
   CPM_MA
   Group u; // unity
   Rdim mpx=map(pix);
```

```
Zdim rk=linearCodingInv(rank);
R mpx1=mpx.x1;
R mpx2=mpx.x2;
V<ZG> zg1=toSend1(rk[1],mpx1);
V<ZG> zg2=toSend2(rk[2],mpx2);
V<ZG> zg3(0);
if (dimVec==3){
   R mpx3=mpx[3];
   zg3=toSend3(rk[3],mpx3);
}

Z d1=zg1.dim()+1;
Z d2=zg2.dim()+1;
Z d3=zg3.dim()+1;

Z d=d1*d2*d3;

if (d==1){
   cpmmessage(mL,loc&" trivially done, nothing to send");
   return V<ZG>(0);
}

V<ZG> s1(d1),s2(d2),s3(d3);

s1[1]=zg(rk[1],u);
s2[1]=zg(rk[2],u);
s3[1]=zg(rk[3],u);

if (d1>1) s1[2]=zg1[1];
if (d2>1) s2[2]=zg2[1];
if (d3>1) s3[2]=zg3[1];

V<ZG> res(d);
Zdim val;
Z i,j,k,l=0;
Group gi,gj,gk,gl;
// till 2003-06-16 a wrong logic in forming gl created
// some neighbors at wrong (too distant) places, so the dynamics of
// some particles may have been spoiled seriously
if (dimVec==3){
    for (i=1;i<=d1;i++){
      val[1]=s1[i].c1();
      gi=s1[i].c2();
      for (j=1;j<=d2;j++){
         val[2]=s2[j].c1();
         gj=s2[j].c2();
         for (k=1;k<=d3;k++){
            val[3]=s3[k].c1();
            gk=s3[k].c2();
            gl=gi*gj*gk;
```

```
                Z il=linearCoding(val);
                if (gl!=u || il!=rank){
                   l++;
                   res[l]=zg(il,gl);
                }
             }
          }
       }
    }
    else{
       for (i=1;i<=d1;i++){
          val[1]=s1[i].c1();
          gi=s1[i].c2();
          for (j=1;j<=d2;j++){
             val[2]=s2[j].c1();
             gj=s2[j].c2();
             Z il=linearCoding(val);
             gl=gi*gj;
             if (gl!=u || il!=rank){
                l++;
                res[l]=zg(il,gl);
             }
          }
       }
    }
    CPM_MZ
    return res.resize(l);
}
#undef zg

template <class Dat>
Z1 ParOfSpc<Dat>::linearCoding(const Zdim& z)const
// i=nL1, j=nL2, k=nL3 should give nL1*nL2*nL3
// i=1, j=1,k=1 should give 1
{
    Z i=z[1]-1;
    Z j=z[2]-1;
    Z res=nL1*j+i;
    if (dimVec==3){
       Z k=z[3]-1;
       res+=nL2*nL1*k;
    }
    Z1 result(res+1);
    if (result>nTot)
       cpmerror("ParOfSpc<Dat>::linearCoding(...): result>nTot");
    return result;
}

template <class Dat>
Zdim ParOfSpc<Dat>::linearCodingInv(Z z)const
```

```
{
   z--;  // now z starts with 0
   Zdim res;
   if (dimVec==2){
      Z i1=z/nL1;
      z-=nL1*i1;
      res[1]=z+1;
      res[2]=i1+1;
   }
   else{ // thus dimVec==3
      Z n1=nL1;
      Z n2=n1*nL2;
      Z i2=z/n2;
      z-=i2*n2;
      Z i1=z/n1;
      z-=i1*n1;
      res[1]=z+1;
      res[2]=i1+1;
      res[3]=i2+1;
   }
   return res;
}

template <class Dat>
bool ParOfSpc<Dat>::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("ParOfSpc<>",str);
   cpmp(dat);
   cpmp(nL1);
   cpmp(nL2);
   cpmp(nL3);
   cpmp(overlap1);
   cpmp(overlap2);
   cpmp(overlap3);
   cpmp(nS1);
   cpmp(nS2);
   cpmp(nS3);
   cpmp(isTorus1);
   cpmp(isTorus2);
   cpmp(isTorus3);
   cpmp(g1);
   cpmp(g2);
   cpmp(g3);
   return true;
}

template <class Dat>
bool ParOfSpc<Dat>::scanFrom(istream& str)
{
   cpms(dat);
```

```
   cpms(nL1);
   cpms(nL2);
   cpms(nL3);
   cpms(overlap1);
   cpms(overlap2);
   cpms(overlap3);
   cpms(nS1);
   cpms(nS2);
   cpms(nS3);
   cpms(isTorus1);
   cpms(isTorus2);
   cpms(isTorus3);
   cpms(g1);
   cpms(g2);
   cpms(g3);
   adjust();
   return true;
}

template <class Dat>
C ParOfSpc<Dat>::toColor(const V<Z>& vz)const
{
   Z i,n=vz.dim();
   R r1=1;
   for (i=1;i<=n;i++) r1+=vz[i]*pow(0.13,i);
   R r2=1.14;
   for (i=1;i<=n;i++) r2+=vz[i]*vz[i]*pow(0.57,i);
   R x1=r1*1000;
   R x2=r2*1370;
   R s1=50*(sin(x1)+1);
   s1=5*(s1-floor(s1)-0.5);
   R s2=50*(sin(x2)+1);
   s2=5*(s2-floor(s2)-0.5);
   return C(s1,s2);
}

namespace{ // tool for ParOfSpc<Dat>::show(...)
   template <class Dat>
   C fMap(const C& z, const ParOfSpc<Dat>& p)
   {
       return z;
//      return p.toColor(p.proLis(z));
   }
}

template <class Dat>
void ParOfSpc<Dat>::mark(const Camera& ca)const
{
   Rdim x1,x2;
   x1[2]=0;
```

```
   x2[2]=1;
   x1[3]=0.5;
   x2[3]=0.5;
   Z i;
   Spc p1,p2;
   R xi;
   Color c(WHITE);
   for (i=1;i<=nL1;i++){
      xi=xv1[i].inf();
      x1[1]=xi;
      x2[1]=xi;
      p1=mapInv(x1);
      p2=mapInv(x2);
      ca.mark(p1,p2,c);
   }
   xi=xv1[nL1].sup();
   x1[1]=xi;
   x2[1]=xi;
   p1=mapInv(x1);
   p2=mapInv(x2);
   ca.mark(p1,p2,c);
}

template <class Dat>
void ParOfSpc<Dat>::show(const Frame& fr, bool undistorded)const
{
   const R facExcess=2;
   const R gamma=0.75;

   using CpmFunctions::F1;
   using CpmGraphics::Graph;
   Rdim sL(0,0,0);
   Rdim sU(1,1,1);
   Spc pL=mapInv(sL);
   Spc pU=mapInv(sU);

   R xL=pL[1]*facExcess;
   R xU=pU[1]*facExcess;
   R yL=pL[2]*facExcess;
   R yU=pU[2]*facExcess;

   Iv ivx(xL,xU);
   Iv ivy(yL,yU);

   Graph gr(fr);
//   gr.setCol(); // setting colors, not adding them
   gr.setXY(ivx,ivy);
   gr.show(F1<C,ParOfSpc,C>(*this)(fMap<Dat>),gamma);
   gr.display();
   cpmwait(10);
```

```
}

template <class Dat>
void ParOfSpc<Dat>::showScalarField(const Camera& ca,
   const ScalarField& f, Z nx, Z ny, const Response& rsp )const
{
   Z mL=3;
   Word loc("ParOfSpc<Dat>::showScalarField(..)");
   cpmmessage(mL,loc&" started");
   Graph gr=ca.getGraph();
   Iv ivx=gr.getIvX();
   Iv ivy=gr.getIvY();
   R unitPlus=1.25;

   Rdim x00(0,0);
   Rdim x01(0,unitPlus);
   Rdim x10(1,0);
   Rdim x11(1,unitPlus);
   Spc s00=mapInv(x00);
   Spc s01=mapInv(x01);
   Spc s10=mapInv(x10);
   Spc s11=mapInv(x11);
   R2 i00=ca.captureMono(s00);
   R2 i01=ca.captureMono(s01);
   R2 i10=ca.captureMono(s10);
   R2 i11=ca.captureMono(s11);

   R xL=cpmsup<R>(ivx.inf(),i00[1],i01[1]);
   R xU=cpminf<R>(ivx.sup(),i10[1],i11[1]);
   R yL=cpmsup<R>(ivy.inf(),i00[2],i10[2]);
   R yU=cpminf<R>(ivy.sup(),i01[2],i11[2]);
   cpmassert(xL<=xU,loc);
   cpmassert(yL<=yU,loc);

   Iv iv_x(xL,xU);
   Iv iv_y(yL,yU);
   V<R> vx=iv_x.centers(nx);
   V<R> vy=iv_y.centers(ny);
   VV<R> val(ny,nx); // indexed like a image matrix
   Z i,j;
  // R2 xij;
   C xij;
   for (i=1;i<=nx;i++){
      xij[1]=vx[i];
      for (j=1;j<=ny;j++){
         xij[2]=vy[j];
         Spc sij=ca.inverse(xij,1.);
         val[ny-j+1][i]=f(sij);
      }
   }
```

```
   R2_Func r0(val,1,iv_x,iv_y);
   gr.show(r0,rsp,iv_x,iv_y);
   cpmmessage(mL,loc&" done");
}

template <class Dat>
V<Spc> ParOfSpc<Dat>::makePoints(Z ip)const
{
   Zdim zd=linearCodingInv(ip);
   Iv iv1,iv2,iv3;
   iv1=xv1[zd[1]];
   iv2=xv2[zd[2]];
   if (dimVec==3) iv3=xv3[zd[3]];
   V<R> vr1,vr2,vr3;
   vr1=iv1.centers(nS1);
   vr2=iv2.centers(nS2);
   if (dimVec==3) vr3=iv3.centers(nS3);
   Z nRes=nS1*nS2;
   if (dimVec==3) nRes*=nS3;
   V<Spc> res(nRes);
   Z l=0,i,j,k;
   Rdim val;
   if (dimVec==3){
      for (i=1;i<=nS1;i++){
         val[1]=vr1[i];
         for (j=1;j<=nS2;j++){
            val[2]=vr2[j];
            for (k=1;k<=nS3;k++){
               val[3]=vr3[k];
               res[++l]=mapInv(val);
            }
         }
      }
   }
   else{
      for (i=1;i<=nS1;i++){
         val[1]=vr1[i];
         for (j=1;j<=nS2;j++){
            val[2]=vr2[j];
            res[++l]=mapInv(val);
         }
      }
   }
   return res;
}

template <class Dat>
void ParOfSpc<Dat>::iniPoints()
{
   Z i,j,k,l;
```

```
   points=VV<Spc>(nTot,0);
   for (l=1;l<=nTot;l++) points[l]=makePoints(l);

   coveringRadii=V<R>(nTot);
   for (l=1;l<=nTot;l++){
      V<Spc> p=points[l];
      Z pd=p.dim();
      Vo<R> mins(pd);
      for (i=1;i<=pd;++i){
         Vo<R> d(pd-1);
         k=0;
         for (j=1;j<=pd;++j){
            if (j==i) continue;
            k++;
            d[k]=(p[i]-p[j]).absVal();
         }
         mins[i]=d.inf();
      }
      coveringRadii[l]=mins.sup();
   }
}

template <class Dat>
void ParOfSpc<Dat>::iniResolutions()
{
   const R facSmall=1e-3;
   Z mL=3;
   static Word loc("ParOfSpc<Dat>::iniResolutions()");
   CPM_MA
   resolutions=V<R>(nTot);
   if (dimVec==2) nS3=1;
   Z zCells=nS1*nS2*nS3;
   cpmassert(zCells>0,loc);
   R zCellsInv=1./zCells;
   for (Z i=1;i<=nTot;i++){
      resolutions[i]=Spc::radFrmVol(volumes[i]*zCellsInv);
   }
   smallLength_=R_Vector(resolutions).inf()*facSmall;

   CPM_MZ
}

template <class Dat>
void ParOfSpc<Dat>::iniVolumes()
{
   static Word loc("ParOfSpc<Dat>::iniVolumes()");
   volumes=V<R>(nTot);
   for (Z i=1;i<=nTot;i++){
      Zdim zd=linearCodingInv(i);
      Iv iv1=xv1[zd[1]];
```

```
Iv iv2=xv2[zd[2]];
if (dimVec==2){

   Rdim z11;
   z11[1]=iv1[1];
   z11[2]=iv2[1];
   Spc x11=mapInv(z11);

   Rdim z12;
   z12[1]=iv1[1];
   z12[2]=iv2[2];
   Spc x12=mapInv(z12);

   Rdim z21;
   z21[1]=iv1[2];
   z21[2]=iv2[1];
   Spc x21=mapInv(z21);

   Vec a=x12-x11;
   Vec b=x21-x11;
   volumes[i]=a.cross(b).absVal();
}
else{
   Iv iv3=xv3[zd[3]];

   Rdim z111;
   z111[1]=iv1[1];
   z111[2]=iv2[1];
   z111[3]=iv3[1];
   Spc x111=mapInv(z111);

   Rdim z121;
   z121[1]=iv1[1];
   z121[2]=iv2[2];
   z121[3]=iv3[1];
   Spc x121=mapInv(z121);

   Rdim z112;
   z112[1]=iv1[1];
   z112[2]=iv2[1];
   z112[3]=iv3[2];
   Spc x112=mapInv(z112);

   Rdim z211;
   z211[1]=iv1[2];
   z211[2]=iv2[1];
   z211[3]=iv3[1];
   Spc x211=mapInv(z211);

   Vec a=x112-x111;
```

```
         Vec b=x121-x111;
         Vec c=x211-x111;
         volumes[i]=cpmabs(a.spat(b,c));
      }
   }
}

template <class Dat>
void ParOfSpc<Dat>::iniSub()
{
   Z mL=3;
   static Word loc("ParOfSpc<Dat>::iniSub()");
   CPM_MA
   DisPar dp(smallLength_,2*smallLength_);
      // 'discretization data'
   cpmassert(smallLength_>0,loc);
   Rdim x0;
   Rdim s0;
   x0[1]=0.5;
   x0[2]=0.5;
   x0[3]=0.5;
   s0[1]=0.5;
   s0[2]=0.5;
   s0[3]=0.5;
   subX=SubsetOfSpace::boxoid(x0,s0,toMap());
   subX.set_dp(dp);
   sub=V<SubsetOfSpace>(nTot);
   Z i;
   for (i=1;i<=nTot;i++){
      Zdim zd=linearCodingInv(i);
      Iv iv1=xv1[zd[1]];
      Iv iv2=xv2[zd[2]];
      Iv iv3;
      if (dimVec==3){
         iv3=xv3[zd[3]];
      }
      x0[1]=iv1.mean();
      x0[2]=iv2.mean();
      x0[3]=iv3.mean();
      s0[1]=iv1.radius();
      s0[2]=iv2.radius();
      s0[3]=iv3.radius();
      sub[i]=SubsetOfSpace::boxoid(x0,s0,toMap());
         // box transformed by toMap()
      sub[i].set_dp(dp);
   }
// defining box
   Z nC=4;
   if (dimVec==3) nC*=2;
   V<Rdim> corn(nC);
```

```
   corn[1]=Rdim(0,0);
   corn[2]=Rdim(0,1);
   corn[3]=Rdim(1,0);
   corn[4]=Rdim(1,1);
   if (dimVec==3){
      Rdim co=corn[1];
      co[3]=1;
      corn[5]=co;

      co=corn[2];
      co[3]=1;
      corn[6]=co;

      co=corn[3];
      co[3]=1;
      corn[7]=co;

      co=corn[4];
      co[3]=1;
      corn[8]=co;
   }

   corners=V<Spc>(nC);
   for (i=1;i<=nC;i++) corners[i]=mapInv(corn[i]);

   Rdim cen(0.5,0.5);
   if (dimVec==3) cen[3]=0.5;
   center=mapInv(cen);

   RefSys rs0;
   box=Box(corners,rs0);
   CPM_MZ
}

template <class Dat>
Z ParOfSpc<Dat>::repNegDim()const
// we have neighbors only in the direction where we have a torus
// In each direction we consider only one neighbor
// Thus we have 1+2^count neighbors
{
   Z res=1;
   if (isTorus1) res*=2;
   if (isTorus2) res*=2;
   if (isTorus3 && dimVec==3) res*=2;
   if (res==1) return 1;
   else return res+1;
}

template <class Dat>
V<Spc> ParOfSpc<Dat>::repNeg(const Spc& x)const
```

```
{
   Z mL=3;
   Word loc("ParOfSpc<Dat>::repNeg(const Spc& x)const");
   cpmmessage(mL,loc&" started");
   Z d=repNegDim();
   V<Spc> res(d,x);
   if (d==1) return res;
   Group g_1=!g1;
   Group g_2=!g2;
   Group g_3;
   if (isTorus3 && dimVec==3) g_3=!g3;

   Z l=1;
   if (isTorus1){
      l++;
      res[l]*=g1;
      l++;
      res[l]*=g_1;
   }
   if (isTorus2){
      l++;
      res[l]*=g2;
      l++;
      res[l]*=g_2;
   }
   if (isTorus3 && dimVec==3){
      l++;
      res[l]*=g3;
      l++;
      res[l]*=g_3;
   }

   cpmmessage(mL,loc&" done");
   return res;
}

template <class Dat>
Record ParOfSpc<Dat>::toRecord()const
{
   Record rc;
   Word sec="partition_data";
   rc.set(sec,"xL",dat.get_xL());
   rc.set(sec,"xU",dat.get_xU());
   rc.set(sec,"zL",dat.get_zL());
   rc.set(sec,"zU",dat.get_zU());
   rc.set(sec,"nL1",nL1);
   rc.set(sec,"nL2",nL2);
   rc.set(sec,"nL3",nL3);
   rc.set(sec,"nS1",nS1);
   rc.set(sec,"nS2",nS2);
```

```
   rc.set(sec,"nS3",nS3);
   rc.set(sec,"overlap1",overlap1);
   rc.set(sec,"overlap2",overlap2);
   rc.set(sec,"overlap3",overlap3);
   rc.set(sec,"isTorus1",isTorus1);
   rc.set(sec,"isTorus2",isTorus2);
   rc.set(sec,"isTorus3",isTorus3);
   return rc;
}

template <class Dat>
V< X2<Spc,Z> > ParOfSpc<Dat>::epLatticePrel
   (const V<X2<R,bool> >& vx, Rdim xL, Rdim xU, R& facCell)const
{
   Z mL=2; //2
   Z mL2=3; //3
   const R facCellReduc=0.7071; // 1./sqrt(2)
   const Z nTryMax=5;
   Word loc("epLatticePrel(...)");
   Z nTry=0;
again:
   if (nTry>0){
      facCell*=facCellReduc;
   }
   cpmassert(nTry<=nTryMax,loc);
   cpmmessage(1,loc&" started, facCell="&cpmwrite(facCell)&", nTry="&
      cpmwrite(nTry));

   Z n=vx.dim();
   if (n==1){
      Rdim pc=(xL+xU)*0.5;
      Spc xc=mapInv(pc);
      V< X2<Spc,Z> > res(1);
      res[1]=X2<Spc,Z>(xc,1);
      return res;
   }
   if (n==2){
      Spc xc1=mapInv(xL);
      Spc xc2=mapInv(xU);
      V< X2<Spc,Z> > res(2);
      res[1]=X2<Spc,Z>(xc1,1);
      res[2]=X2<Spc,Z>(xc2,1);
      return res;
   }
// Counting the carriers;
   Z i,nCarrier=0;
   for (i=1;i<=n;i++) if (vx[i].c2()==true) nCarrier++;
   Z nToner=n-nCarrier;
   cpmmessage("nCarrier on input: "&cpmwrite(nCarrier));
   cpmmessage("nToner on input: "&cpmwrite(nToner));
```

```
   R_Vector vc(nCarrier);
   V<Z> zc(nCarrier);
   R_Vector vt(nToner);
   V<Z> zt(nToner);
   Z it=0,ic=0;
   for (i=1;i<=n;i++){
      R r=vx[i].c1();
      bool c=vx[i].c2();
      if (c){
         ic++;
         vc[ic]=r;
         zc[ic]=i;
      }
      else{
         it++;
         vt[it]=r;
         zt[it]=i;
      }
   }

// Determining the spacing of the lattice
   R fracTC=(1.*nToner)/nCarrier;

   R_Vector mvc=CpmArrays::moments(vc);
   R_Vector mvt=CpmArrays::moments(vt);
   R rC=(mvc[1]+mvc[3])*facCell;
   R rT=(mvt[1]+mvt[3])*facCell;
      // mv.[3] is the standard deviation
   // These are the radii of bounding sheres to be taken into account
   // in real space. Since we will build the lattice in the unit cube
   // (square) we have to form the corresponding diameters in
   // the cube parameter space:
   Rdim x0;
   x0.set(0.5);
   Spc sp0=mapInv(x0);
      // central point in Spc controlled by ParOfSpc
      // not of the cell defined by xL,xU

   V<Spc> spC(dimVec);
   V<Spc> spT(dimVec);
   for (i=1;i<=dimVec;i++){
      Rdim dC,dT; // all components 0
      dC[i]=rC;
      dT[i]=rT; // i-th component set
      spC[i]=sp0+Vec(dC);
      spT[i]=sp0+Vec(dT);
   }
   V<Rdim> mapC(dimVec);
   V<Rdim> mapT(dimVec);
   for (i=1;i<=dimVec;i++){
```

```
      mapC[i]=map(spC[i]);
      mapT[i]=map(spT[i]);
   }
   Rdim maprC;
   Rdim maprT;
   for (i=1;i<=dimVec;i++){
      maprC[i]=mapC[i][i]-x0[i];
      maprT[i]=mapT[i][i]-x0[i];
   }
   // maprC gives the radii for carrier particles to be used in the unit
   // cube(square) in the 3(or 2) orthogonal space directions;
   // for T correspondingly.
// now we define a lattice for carrier positions first.
// characteristics: Lines in 1-direction, since this is the assumed
// direction of the magnetic field. Spacing in i-direction ( i=2,3):
// 2*(maprC[i]+maprT[i]).
// How many lines?.
   Rdim width;
   for (i=1;i<=dimVec;i++) width[i]=xU[i]-xL[i];
   V<Z> nC(dimVec);
   nC[1]=(Z)(width[1]/(2*maprC[1]));
   for (i=2;i<=dimVec;i++) nC[i]=(Z)(width[i]/(2*(maprC[i]+maprT[i])));
   for (i=1;i<=dimVec;i++) if (nC[i]<1) nC[i]=1;
   Z nTotC=nC[1]*nC[2];
   if (dimVec==3) nTotC*=nC[3];
      // nTotC is the number of carrier particles to be placed in total
      // if the volume defined by the xU[i] and xL[i] is to be filled
      // according to the carrier line scheme completely.
// Actually, we should use an integer multiple of our initial particle
// supply vx.
   R dy2,dy3,f2,f3;
   dy2=width[2]-nC[2]*2*(maprC[2]+maprT[2]);
      // I had (maprC[i]+maprT[i]) till 03-02-12
   if (dy2<0){
      nTry++;
      goto again;
   }

   f2=width[2]/(width[2]-dy2);
      // for filling the range of width[2], which in
      // the main application is related to the space defined by the
      // fill-in space
   if (dimVec==3){

      dy3=width[3]-nC[3]*2*(maprC[3]+maprT[3]);
      if (dy3<0){
         nTry++;
         goto again;
      }
      f3=width[3]/(width[3]-dy3);
```

```
   }

// Let us generate the list of carrier positions:
   cpmassert(nTotC>0,loc);
   V<Spc> xC(nTotC);

   Z j,k,l;
   Rdim pRun;
   l=0;
   R dsC1=2*maprC[1];
   R dsC2=2*(maprC[2]+maprT[2]);
   R dsC3=(dimVec==2 ? 0 : 2*(maprC[3]+maprT[3]));
      // factor 2 was missing till 03-02-20
   R s0C2=maprC[2]+maprT[2];
   R s0C3=(dimVec==2 ? 0 : maprC[3]+maprT[3]);
   dsC2*=f2; // adjusting to fill width[2]
   s0C2*=f2;
   if (dimVec==3){
      dsC3*=f3; // adjusting to fill width[3]
      s0C3*=f3;
   }
   for (i=1;i<=nC[1];i++){
      pRun[1]=xL[1]+dsC1*(i-0.5);
         // xL[1]+ and corresponding terms
         // in the interior loops and in the toner loops were
         // missing till 02-09-13
      for (j=1;j<=nC[2];j++){
         pRun[2]=xL[2]+s0C2+dsC2*(j-1);
         if (dimVec==2){
            xC[++l]=mapInv(pRun);
         }
         else{
            for (k=1;k<=nC[3];k++){
               pRun[3]=xL[3]+s0C3+dsC3*(k-1);
               xC[++l]=mapInv(pRun);
            }
         }
      }
   }

   V<Z> nT(dimVec);
   nT[1]=cpmsup(Z(1),(Z)(nC[1]*fracTC));
   for (i=2;i<=dimVec;i++){
      nT[i]=cpmsup(Z(1),(nC[i]-1));
   }

   Z nTotT=nT[1]*nT[2];
   if (dimVec==3) nTotT*=nT[3];

   R sLine=dsC1*cpmsup(Z(1),(nC[1]-1));
```

```
   // length of a toner line

R dsT1=sLine/nT[1]; // nT[1]>=1 !
   // space for a toner
R dsT2=dsC2;
R dsT3=dsC3;
R s0T2=s0C2+0.5*dsC2;
R s0T3=(dimVec==2 ? 0 : s0C3+0.5*dsC3);
cpmassert(nTotT>0,loc);
V<Spc> xT(nTotT);
Vec shift2, shift3;;
shift2[2]=0.75*rT;
shift3[3]=0.75*rT;
R r2,r3;
l=0;
for (i=1;i<=nT[1];i++){
   pRun[1]=xL[1]+0.5*(dsC1+dsT1)+(i-1)*dsT1;
   r2=2*(i%2)-1; // toggles between +1 and -1
         // this is more effective as a random shift
   for (j=1;j<=nT[2];j++){ // was <
      pRun[2]=xL[2]+s0T2+dsT2*(j-1);
      if (dimVec==2){
         xT[++l]=mapInv(pRun)+shift2*r2;
      }
      else{
         for (k=1;k<=nT[3];k++){  // was <
            pRun[3]=xL[3]+s0T3+dsT3*(k-1);
            r3=2*(k%2)-1;
            xT[++l]=mapInv(pRun)+shift2*r2+shift3*r3;
         }
      }
   }
}
Z nTotDev=nTotC+nTotT;
V<X2<Spc,Z> > res(nTotDev);
cpmmessage(loc&", nTotC="&cpmwrite(nTotC));
cpmmessage(loc&", nTotT="&cpmwrite(nTotT));
cpmmessage(loc&", nTotDev="&cpmwrite(nTotDev));

l=0;
for (i=1;i<=nTotC;i++){
   res[i].c1()=xC[i];
   cpmmessage(mL2,"i="&cpmwrite(i)&", xC[i][1]="
      &cpmwrite(xC[i][1])&", xC[i][2]="&cpmwrite(xC[i][2]));
   res[i].c2()=zc.cyc(++l);
}

l=0;
Z iAct=nTotC;
for (i=1;i<=nTotT;i++){
```

```
      iAct++;
      res[iAct].c1()=xT[i];
      cpmmessage(mL2,"i="&cpmwrite(i)&", xT[i][1]="
         &cpmwrite(xT[i][1])&", xT[i][2]="&cpmwrite(xT[i][2]));
      res[iAct].c2()=zt.cyc(++l);
   }
   cpmmessage(mL,loc&" done");
   return res;
}

template <class Dat>
V< X2<Spc,Z> > ParOfSpc<Dat>::epLattice(const V<X2<R,bool> >& vx,
      Rdim xL, Rdim xU, R facCell, Z nPar)const
// implements a very efficient strategy to reach the
// prescribed particle number nPar approximately
{
   Z mL=2;
   static Word loc("epLattice(...)");
   CPM_MA
   const R incr=1.1;
   const R decr=1./incr;
   if (nPar<=0){
        // here, only the facCell input is used
      cpmmessage(mL, loc&" to be done for nPar<=0");
      return epLatticePrel(vx,xL,xU,facCell);
   }
   else if (nPar<=2){
      cpmmessage(mL, loc&" to be done for 0<nPar<=2");
      return epLatticePrel(vx.resize(nPar),xL,xU,facCell);
   }
   else ;
   // now the facCell is taken as a starting point
   // to get approximately nPar as the dimension of the
   // result (return value)
   bool succ=0;
   R fac=facCell;
   V< X2<Spc,Z> > res=epLatticePrel(vx,xL,xU,fac);
   V< X2<Spc,Z> > resLast=res;
   Z d=res.dim();
   Z dLast=resLast.dim();
   Z dDiff=d-nPar; // >0 if we create more lattice points as we need;
   Z dDiffLast=dLast-nPar;
   if (dDiff==0){
      cpmmessage(mL, loc&" done for nPar==res.dim()");
      return res;
   }
      // thus we have to increase particle spacing
   R fid=(dDiff>0 ? incr : decr);
   while (!succ){
      fac*=fid;
```

```
      resLast=res;
      res=epLatticePrel(vx,xL,xU,fac);
      dLast=resLast.dim();
      d=res.dim();
      dDiff=d-nPar;
      dDiffLast=dLast-nPar;
      succ=(dDiff*dDiffLast<=0);
   }
   if (cpmabs(dDiff)<=cpmabs(dDiffLast)){
      cpmmessage(mL,loc&" done exit 1");
      return res;
   }
   else{
      cpmmessage(mL,loc&" done exit 2");
      return resLast;
   }
}

template <class Dat>
V<Spc> ParOfSpc<Dat>::testPoints(Z n)const
{
   V<Rdim> vp=dat.testPoints(n);
   Z vpn=vp.dim();
   V<Spc> res(vpn);
   for (Z i=1;i<=vpn;i++) res[i]=Spc(vp[i]);
   return res;
}

template <class Dat>
void ParOfSpc<Dat>::autoFrame1(Camera& ca, R autoFrameFactor)
{
   Z mL=3;
   Word loc("ParOfSpc<Dat>::autoFrame1(...)");
   CPM_MA
   if (autoFrameFactor<=0);
   else{
      Spc xp=center;
      Spc xc=ca.get_x();
      Vec shift=xp-xc;
      ca*=Group(shift);
      pair<V<C>,V<C> > pvv=ca.capt(corners);
      V<C> capt=pvv.first&pvv.second;
      Z dc=capt.dim();
      V<R> captx(dc);
      V<R> capty(dc);
      Z i;
      for (i=1;i<=dc;i++){
         captx[i]=capt[i][1];
         capty[i]=capt[i][2];
      }
```

```
      Iv ivx(captx);
      Iv ivy(capty);
      ivx.stretch(autoFrameFactor);
      ivy.stretch(autoFrameFactor);
      R dx=ivx.absVal();
      R dy=ivy.absVal();
      cpmassert(dy>0,loc);
      R aspRatObject=dx/dy;
      R aspRatFrame=ca.getGraph().getAspRat();
      R relFac=aspRatFrame/aspRatObject;
      if (aspRatFrame>=aspRatObject){
         ca.getGraph().setXY(ivx.stretch(relFac),ivy);
      }
      else{
         ca.getGraph().setXY(ivx,ivy.stretch(1./relFac));
      }
   }
   CPM_MZ
}
```

139 cpmparticleviewer.h

```
//? cpmparticleviewer.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_PARTICLEVIEWER_H_
#define CPM_PARTICLEVIEWER_H_

#include <cpminterval.h>
#include <cpmlinalg.h>
#include <cpmnumbers.h>
#include <cpmimg24.h>
#include <cpmcamera.h>
#include <cpmimage.h>
#include <cpmmultiimage.h>
#include <cpmr2func.h>

namespace CpmPhysics{

//////////////////////////// class ParticleViewer ////////////////////

   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmRoot::Word;
   using CpmRoot::C;
   using CpmRoot::L;

   using CpmArrays::V;
   using CpmArrays::VV;
   using CpmArrays::VVV;
   using CpmArrays::X2;
   using CpmArrays::R_Matrix;

   using CpmGeo::Iv;

   using CpmImaging::Image;
   using CpmImaging::MultiImage;
   using CpmImaging::Mask;
   using CpmImaging::Rect;
   using CpmImaging::Color;
   using CpmImaging::RED;
   using CpmImaging::GREEN;
   using CpmImaging::BLUE;
   using CpmImaging::Response;

   using CpmGraphics::Graph;
   using CpmGraphics::ColRef;
   using CpmGraphics::Img24;
```

```
   using CpmFunctions::R_Func;
   using CpmFunctions::R2_Func;



template <class Part, class Space, class Camera>
class ParticleViewer{ // particle viewer
// Visualizing distributed quantities ('densities' as
// contrasted to intensities) that arrise from a set of particles.
// Presently only the charge distribution is treated. The logic
// of parcellating the image plane of a camera to get a
// visual rendition of the density can easily be generalized to other
// quantities such as the kinetic energy.
// Addition 03-09-25: now all scalar value implemented in
// R Clust::sclVal(const Word& w) for some w (e.g. w="charge",
// or w="kinetic_energy" or w="velocity") are treated in the new function
// void showScalarValue(...)

   V<Part> vp;
public:
   ParticleViewer(const V<Part>& vpIn):
      vp(vpIn){}

   void showCharge( const Camera& cam, Z ny, bool pixelized=false)const;
   // notice that the mutable attribute cam.gr gets normally
   // changed, and that this is essential for the intended
   // functionality
   // special elaborate color coding for charge density
   // where a hypothetical 'unsigned total charge density'
   // (to which a particle charge q always contributes |q|)
   // controls the brightness of an area. Gray means the
   // neutrality with respect to the true (signed) charge
   // and blue saturation increases with net positive
   // charge density (all my movie-files show positive
   // (carrier) particles as blue) wheras red saturation increases
   // with negative charge density.

   void showScalarValue( const Word& prop,
      const Camera& cam, Z ny, bool pixelized=false)const;
   // shows on the screen window of cam a distributed (positive) property
   // which is determined by the behavior of Clust::sclVal(prop).
   // The distribution is constructed in the following manner.
   // All particles of vp get imaged by cam. The 'film plane'
   // gets pixelized into ny(height)*nx cells , where nx is determined
   // the aspect ratio of cam's Graph attribute. To each cell
   // a real value val is defined by initialization as 0
   // and updating any present value by the maximum of the
   // present value and the value of any particle the center-of-mass
   // lies in this cell.
   // This is especially suited for finding hot-spots of e.g. the
```

```
   // velocity. Not for all values of prop for which
   // Clust::sclVal(prop) is defined, this is expected to be useful.
   // For potential extensions to quantities not yet taken into
   // consideration, one has simply to extend the definition of
   // Clust::sclVal(Word).

};

template <class Part, class Space, class Camera>
void ParticleViewer<Part,Space,Camera>::showCharge(
   const Camera& cam, Z ny, bool pixelized)const
{
   bool styleTD=true;
      // if true, parts of space that are electrically neutral
      // by containing no charge at all get displayed as dark
      // Always: negative charge is read, positive one is blue

   using namespace CpmRoot;
   static bool firstrun=true;
   static Z imgCount=0;
   Z mL=1;
   Word loc("ParticleViewer::showScalarValue(...)");
   cpmmessage(mL,loc&" started");
   using CpmGeo::Iv;
   using CpmLinAlg::R2;
   using CpmRoot::L;
   using CpmGraphics::Img24;
   using CpmImaging::Response;

   Z i,j,np=vp.dim();
   Z nx=(Z)(ny*cam.getGraph().getAspRat());
   cpmassert(nx>0,loc);
   Iv ivx=cam.getGraph().getIvX();
   Iv ivy=cam.getGraph().getIvY();
   cpmassert(ivx.absVal()>0,loc);
   cpmassert(ivy.absVal()>0,loc);
   R valIni=0;
   VV<R> val0(ny,nx,valIni); // sic !
   VV<R> val1(ny,nx,valIni); // sic !
   VV<R> val2(ny,nx,valIni); // sic !

   bool restrict=true;
   for (i=1;i<=np;i++){
      Space pi=vp[i].get_x();
      C xi=cam.captureMono(pi);
      R qi=vp[i].sclVal("charge");
      Z ix=ivx.quant(xi[1],nx,restrict); // without restrict one gets
         // points outside the index range too
      Z iy=ivy.quant(xi[2],ny,restrict);
      Z iyc=ny-iy+1;
```

```
   val0[iyc][ix]+=qi; // cumulative channel
   if (qi>0){
      val1[iyc][ix]+=qi;
   }
   else{
      val2[iyc][ix]-=qi;
   } // registrating the value to the right box
}
// now we have registered each particle's contribution
// in the cumulative channel and in the approriate signed channel
R rMask=4.;
R sigmaMask=1.5;
static Mask mask;
static Response rsp0;
static Response rsp1;
static Response rsp2;
static R valAbs0;
static R valAbs1;
static R valAbs2;
if (firstrun){
   // my monochrome image class already provides
   // flexible convolution
   mask=Mask(rMask,R_Func::Gauss(0,sigmaMask));

   Image<R> im0(val0);
   im0=im0.conv(mask);
   val0=im0.toVV();

   Image<R> im1(val1);
   im1=im1.conv(mask);
   val1=im1.toVV();

   Image<R> im2(val2);
   im2=im2.conv(mask);
   val2=im2.toVV();

   valAbs0=0;
   valAbs1=0;
   valAbs2=0;
   for (i=1;i<=ny;i++){
      for (j=1;j<=nx;j++){
         R v0=cpmabs(val0[i][j]);
         R v1=val1[i][j];
         R v2=val2[i][j];
         if (v0>valAbs0) valAbs0=v0;
         if (v1>valAbs1) valAbs1=v1;
         if (v2>valAbs2) valAbs2=v2;
      }
   }
   rsp0.setGamma(1.);
```

```
   rsp1.setGamma(1.);
   rsp2.setGamma(1.);
   rsp0=cam.getResponse();
   rsp1=cam.getResponse();
   rsp2=cam.getResponse();
   rsp0.setRangeY(0,255);
   rsp1.setRangeY(0,255);
   rsp2.setRangeY(0,255);
   R valAbs=cpmsup(valAbs1,valAbs2);
   R security=1.2;
   rsp0.setRangeX(-security*valAbs0,security*valAbs0);
   rsp1.setRangeX(0,security*valAbs);
   rsp2.setRangeX(0,security*valAbs);
   firstrun=false;
}
else{
   Image<R> im0(val0);
   im0=im0.conv(mask);
   val0=im0.toVV();
   Image<R> im1(val1);
   im1=im1.conv(mask);
   val1=im1.toVV();
   Image<R> im2(val2);
   im2=im2.conv(mask);
   val2=im2.toVV();
}
VV<ColRef> imDat(ny,nx);
for (i=1;i<=ny;i++){
   for (j=1;j<=nx;j++){
      if (styleTD){
         R p1=rsp1.getRangeX().ramp(val1[i][j]);
         R p2=rsp2.getRangeX().ramp(val2[i][j]);
         Color col1(BLUE);
         Color col2(GREEN);
         Color col3(RED);
         R p12=cpminf(p1,p2);
         Color col=col1*p1+col2*p12+col3*p2;
         imDat[i][j]=col.toColRef();
      }
      else{
         R p0=rsp0.getRangeX().ramp(val0[i][j]);
         Color col(p0);
         imDat[i][j]=col.toColRef();
      }
   }
}
Img24 im(imDat);
MultiImage<L> imm(im);
Z method=(pixelized ? 0 : 1);
Z faithful=1;
```

```
   imm.markDirect(cam.getGraph(),method,faithful);
   cam.captureGraph();
   cpmmessage(mL,loc&" done");
}

template <class Part, class Space, class Camera>
void ParticleViewer<Part,Space,Camera>::showScalarValue(
   const Word& prop, const Camera& ca, Z ny, bool pixelized)const
{
   Z mL=3;
   Word loc("ParticleViewer<>::showScalarValue(...)");
   CPM_MA
   pixelized; // not used
   Response rsp=ca.getResponse();
   rsp.setLevels(10);
   bool adjust=!rsp.normalizeGamma();
   Z nx=(Z)(ny*ca.getGraph().getAspRat());
   Iv ivx=ca.getGraph().getIvX();
   Iv ivy=ca.getGraph().getIvY();
   V<R> vx=ivx.centers(nx);
   V<R> vy=ivy.centers(ny);
   Z np=vp.dim();
   Z i;
   Z count=0;
   R allInv=1./np;
   R_Matrix val(ny,nx); // indexed like an image matrix
   bool restrict=true; // true is OK
   for (i=1;i<=np;i++){
      Space pi=vp[i].get_x();
      C xi=ca.captureMono(pi);
      if ( ivx.contains(xi[1]) && ivy.contains(xi[2]) ){
         Z ix=ivx.quant(xi[1],nx,restrict); // without restrict one gets
         // points outside the index range too
         Z iy=ivy.quant(xi[2],ny,restrict);
         Z iyc=ny-iy+1;
         R propi=vp[i].sclVal(prop);
         val[iyc][ix]=CpmRootX::sup<R>(propi,val[iyc][ix]);
            // cumulation as for charges is much less attractive here
         count++;
         cpmprogress("density-loop",count*allInv);
      }
   }
   Iv riv;
   if (adjust){
      Iv ivRange=rsp.getRangeX();
      X2<R_Matrix,Iv> xri=val.adjustRange(ivRange);
      val=xri.c1();
      riv=xri.c2();
   }
   Z iPol=1; // interpolates acording to my standard image
```

```
      // interpolation (simplex method).
   R2_Func r0(val.toVV(),iPol,ivx,ivy);
   ca.getGraph().show(r0,rsp,ivx,ivy);
   if (adjust){
      R vL=riv.inf();
      R vU=riv.sup();
      if (cpmverbose>0)
         ca.getGraph().addText(cpm(vL)&" <=val<= "&cpm(vU),0.25,0.1);
   }
   CPM_MZ
}
} // namespace
#endif
```

140 cpmperm.h

```
//? cpmperm.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_PERM_H_
#define CPM_PERM_H_
/*
Class for permutations of the numbers
{1,2,...,n} for n arbitrary and set by a constructor of the class.
*/

#include <cpmvo.h>

namespace CpmArrays{

class R_Vector;

class Perm{ // group of permutations

   Vo<Z> rep;

   Perm(const Vo<Z>& rep_, bool b):rep(rep_){b;}
      // efficient but completely unsave, since permutation
      // nature of argument not checked.
      // The second argument avoids interference with the
      // public Perm constructor from lists
public:
   typedef Perm Type;
   CPM_INVERSION
   CPM_PRODUCT_C
   CPM_IO
   CPM_ORDER

   template <class T>
   explicit Perm(const Vo<T>& v):
      rep(v.permutationForIncreasingOrder(),true){}
      // 'king's road to permutations'. Let the argument be any array of
      // orderable objects. Then the permutation which brings them in
      // increasing order will be created.

   explicit Perm(const R_Vector& v);
    //  :rep(v.permutationForIncreasingOrder()){}
      // special case of the above

   explicit   Perm(Z n=1);
      // creates the identity permutation of the set {1,2,...,n}

   Perm( Z n, Z i, Z j);
```

```
      // Constructor for a transposition i <---> j in the set {1,2,...,n}

   Perm( Z n, Z i);
      // Constructor for a cyclic permutation k-->k+i mod(n)+1
      // the set {1,2,...,n}

   Z par()const;
      // parity

   V<Z> cycDec()const;
      // cycle decomposition

   // The resulting array iv characterizes the decomposition of v
   // into a product of cycles (it does not output the factors
   // themselves).
   // iv.dim() is the number of cycles and iv[i] is the length of the
   // i-th cycle. The cycles are orderd as they arrise when always the
   // smallest possible starting value is chosen for a new cycle.

   Z operator[](Z i)const{ return rep[i];}
      // no reference returned, since changing the components is stricly
      // forbidden. This would in most cases destroy the property of
      // being a permutation!

   Z dim()const{ return rep.dim();}
      // if the return value is n, the set {1,2,...,n} is the set
      // for which *this is a permutation (it may not all elements
      // of this set really change; actually the indentity permutation
      // changes none of them).

   V<Z> getRep()const{ return rep;}

   Perm ranVal(Z j=0)const;
      // The return value is a Perm res such that res.dim()==dim()=:n.
      // It is 'random' since it is obtained by ordering a
      // random-generated list of n real numbers
};

} // namespace

#endif
```

141 cpmperm.cpp

```
//? cpmperm.cpp
//? Status of work 2008-10-25.
//?
#include <cpmperm.h>
#include <cpmrvector.h>

using namespace CpmArrays;
using namespace CpmRoot;

Perm::Perm(const R_Vector& v):rep(v.permutationForIncreasingOrder()){}

Perm::Perm(Z n):rep(n)
{
   for (Z i=1;i<=n;i++) rep[i]=i;
}

Perm::Perm( Z n, Z s):rep(n)
{
   Z i,sr=s;
   while (sr<0) sr+=n; // does not change the meaning of sr
   while (sr>=n) sr-=n; // now 0 <= sr <= n-1
   for (i=1;i<=n-sr;i++) rep[i]=i+sr;
   for (i=n-sr+1;i<=n;i++) rep[i]=i+sr-n;
}

Perm::Perm( Z n, Z i, Z j):rep(n)
   // constructor for a transposition of i and j.
{
   if (i<1 || i>n) cpmerror("Perm", "transposing index out of range");
   if (j<1 || j>n) cpmerror("Perm", "transposing index out of range");
   for (Z k=1;k<=n;k++) rep[k]=k;
   rep[i]=j;
   rep[j]=i;
}

Perm Perm::operator * (const Perm& p)const
{
   Z n=dim();
   cpmassert(n==p.dim(),"Perm::operator * (const Perm& p)const");
   Perm res(n);
   for (Z i=1;i<=n;i++) res.rep[i]=rep[p[i]];
   return res;
}

Perm Perm::operator ! (void)const
   // Returns a Perm which is the inverse of *this.
{
```

```
     Vo<Z> repRes=rep.permutationForIncreasingOrder();
   return Perm(repRes,true);
}

V<Z> Perm::cycDec()const
 // The programm is tested with many examples.
 // Presently the unorthodox control structure looks well understandable
 // to the author.
{
   Z n=rep.dim(), i, i_start=0, i_current, m=n, i_cy=1;
   V<Z> present(n);
      // An integer-valued array which monitors whether an element is
      // still present in the list and waits for being grouped into a
      // cycle.
   V<Z> cyclelength(n);
   for (i=1;i<=n;i++) present[i]=1;
Newcycle:     // Begin with creating a new cycle
   for (i=1;i<=n;i++){
      if (present[i]){     // find smallest available starting point
         i_start=i;
         break;
      }
   }
   cyclelength[i_cy]++;
   i_current=i_start;
Newelement:
   present[i_current]=0; // this value of i_current should never be
      // chosen as a starting point
   m--;
   i_current=rep[i_current];
   if (i_current==i_start){
      if (m==0) goto End; else {i_cy++; goto Newcycle;}
   }
   else{
      cyclelength[i_cy]++;
      goto Newelement;
   }
End:
   n=i_cy;
   V<Z> vr(n);
   for (i=1;i<=n;i++) vr[i]=cyclelength[i];
   return vr;
}

namespace{
Z numberParity(Z i)
   // Parity of integers
{
   if (2*(i/2)==i) return 1; else return -1;
}
```

```
}

Z Perm::par()const
   // Parity of permutations based on cycle decomposition.
   // See S. Mac Lane, G. Birkhoff: Algebra, MacMillan Company, 1967
   // p.95-96, Theorem 17 and Corollary 2.
{
   V<Z> vr=cycDec();
   Z n=vr.dim();
   Z x=-n;
   for (Z i=1;i<=n;i++) x+=vr[i];
   return numberParity(x);
}

Z Perm::com(const Perm& p)const{ return rep.com(p.rep);}

bool Perm::prnOn(ostream& out)const
// return preliminary
{
   Z i,n=dim();
   CpmRoot::writeTitle("Perm",out);
   CpmRoot::write(n,out);
   for (i=1;i<=n;i++) CpmRoot::write(rep[i],out);
   return true;
}

bool Perm::scanFrom(istream& in)
// return preliminary
{
   Z i,n=dim();
   CpmRoot::read(n,in);
   for (i=1;i<=n;i++) CpmRoot::read(rep[i],in);
   return true;
}

Perm Perm::ranVal(Z j)const
{
   Z n=dim();
   R_Vector r(n,1);
   r=r.ranVal(j);
   return Perm(r);
}
```

142 cpmproperties.h

```
//? cpmproperties.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_PROPERTIES_H_
#define CPM_PROPERTIES_H_
/*
   Description: Declares basic quantities for a classes describing
      properties of particles, confining structures, and the system as
      a whole.
*/

#include <cpmrecordhandler.h>
#include <cpmimagingtools.h>
#include <cpmrvector.h>

namespace CpmProperties{  // a namespace for managing common data

   using namespace CpmRoot;
   using CpmRootX::RecordHandler;
   using CpmRootX::Record;
   using CpmRootX::Recordable;
   using CpmRootX::R1;
   using CpmRootX::Z1;
   using CpmRootX::B;
   using CpmImaging::Palette;
   using CpmImaging::Color;
   using CpmArrays::V;
   using CpmArrays::R_Vector;
   using CpmAlgorithms::CyclicAlarm;

R nonHarmonicMean(R x1, R x2);
// For combination of some physical quantities the property
// of the harmonic mean to let a 0 always dominate the result
// is inconvenient, so we use the simple mean instead. Could
// be changed to some L^p-mean if needed

R harmonicMean(R x1, R x2);
   // 1/res = (1/x1 + 1/x2)/2, => res=2*x1*x2/(x1+x2)
   // notice:
   //   harmonicMean(a,b) = harmonicMean(b,a)
   //   harmonicMean(a,a) = a
   //   harmonicMean(a,0) = 0
   //   harmonicMean(a,infinity) = 2*a
   // Important approximate relation:
   //   harmonicMean(a,b) '=' 2*a  for  a << b

R mixedMean(R x1, R x2);
```

```
   // harmonic mean normally but switches to nonHarmonicMean if at
   // least one argument is 0

R transitionFunction(const R& x, const R& xNear, const R& xFar);
   // makes a differentiable third-order polynomial transition from 1
   // valid for x<=xNear, and 0 valid for x>=xFar. Computationally very
   // effective.

struct SysControl{ // system control parameters

   typedef SysControl Type;
   CPM_IO

   static Z nyScalarFields;

   static Z nxElectricField;
   static Z nyElectricField;
   static Z markElectricField;

   static Z nxMagneticField;
   static Z nyMagneticField;
   static Z markMagneticField;

   static Z nxGravityField;
   static Z nyGravityField;
   static Z markGravityField;

   static R fieldGamma;

   static Z verbose;

   static Z fixedStep;
   static R intactRange;
   static R singRange;
   static Z noAdhesion;
      // adhesion forces=0 in the present program
   static Z noElasticRepulsion;
   static Z noFriction;
      // if this is different from 0 all energy dissipating mechanisms
      // in interaction get disabled
   static Z noInteractionChargeToWall;
   static Z noInteractionChargeToCharge;
   static Z noMagneticInteractionChargeToCharge;
   static Z noInteractionMagnetToMagnet;
   static Z noInteractionMassToMass;
   static Z noInteractionElectricField;
   static Z noInteractionElectricDipole;
   static Z noInteractionMagneticField;
   static Z noForceMagneticField;
   static Z noInteractionGravitationField;
```

```
static Z noMeanFieldPar;
static Z ignoreExternalSystems;
   // if this is true, all external systems (typically
   // instances of classes derived from ExtSys have
   // the mark(...) function and the field evaluation
   // 'switched off'
static Z coloredParticles;
static Z singleCharge;
static bool accMem;
   // acceleration memory
static bool newRigBodyInt;
   // new rigid body integrator
static R eta;
   // dynamical viscosity used for generating a damping force
   // on any particle according to Stokes' formula
static R vCrit;
static R aCrit;
static bool veryFar;
static Z bitmapToShow;
static Z potentialToShow;
   // selects between showing the electric potential (1)
   // scalar magnetical potential (2) and the gravitational
   // potential (3)
static Z howToMarkParticles;
   // 0 for classical representation by a cross of axes
   // 1 via WireFrame::globe
 static bool showMeanFil;
static bool showParticles;
static bool showPartFields;
static bool showPartition;
static bool showPrimaryChargesOnly;
static R nearFactor;
static R tSmall;
static R tStep;
static R tWait;
static R dtMult;
   // v*dt, with
   // dt=SysControl::tStep*SysControl::dtMult
   // is the distance that is shown to indicate the velocity
   // of a particle a value <=0 disables graphical representation of
   // particle velocities

SysControl(void){ini();}
   // trivial constructor which does not access files
   // and does not change the data. This is neccassary in order
   // to let
   //    SysControl sc;
   //    sc.prnOn(out);
   // write the presently valid values of SysControl data
```

```
   static void read(const RecordHandler& rch, const Word& sec, Z ped);
      // sets the values from rch, which is a data type conveniently
      // created from ini-file input. ped means pedantry: value 1
      // creates warnings if an item was not found in rch, and
      // 2 stops on failure in this case.

   SysControl(RecordHandler const& rch,
      Word const& sec="syscontrol", Z ped=1){read(rch,sec,ped);}

   static void ini();
      // resets the state of the static variables to the one valid after
      // initialization
};

struct SysControlState{ // system control parameters as non-static data
   // holds the same data (attributes) as SysControl, however here
   // the data are non-static so that there are different instances
   // of the class. Non-trivial setting of the values is intended
   // to be achieved by calling mem().
   // Notice that all members are class instances and thus are
   // automatically initialized.

   typedef SysControlState Type;
   CPM_IO

   Z1 nyScalarFields;

   Z1 nxElectricField;
   Z1 nyElectricField;
   Z1 markElectricField;

   Z1 nxMagneticField;
   Z1 nyMagneticField;
   Z1 markMagneticField;

   Z1 nxGravityField;
   Z1 nyGravityField;
   Z1 markGravityField;

   R1 fieldGamma;

   Z1 verbose;

   Z1 fixedStep;
   R1 intactRange;
   R1 singRange;
   Z1 noAdhesion;
   Z1 noElasticRepulsion;
   Z1 noFriction;
   Z1 noInteractionChargeToWall;
```

```
   Z1 noInteractionChargeToCharge;
   Z1 noMagneticInteractionChargeToCharge;
   Z1 noInteractionMagnetToMagnet;
   Z1 noInteractionMassToMass;
   Z1 noInteractionElectricField;
   Z1 noInteractionElectricDipole;
   Z1 noInteractionMagneticField;
   Z1 noForceMagneticField;
   Z1 noInteractionGravitationField;
   Z1 noMeanFieldPar;
   Z1 ignoreExternalSystems;
   Z1 coloredParticles;
   Z1 singleCharge;
   B  accMem;
   B  newRigBodyInt;
   R1 eta;
   R1 vCrit;
   R1 aCrit;
   B  veryFar;
   Z1 bitmapToShow;
   Z1 potentialToShow;
   Z1 howToMarkParticles;
   B  showMeanFil;
   B  showParticles;
   B  showPartFields;
   B  showPartition;
   B  showPrimaryChargesOnly;
   R1 nearFactor;
   R1 tSmall;
   R1 tStep;
   R1 tWait;
   R1 dtMult;

   SysControlState(void){}
      // trivial constructor

   void act()const;
      // copies the data of *this into the static equivalents and thus
      // controls the behavior of all functions depending on SysControl

   void mem();
      // copies the valid static values (of SysControl) into the
      // corresponding data of *this
};

////////////////////////////// ContactProperties ///////////////////////

struct ContactProperties{ // data describing particle contact

   typedef ContactProperties  Type;
```

```
   CPM_IO

// adhesion, not implemented in the present program
   R alphaAdhesion_;
   R relativeRangeAdhesion_;

// elasticity
   R E_;
      // a modified module of elasticity E (e.g. 3.2e9 for Polystyrol) in
      // SI units effectiveE = E/(1-mu*mu), mu=Poisson number (0.35 for
      // Polystyrol)
      // Thus effectiveE approximately equal to E
// tangential friction
   R mu_;
      // coefficient of friction
   R v0_;
      // a small regularizing velocity
// normal friction
   R scnr_;
      // square of coefficient of normal restitution

   ContactProperties(void):alphaAdhesion_(0.),relativeRangeAdhesion_(0.),
      E_(0.),mu_(0.),v0_(0.),scnr_(1.){}
        // default constructor should not contain I/O operations
        // for efficiency. For scnr_ is 1 the natural default value.
   ContactProperties(RecordHandler const& rch, Word const& section);
      // 'modern' input from a section of a ini-file
   ContactProperties combine(const ContactProperties& cp)const;
      // returns an instance, the components of which are the harmonic
      // mean of the components of *this and cp
};

////////////////// class ParticleProperties ////////////////////////////

struct ParticleProperties: public ContactProperties{
   // particle properties

   Palette color;
      // not a physical property; artificially assigned color for
      // visualization; We have in mind the constructor Color(Z i)
      // which creates INVISIBLE for i<0. The class Color needs too much
      // storage to be the right type to appear here
   R rho;
      // density in SI unit kg/(cubic meter)
   R sigma;
      // surface charge density in SI unit Coulomb/(square meter)
   R J;
      // magnetic polarization in SI unit Tesla
   R eps;
      // relative dielectric constant
```

```
   R fRelax;
      // inverse of a dielectric relax time

   typedef ParticleProperties Type;
   CPM_IO
   ParticleProperties():ContactProperties(),color(CpmImaging::STEREO),
      rho(0.),sigma(0.), J(0.),eps(1.), fRelax(0.){}
        // default constructor should not contain I/O operations
        // for efficiency. color is INVISIBLE for default

   explicit ParticleProperties(ContactProperties const& cp):
      ContactProperties(cp),color(CpmImaging::STEREO),
      rho(0.),sigma(0.), J(0.),eps(1.), fRelax(0.){}

   ParticleProperties(const RecordHandler& rch, const Word& section);
      // 'modern' input from a section of a ini-file

   void set_cp(const ContactProperties& cp)
     // {(ContactProperties)(*this)=cp;} dubious construction
      // setting the base class part of data from base class
   {
      alphaAdhesion_=cp.alphaAdhesion_;
      relativeRangeAdhesion_=cp.relativeRangeAdhesion_;
      E_=cp.E_;
      mu_=cp.mu_;
      v0_=cp.v0_;
      scnr_=cp.scnr_;
   }
};

//////////////////////// class BodyProperties ////////////////////////////

struct BodyProperties:public ContactProperties{//prop. surrounding bodies

   typedef BodyProperties  Type;
   typedef ContactProperties  Base;
   CPM_IO

// extendable list of material properties and method parameters

   R eps;
      // relative constant of dielectricity
   R d;
      // thickness of the dielectric layer

   BodyProperties(void):eps(1.),d(0.){}
        // default constructor should not contain I/O operations
        // for efficiency
   BodyProperties(const Base& b):
      Base(b),eps(1.),d(0.){}
```

```
   BodyProperties(const RecordHandler& rch, const Word& section);
      // 'modern' input from a section of a ini-file
};

//////////// structs NuggetData, DumbbellData, PairData ////////////////
//
// The following three structs serve as types for Clust constructor
// arguments. They do not only give a suitable means to handle the
// moderately large set of data going into the construction of a Clust,
// but also allow to distinquish by name several distinct methods of
// construction which can be considered to belong to different types
// of Clust's ('morphological types'). These types don't express
// themselves as different data types for describing a state but only by
// different structure of the values ('value-polymorphism'). To treat
// 'subspecies' of a type in this way evolved from my earlier method to
// use a Word-typed variable as a constructor argument to clearly
// indicate the nature of the construction process under consideration.
// To use different types instead of different string-values offers the
// essential advantage that the construction data may vary from method to
// method.

struct NuggetData{ // data of nugget-like grains
   // see Clust(R d, const ParticleProperties& pp, const NuggetData& pd);
   // or ComPar(R d, const ParticleProperties& pp, const NuggetData& pd);
   R1 overlapFac;
   R1 isotropyFac;
   Z1 np;
   Z1 trialMax;
   typedef  NuggetData Type;
   CPM_IO
   NuggetData(){}
   NuggetData(Z n, R overlap=1.75, R isotr=1.0):
   overlapFac(overlap),isotropyFac(isotr),np(n),trialMax(40000){}
   NuggetData(const RecordHandler& rch, const Word& section);
   NuggetData(Record const& rc, const Word& section);
};

struct DumbbellData{// data of dumbbell-shaped grains
   // see Clust(R d, const ParticleProperties& pp, const DumbbellData&
   // pd);
   R1 rc;
   R1 re;
   R1 phi;
   typedef  DumbbellData Type;
   CPM_IO
   DumbbellData(){}
   DumbbellData(const RecordHandler& rch, const Word& section);
   DumbbellData(Record const& rc, const Word& section);
};
```

```
struct PairData{// data of pair-shaped grains
   // see Clust(R d, const ParticleProperties& pp, const PairData& pd);
   R1 r1;
   R1 r2;
   R1 phi;
   typedef  PairData Type;
   CPM_IO
   PairData(){}
   PairData(const RecordHandler& rch, const Word& section);
   PairData(Record const& rc, const Word& section);
};

class ParEnsPrp: public Recordable{ // Particle Ensemble Properties
   // describes the design parameters of a particle ensemble using my
   // flexible new data type for such purposes

public:

   ParEnsPrp(){}

   ParEnsPrp(const RecordHandler& rch, const Word& sec):
      Recordable(rch.getRecord(),sec){}

   ParEnsPrp(const Record& rc, const Word& sec):Recordable(rc,sec){}
   ParEnsPrp(const Record& rc):Recordable(rc){}
};

} //

#endif
```

143 cpmproperties.cpp

```
//? cpmproperties.cpp
//? Status of work 2008-10-25.
//?
#include <cpmproperties.h>
#include <cpmconstphys.h>
#include <cpmrfunc.h>

using namespace CpmRoot;
using namespace CpmProperties;
using namespace CpmPhysics;
using CpmRootX::RecordHandler;

R CpmProperties::nonHarmonicMean(R x1, R x2){ return 0.5*(x1+x2);}

R CpmProperties::harmonicMean(R x1, R x2)
// a factor 2 was missing till 2000-3-10, without this factor we have
// harmonicMean(x,x) != x
{
   R nom=x1+x2;
   if (nom==0.){
      if (x1==0) return 0;
         // normal
      cpmwarning("CpmProperties::harmonicMean(...): x1+x2==0");
      cpmmessage("x1="&cpmwrite(x1));
      cpmmessage("x2="&cpmwrite(x2));
      return 0;
   }
   else{
      return (2*x1*x2)/nom;
   }
}

R CpmProperties::mixedMean(R x1, R x2)
{
   return (x1==0 || x2==0) ? nonHarmonicMean(x1,x2) : harmonicMean(x1,x2)\
   ;
}

R CpmProperties::transitionFunction(const R& x, const R& xNear,
   const R& xFar )
// this is a differentiable transit function
{
   if (x<=xNear) return 1;
   if (x>=xFar) return 0;
   cpmassert(xNear<xFar,"CpmProperties::transitionFunction(R,R,R)");
   R p=(x-xNear)/(xFar-xNear);
   return CpmFunctions::R_Func::hill3(p);
```

```
}

////////////////////// struct SysControl ////////////////////////////
// collects most of the control parameters for the PaLa program.
// Input of these quantities typically is via a 'ini-file' as described
// in the context of class RecordHandler

Z SysControl::nyScalarFields=32;
Z SysControl::nxElectricField=10;
Z SysControl::nyElectricField=6;
Z SysControl::markElectricField=-1;
Z SysControl::nxMagneticField=11;
Z SysControl::nyMagneticField=8;
Z SysControl::markMagneticField=-1;
Z SysControl::nxGravityField=12;
Z SysControl::nyGravityField=9;
Z SysControl::markGravityField=-1;
R SysControl::fieldGamma=0.25;
Z SysControl::verbose=0;
Z SysControl::fixedStep=0;
R SysControl::intactRange=1e100; // should never become active
R SysControl::singRange=1e-6;
Z SysControl::noAdhesion=1;
   // adhesion forces=0 in the present program anyway
Z SysControl::noElasticRepulsion=0;
Z SysControl::noFriction=0;
Z SysControl::noInteractionChargeToWall=1;
Z SysControl::noInteractionChargeToCharge=0;
Z SysControl::noMagneticInteractionChargeToCharge=1;
Z SysControl::noInteractionMagnetToMagnet=0;
Z SysControl::noInteractionMassToMass=1;
Z SysControl::noInteractionElectricField=0;
Z SysControl::noInteractionElectricDipole=0;
Z SysControl::noInteractionMagneticField=0;
Z SysControl::noForceMagneticField=0;
Z SysControl::noInteractionGravitationField=0;
Z SysControl::noMeanFieldPar=1; // clean 'classical' treatment
   // for small particle numbers, was 0 till 2005-03-26
Z SysControl::ignoreExternalSystems=0;
Z SysControl::coloredParticles=0;
Z SysControl::singleCharge=1;
bool SysControl::accMem=true;
   // true works extremely better for Lorentz forces,
   // for non-velocity dependent forces no obvious difference
bool SysControl::newRigBodyInt=true;
R SysControl::eta=0; // if needed, it should be set more transparently
//R SysControl::eta=1.72e-6; // air
R SysControl::vCrit=0; // no influence if <=0
R SysControl::aCrit=0; // no influence if <=0
   // critical acceleration smallest meaningful length
```

```
   // divided by smallest meaningful time squared
bool SysControl::veryFar=true;
Z SysControl::bitmapToShow=0;
Z SysControl::potentialToShow=1;
Z SysControl::howToMarkParticles=0;
bool SysControl::showMeanFil=false;
bool SysControl::showParticles=true;
bool SysControl::showPartFields=false;
bool SysControl::showPartition=false;
bool SysControl::showPrimaryChargesOnly=true;
R SysControl::nearFactor=1;
R SysControl::tSmall=1e-10;
R SysControl::tStep=0;
R SysControl::tWait=0;
R SysControl::dtMult=0;

void SysControl::ini()
{
   SysControl::nyScalarFields=32;
   SysControl::nxElectricField=10;
   SysControl::nyElectricField=6;
   SysControl::markElectricField=-1;
   SysControl::nxMagneticField=11;
   SysControl::nyMagneticField=8;
   SysControl::markMagneticField=-1;
   SysControl::nxGravityField=12;
   SysControl::nyGravityField=9;
   SysControl::markGravityField=-1;

   SysControl::fieldGamma=0.25;
   SysControl::verbose=0;
   SysControl::fixedStep=0;
   SysControl::intactRange=1e100; // should never become active
   SysControl::singRange=1e-6;
   SysControl::noAdhesion=1;
   SysControl::noElasticRepulsion=0;
   SysControl::noFriction=0;
   SysControl::noInteractionChargeToWall=1;
   SysControl::noInteractionChargeToCharge=0;
   SysControl::noMagneticInteractionChargeToCharge=1;
   SysControl::noInteractionMagnetToMagnet=0;
   SysControl::noInteractionMassToMass=1;
   SysControl::noInteractionElectricField=0;
   SysControl::noInteractionElectricDipole=0;
   SysControl::noInteractionMagneticField=0;
   SysControl::noForceMagneticField=0;
   SysControl::noInteractionGravitationField=0;
   SysControl::noMeanFieldPar=1;
   SysControl::ignoreExternalSystems=0;
   SysControl::coloredParticles=0;
```

```
   SysControl::singleCharge=1;
   SysControl::accMem=true;
   SysControl::newRigBodyInt=true;
   SysControl::eta=0;
   SysControl::vCrit=0;
   SysControl::aCrit=0;
   SysControl::veryFar=true;
   SysControl::bitmapToShow=0;
   SysControl::potentialToShow=1;
   SysControl::howToMarkParticles=0;
   SysControl::showMeanFil=false;
   SysControl::showParticles=true;
   SysControl::showPartFields=false;
   SysControl::showPartition=false;
   SysControl::showPrimaryChargesOnly=true;
   SysControl::nearFactor=1;
   SysControl::tSmall=1e-10;
   SysControl::tStep=0;
   SysControl::tWait=0;
   SysControl::dtMult=0;
}

bool SysControl::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("SysControl",str);
   cpmp(nyScalarFields);
   cpmp(nxElectricField);
   cpmp(nyElectricField);
   cpmp(markElectricField);
   cpmp(nxMagneticField);
   cpmp(nyMagneticField);
   cpmp(markMagneticField);
   cpmp(nxGravityField);
   cpmp(nyGravityField);
   cpmp(markGravityField);
   cpmp(fieldGamma);
   cpmp(verbose);
   cpmp(fixedStep);
   cpmp(intactRange);
   cpmp(singRange);
   cpmp(noAdhesion);
   cpmp(noElasticRepulsion);
   cpmp(noFriction);
   cpmp(noInteractionChargeToWall);
   cpmp(noInteractionChargeToCharge);
   cpmp(noMagneticInteractionChargeToCharge);
   cpmp(noInteractionMagnetToMagnet);
   cpmp(noInteractionMassToMass);
   cpmp(noInteractionElectricField);
   cpmp(noInteractionElectricDipole);
```

```
   cpmp(noInteractionMagneticField);
   cpmp(noForceMagneticField);
   cpmp(noInteractionGravitationField);
   cpmp(noMeanFieldPar);
   cpmp(ignoreExternalSystems);
   cpmp(coloredParticles);
   cpmp(singleCharge);
   cpmp(accMem);
   cpmp(newRigBodyInt);
   cpmp(eta);
   cpmp(vCrit);
   cpmp(aCrit);
   cpmp(veryFar);
   cpmp(bitmapToShow);
   cpmp(potentialToShow);
   cpmp(howToMarkParticles);
   cpmp(showMeanFil);
   cpmp(showParticles);
   cpmp(showPartFields);
   cpmp(showPartition);
   cpmp(showPrimaryChargesOnly);
   cpmp(nearFactor);
   cpmp(tSmall);
   cpmp(tStep);
   cpmp(tWait);
   cpmp(dtMult);
   return true;
}

bool SysControl::scanFrom(istream& str)
{
   cpms(nyScalarFields);
   cpms(nxElectricField);
   cpms(nyElectricField);
   cpms(markElectricField);
   cpms(nxMagneticField);
   cpms(nyMagneticField);
   cpms(markMagneticField);
   cpms(nxGravityField);
   cpms(nyGravityField);
   cpms(markGravityField);
   cpms(fieldGamma);
   cpms(verbose);
   cpms(fixedStep);
   cpms(intactRange);
   cpms(singRange);
   cpms(noAdhesion);
   cpms(noElasticRepulsion);
   cpms(noFriction);
   cpms(noInteractionChargeToWall);
```

```
   cpms(noInteractionChargeToCharge);
   cpms(noMagneticInteractionChargeToCharge);
   cpms(noInteractionMagnetToMagnet);
   cpms(noInteractionMassToMass);
   cpms(noInteractionElectricField);
   cpms(noInteractionElectricDipole);
   cpms(noInteractionMagneticField);
   cpms(noForceMagneticField);
   cpms(noInteractionGravitationField);
   cpms(noMeanFieldPar);
   cpms(ignoreExternalSystems);
   cpms(coloredParticles);
   cpms(singleCharge);
   cpms(accMem);
   cpms(newRigBodyInt);
   cpms(eta);
   cpms(vCrit);
   cpms(aCrit);
   cpms(veryFar);
   cpms(bitmapToShow);
   cpms(potentialToShow);
   cpms(howToMarkParticles);
   cpms(showMeanFil);
   cpms(showParticles);
   cpms(showPartFields);
   cpms(showPartition);
   cpms(showPrimaryChargesOnly);
   cpms(nearFactor);
   cpms(tSmall);
   cpms(tStep);
   cpms(tWait);
   cpms(dtMult);
   return true;
}

void SysControl::read(const RecordHandler& rch, const Word& sec, Z ped)
{
    rch.read(sec, "nyScalarFields", nyScalarFields,ped);
    rch.read(sec, "nxElectricField", nxElectricField,ped);
    rch.read(sec, "nyElectricField", nyElectricField,ped);
    rch.read(sec, "markElectricField", markElectricField,ped);
    rch.read(sec, "nxMagneticField", nxMagneticField,ped);
    rch.read(sec, "nyMagneticField", nyMagneticField,ped);
    rch.read(sec, "markMagneticField", markMagneticField,ped);
    rch.read(sec, "nxGravityField", nxGravityField,ped);
    rch.read(sec, "nyGravityField", nyGravityField,ped);
    rch.read(sec, "markGravityField", markGravityField,ped);
    rch.read(sec, "fieldGamma", fieldGamma,ped);
    rch.read(sec, "verbose", verbose,ped);
    rch.read(sec, "fixedStep", fixedStep,ped);
```

```
    rch.read(sec, "intactRange", intactRange,ped);
    rch.read(sec, "singRange", singRange,ped);
    rch.read(sec, "noAdhesion", noAdhesion,ped);
    rch.read(sec, "noElasticRepulsion", noElasticRepulsion,ped);
    rch.read(sec, "noFriction", noFriction,ped);
    rch.read(sec, "noInteractionChargeToWall",
       noInteractionChargeToWall,ped);
    rch.read(sec, "noInteractionChargeToCharge",
       noInteractionChargeToCharge,ped);
    rch.read(sec, "noMagneticInteractionChargeToCharge",
       noMagneticInteractionChargeToCharge,ped);
    rch.read(sec, "noInteractionMagnetToMagnet",
       noInteractionMagnetToMagnet,ped);
    rch.read(sec, "noInteractionMassToMass",noInteractionMassToMass,ped);
    rch.read(sec, "noInteractionElectricField",
      noInteractionElectricField,ped);
    rch.read(sec, "noInteractionElectricDipole",
      noInteractionElectricDipole,ped);
    rch.read(sec, "noInteractionMagneticField",
      noInteractionMagneticField,ped);
    rch.read(sec, "noForceMagneticField",noForceMagneticField,ped);
    rch.read(sec, "noInteractionGravitationField",
      noInteractionGravitationField,ped);
    rch.read(sec, "noMeanFieldPar",noMeanFieldPar,ped);
    rch.read(sec, "ignoreExternalSystems", ignoreExternalSystems,ped);
    rch.read(sec, "coloredParticles", coloredParticles,ped);
    rch.read(sec, "singleCharge", singleCharge,ped);
    rch.read(sec, "accMem", accMem,ped);
    rch.read(sec, "newRigBodyInt", newRigBodyInt,ped);
    rch.read(sec, "eta",eta,ped);
    rch.read(sec, "vCrit",vCrit,ped);
    rch.read(sec, "aCrit",aCrit,ped);
    rch.read(sec, "veryFar",veryFar,ped);
    rch.read(sec, "bitmapToShow",bitmapToShow,ped);
    rch.read(sec, "potentialToShow",potentialToShow,ped);
    rch.read(sec, "howToMarkParticles",howToMarkParticles,ped);
    rch.read(sec, "showMeanFil",showMeanFil,ped);
    rch.read(sec, "showParticles",showParticles,ped);
    rch.read(sec, "showPartFields",showPartFields,ped);
    rch.read(sec, "showPartition",showPartition,ped);
    rch.read(sec, "showPrimaryChargesOnly",showPrimaryChargesOnly,ped);
    rch.read(sec, "nearFactor",nearFactor,ped);
    rch.read(sec, "tSmall",tSmall,ped);
    rch.read(sec, "tStep",tStep,ped);
    rch.read(sec, "tWait",tWait,ped);
    rch.read(sec, "dtMult",dtMult,ped);
}
//////////////// class SysControlState ////////////////////////////////

bool SysControlState::prnOn(ostream& str)const
```

```
{
   CpmRoot::writeTitle("SysControlState",str);
   cpmp(nyScalarFields);
   cpmp(nxElectricField);
   cpmp(nyElectricField);
   cpmp(markElectricField);
   cpmp(nxMagneticField);
   cpmp(nyMagneticField);
   cpmp(markGravityField);
   cpmp(nxGravityField);
   cpmp(nyGravityField);
   cpmp(markMagneticField);
   cpmp(fieldGamma);
   cpmp(verbose);
   cpmp(fixedStep);
   cpmp(intactRange);
   cpmp(singRange);
   cpmp(noAdhesion);
   cpmp(noElasticRepulsion);
   cpmp(noFriction);
   cpmp(noInteractionChargeToWall);
   cpmp(noInteractionChargeToCharge);
   cpmp(noMagneticInteractionChargeToCharge);
   cpmp(noInteractionMagnetToMagnet);
   cpmp(noInteractionMassToMass);
   cpmp(noInteractionElectricField);
   cpmp(noInteractionElectricDipole);
   cpmp(noInteractionMagneticField);
   cpmp(noForceMagneticField);
   cpmp(noInteractionGravitationField);
   cpmp(noMeanFieldPar);
   cpmp(ignoreExternalSystems);
   cpmp(coloredParticles);
   cpmp(singleCharge);
   cpmp(accMem);
   cpmp(newRigBodyInt);
   cpmp(eta);
   cpmp(vCrit);
   cpmp(aCrit);
   cpmp(veryFar);
   cpmp(bitmapToShow);
   cpmp(howToMarkParticles);
   cpmp(potentialToShow);
   cpmp(showMeanFil);
   cpmp(showParticles);
   cpmp(showPartFields);
   cpmp(showPartition);
   cpmp(showPrimaryChargesOnly);
   cpmp(nearFactor);
   cpmp(tSmall);
```

```
   cpmp(tStep);
   cpmp(tWait);
   cpmp(dtMult);
   return true;
}

bool SysControlState::scanFrom(istream& str)
{
   cpms(nyScalarFields);
   cpms(nxElectricField);
   cpms(nyElectricField);
   cpms(markElectricField);
   cpms(nxMagneticField);
   cpms(nyMagneticField);
   cpms(markMagneticField);
   cpms(nxGravityField);
   cpms(nyGravityField);
   cpms(markGravityField);
   cpms(fieldGamma);
   cpms(verbose);
   cpms(fixedStep);
   cpms(intactRange);
   cpms(singRange);
   cpms(noAdhesion);
   cpms(noElasticRepulsion);
   cpms(noFriction);
   cpms(noInteractionChargeToWall);
   cpms(noInteractionChargeToCharge);
   cpms(noMagneticInteractionChargeToCharge);
   cpms(noInteractionMagnetToMagnet);
   cpms(noInteractionMassToMass);
   cpms(noInteractionElectricField);
   cpms(noInteractionElectricDipole);
   cpms(noInteractionMagneticField);
   cpms(noForceMagneticField);
   cpms(noInteractionGravitationField);
   cpms(noMeanFieldPar);
   cpms(ignoreExternalSystems);
   cpms(coloredParticles);
   cpms(singleCharge);
   cpms(accMem);
   cpms(newRigBodyInt);
   cpms(eta);
   cpms(vCrit);
   cpms(aCrit);
   cpms(veryFar);
   cpms(bitmapToShow);
   cpms(potentialToShow);
   cpms(howToMarkParticles);
   cpms(showMeanFil);
```

```
   cpms(showParticles);
   cpms(showPartFields);
   cpms(showPartition);
   cpms(showPrimaryChargesOnly);
   cpms(nearFactor);
   cpms(tSmall);
   cpms(tStep);
   cpms(tWait);
   cpms(dtMult);
   return true;
}

void SysControlState::mem()
{
   nyScalarFields=SysControl::nyScalarFields;
   nxElectricField=SysControl::nxElectricField;
   nyElectricField=SysControl::nyElectricField;
   markElectricField=SysControl::markElectricField;
   nxMagneticField=SysControl::nxMagneticField;
   nyMagneticField=SysControl::nyMagneticField;
   markMagneticField=SysControl::markMagneticField;
   nxGravityField=SysControl::nxGravityField;
   nyGravityField=SysControl::nyGravityField;
   markGravityField=SysControl::markGravityField;
   fieldGamma=SysControl::fieldGamma;
   verbose=SysControl::verbose;
   fixedStep=SysControl::fixedStep;
   intactRange=SysControl::intactRange;
   singRange=SysControl::singRange;
   noAdhesion=SysControl::noAdhesion;
   noElasticRepulsion=SysControl::noElasticRepulsion;
   noFriction=SysControl::noFriction;
   noInteractionChargeToWall=SysControl::noInteractionChargeToWall;
   noInteractionChargeToCharge=SysControl::noInteractionChargeToCharge;
   noMagneticInteractionChargeToCharge=
      SysControl::noMagneticInteractionChargeToCharge;
   noInteractionMagnetToMagnet=SysControl::noInteractionMagnetToMagnet;
   noInteractionMassToMass=SysControl::noInteractionMassToMass;
   noInteractionElectricField=SysControl::noInteractionElectricField;
   noInteractionElectricDipole=SysControl::noInteractionElectricDipole;
   noInteractionMagneticField=SysControl::noInteractionMagneticField;
   noForceMagneticField=SysControl::noForceMagneticField;
   noInteractionGravitationField=
      SysControl::noInteractionGravitationField;
   noMeanFieldPar=SysControl::noMeanFieldPar;
   ignoreExternalSystems=SysControl::ignoreExternalSystems;
   coloredParticles=SysControl::coloredParticles;
   singleCharge=SysControl::singleCharge;
   accMem=SysControl::accMem;
   newRigBodyInt=SysControl::newRigBodyInt;
```

```
   eta=SysControl::eta;
   vCrit=SysControl::vCrit;
   aCrit=SysControl::aCrit;
   veryFar=SysControl::veryFar;
   bitmapToShow=SysControl::bitmapToShow;
   potentialToShow=SysControl::potentialToShow;
   howToMarkParticles=SysControl::howToMarkParticles;
   showMeanFil=SysControl::showMeanFil;
   showParticles=SysControl::showParticles;
   showPartFields=SysControl::showPartFields;
   showPartition=SysControl::showPartition;
   showPrimaryChargesOnly=SysControl::showPrimaryChargesOnly;
   nearFactor=SysControl::nearFactor;
   tSmall=SysControl::tSmall;
   tStep=SysControl::tStep;
   tWait=SysControl::tWait;
   dtMult=SysControl::dtMult;
}

void SysControlState::act()const
{
   SysControl::nyScalarFields=nyScalarFields;
   SysControl::nxElectricField=nxElectricField;
   SysControl::nyElectricField=nyElectricField;
   SysControl::markElectricField=markElectricField;
   SysControl::markElectricField=markElectricField;
   SysControl::nxMagneticField=nxMagneticField;
   SysControl::nyMagneticField=nyMagneticField;
   SysControl::markMagneticField=markMagneticField;
   SysControl::nxGravityField=nxGravityField;
   SysControl::nyGravityField=nyGravityField;
   SysControl::markGravityField=markGravityField;
   SysControl::fieldGamma=fieldGamma;
   SysControl::verbose=verbose;
   SysControl::fixedStep=fixedStep;
   SysControl::intactRange=intactRange;
   SysControl::singRange=singRange;
   SysControl::noAdhesion=noAdhesion;
   SysControl::noElasticRepulsion=noElasticRepulsion;
   SysControl::noFriction=noFriction;
   SysControl::noInteractionChargeToWall=noInteractionChargeToWall;
   SysControl::noInteractionChargeToCharge=noInteractionChargeToCharge;
   SysControl::noMagneticInteractionChargeToCharge=
      noMagneticInteractionChargeToCharge;
   SysControl::noInteractionMagnetToMagnet=noInteractionMagnetToMagnet;
   SysControl::noInteractionMassToMass=noInteractionMassToMass;
   SysControl::noInteractionElectricField=noInteractionElectricField;
   SysControl::noInteractionElectricDipole=noInteractionElectricDipole;
   SysControl::noInteractionMagneticField=noInteractionMagneticField;
   SysControl::noForceMagneticField=noForceMagneticField;
```

```
   SysControl::noInteractionGravitationField=
      noInteractionGravitationField;
   SysControl::noMeanFieldPar=noMeanFieldPar;
   SysControl::ignoreExternalSystems=ignoreExternalSystems;
   SysControl::coloredParticles=coloredParticles;
   SysControl::singleCharge=singleCharge;
   SysControl::accMem=accMem;
   SysControl::newRigBodyInt=newRigBodyInt;
   SysControl::eta=eta;
   SysControl::vCrit=vCrit;
   SysControl::aCrit=aCrit;
   SysControl::veryFar=veryFar;
   SysControl::bitmapToShow=bitmapToShow;
   SysControl::potentialToShow =potentialToShow;
   SysControl::howToMarkParticles=howToMarkParticles;
   SysControl::showMeanFil=showMeanFil;
   SysControl::showParticles=showParticles;
   SysControl::showPartFields=showPartFields;
   SysControl::showPartition=showPartition;
   SysControl::showPrimaryChargesOnly=showPrimaryChargesOnly;
   SysControl::nearFactor=nearFactor;
   SysControl::tSmall=tSmall;
   SysControl::tStep=tStep;
   SysControl::tWait=tWait;
   SysControl::dtMult=dtMult;
}

//////////////////// class ContactProperties ////////////////////////////
// If the third argument is ped=0 the program will not stop
// if the quantity can't be read. This helps allows to stay compatible
// with data structures that are partly obsolete in newer versions of the
//
// program

ContactProperties::ContactProperties(RecordHandler const& rch,
   Word const& sec)
{
// trivial initialization in case that the file gave nothing
   alphaAdhesion_=0;
   relativeRangeAdhesion_=0;
   E_=0;
   mu_=0;
   v0_=0;
   scnr_=1;
   Z ped=1; // pedantic, value 1 warns on read failure
   // default value 2 stops, value 0 does not even warn
   rch.read(sec,"alphaAdhesion",alphaAdhesion_,ped);
   rch.read(sec,"relativeRangeAdhesion",relativeRangeAdhesion_,ped);
   rch.read(sec,"E",E_);
   rch.read(sec,"mu",mu_,ped);
```

```
   rch.read(sec,"v0",v0_,ped);
   R cnr=1;
   bool b=rch.read(sec,"cnr",cnr);
   if (b) scnr_=cnr*cnr;
}

ContactProperties ContactProperties::combine(
   const ContactProperties& cp)const
// combination by harmonic mean lacks a physical basis in most cases
{
   ContactProperties res;
   res.alphaAdhesion_=harmonicMean(alphaAdhesion_,cp.alphaAdhesion_);
   res.relativeRangeAdhesion_=harmonicMean(
   relativeRangeAdhesion_,cp.relativeRangeAdhesion_);
   res.E_=mixedMean(E_,cp.E_);
   res.mu_=nonHarmonicMean(mu_,cp.mu_);
   res.v0_=nonHarmonicMean(v0_,cp.v0_);
   res.scnr_=nonHarmonicMean(scnr_,cp.scnr_);
   return res;
}

bool ContactProperties::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("ContactProperties",str);
   cpmp(alphaAdhesion_);
   cpmp(relativeRangeAdhesion_);
   cpmp(E_);
   cpmp(mu_);
   cpmp(v0_);
   cpmp(scnr_);
   return true;
}

bool ContactProperties::scanFrom(istream& str)
{
   cpms(alphaAdhesion_);
   cpms(relativeRangeAdhesion_);
   cpms(E_);
   cpms(mu_);
   cpms(v0_);
   cpms(scnr_);
   return true;
}

////////////////////// class ParticleProperties //////////////////////

ParticleProperties::ParticleProperties(const RecordHandler& rch,
   const Word& sec):ContactProperties(rch,sec),color(CpmImaging::STEREO)
{
   Z ped=1; // pedantic
```

```
   Z col;
   bool b=rch.read(sec,"color",col,ped);
   if (b) color=Color::toPal(col);
// to get harmless initialization if nothing can be
// found on file
   rho=1;
   sigma=0;
   J=0;
   eps=1;
   fRelax=1;
   rch.read(sec,"rho",rho);
   rch.read(sec,"sigma",sigma,ped);
   rch.read(sec,"J",J,ped);
   rch.read(sec,"eps",eps,ped);
   rch.read(sec,"fRelax",fRelax,ped);
}

bool ParticleProperties::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("ParticleProperties",str);
   if (!ContactProperties::prnOn(str)) return false;
   cpmp(Color::toZ(color)); // color treated as Z for I/O
   cpmp(rho);
   cpmp(sigma);
   cpmp(J);
   cpmp(eps);
   cpmp(fRelax);
   return true;
}

bool ParticleProperties::scanFrom(istream& str)
{
   if (!ContactProperties::scanFrom(str)) return false;
   Z temp;
   cpms(temp);
   color=Color::toPal(temp);
   cpms(rho);
   cpms(sigma);
   cpms(J);
   cpms(eps);
   cpms(fRelax);
   return true;
}

//////////////////////////////// class BodyProperties ////////////////////

BodyProperties::BodyProperties(const RecordHandler& rch,
   const Word& sec): ContactProperties(rch,sec)
{
   Z ped=1;
```

```
   eps=1;
   d=0;
   rch.read(sec,"eps",eps,ped);
   rch.read(sec,"d",d,ped);
}

bool BodyProperties::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("BodyProperties",str);
   if (!ContactProperties::prnOn(str)) return false;
   cpmp(eps);
   cpmp(d);
   return true;
}

bool BodyProperties::scanFrom(istream& str)
{
   if (!ContactProperties::scanFrom(str)) return false;
   cpms(eps);
   cpms(d);
   return true;
}

//////////////////////// struct PairData ////////////////////////////

bool PairData::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("PairData",str);
   cpmp(r1);
   cpmp(r2);
   return true;
}

bool PairData::scanFrom(istream& str)
{
   cpms(r1);
   cpms(r2);
   return true;
}

PairData::PairData(const RecordHandler& rch, const Word& sec)
{
   r1=0;
   r2=0;
   rch.read(sec,"r1",r1);
   rch.read(sec,"r2",r2);
}

PairData::PairData(const Record& rc, const Word& sec)
{
```

```
   RecordHandler rch(rc);
   PairData res(rch,sec);
   *this=res;
}

//////////////////// struct DumbbellData ////////////////////////////

bool DumbbellData::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("DumbbellData",str);
   cpmp(rc);
   cpmp(re);
   cpmp(phi);
   return true;
}

bool DumbbellData::scanFrom(istream& str)
{
   cpms(rc);
   cpms(re);
   cpms(phi);
   return true;
}

DumbbellData::DumbbellData(const RecordHandler& rch, const Word& sec)
{
   rc=0;
   re=0;
   phi=0;
   rch.read(sec,"rc",rc);
   rch.read(sec,"re",re);
   rch.read(sec,"phi",phi);
}

DumbbellData::DumbbellData(const Record& rc, const Word& sec)
{
   RecordHandler rch(rc);
   DumbbellData res(rch,sec);
   *this=res;
}

/////////////////////// struct NuggetData ////////////////////////////

bool NuggetData::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("NuggetData",str);
   cpmp(overlapFac);
   cpmp(isotropyFac);
   cpmp(np);
   cpmp(trialMax);
```

```
   return true;
}

bool NuggetData::scanFrom(istream& str)
{
   cpms(overlapFac);
   cpms(isotropyFac);
   cpms(np);
   cpms(trialMax);
   return true;
}

NuggetData::NuggetData(const RecordHandler& rch, const Word& sec)
{
// a working initialization even if no data are on file
   overlapFac=1.25;
   isotropyFac=0.9;
   np=3;
   trialMax=5000;
   Z ped=1;
   rch.read(sec,"overlapFac",overlapFac,ped);
   rch.read(sec,"isotropyFac",isotropyFac,ped);
   rch.read(sec,"np",np,ped);
   rch.read(sec,"trialMax",trialMax,ped);
}

NuggetData::NuggetData(const Record& rc, const Word& sec)
{
   RecordHandler rch(rc);
   NuggetData res(rch,sec);
   *this=res;
}
```

144 cpmpsi.h

```
//? cpmpsi.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_PSI_H_
#define CPM_PSI_H_

/*
   Description: quantum wave functions in one dimension.
   Time development according to the explicit midpoint method.
*/

#include <cpmrfunc.h>
#include <cpmrmatrix.h>
#include <cpmcvector.h>
#include <cpmconstphys.h>

namespace CpmWaves{

using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::C;
using CpmRootX::Z1;
using CpmRootX::R1;
using CpmRootX::B;
using CpmArrays::C_Vector;
using CpmArrays::R_Vector;
using CpmArrays::R_Matrix;
using CpmFunctions::R_Func;
using CpmFunctions::F;
using CpmGeo::Iv;

const R hq=CpmPhysics::h__PHYS;
   // hQuer
const R hq2=hq*hq;
   // hQuer^2
const R hqi=1./hq;
   // hQuerInv
const C ihq=C(0,hq);
   // i*hQuer
const C ihqi=C(0,-hqi);
   // 1./(i*hQuer) = (i*hQuer)Inv

class Psi{ // 1-dimensional discrete wave function
   // evolving according to Schroedinger's equation given by particle mass
   // and a potential function

protected:
```

```
   Z1 d;
      // number of space points that make up the space
      // Thus d=psi.dim();
   Iv iv;
      // available space as an interval
   R1 m;
      // mass
   R1 k;
   Z1 method;
   R_Func V;
      // potential
   C_Vector dot(const C_Vector&);
      // creator of the time derivative, up to a factor ihqi
      // the Hamiltonian determined by m and V
      // H psi=-hq2*psi''/2m + k*V
   C_Vector psi;
   C_Vector psiDot;
   R1 dx;
   R_Vector x;
      // dependent
public:
   typedef Psi Type;
   CPM_IO

   Psi(){}
   Psi(Z d0, Iv iv0, R m0, R k0, const R_Func& V0);
   void setState(const R_Func& psiRe, const R_Func& psiIm=R_Func());
   void step(R dt);
   void dephase(R dPhi);
      // apply to the psi-value at each point a stochastically
      // chosed factor exp(i*dPhi*random) where random
      // varies in [0,1]
   Z dim()const{ return d;}
   R get_dx()const{ return dx;}
   Iv getSpace()const{ return iv;}
   R_Matrix rep()const;
      // graphical object representing the amplitude distribution
   R_Matrix repReal(R n=3)const;
      // real representation according to electrical engineer's
      // interpretation of complex numbers. The argument n
      // governs the resolution

   void setMethod(Z method0){ method=method0;}
   R norm()const{ return sqrt(normSquare());}
   R normSquare()const;
   R normalize();
      // makes a unit-vector
   C energy()const;
      // expectation value of the Hamiltonian. Could
```

```
      // get a small imaginary component due to numerical
      // in-accuracies, so we return a complex number

   R naturalTime()const;
      // a span of time associated with psi and psiDot
      // not taking account discretization

   R naturalTimeStep()const;
      // returns a time span which is a save time step
      // for the non-Euler time stepping methods.
      // Using only half this time step is even more save
      // and using twice the step seems to work also.
      // Taking 4 times this as a time step is asking for trouble.
      // Closely related to Courant's criterion

   void boost(R v);
      // active transformation in which an coordinate system in
      // relative motion plays the same role, that in the previous
      // function was played by a shifted system
      // See Fonda, Ghirardi p. 89, (2.111)
      // changes *this into the 'boosted state'
};

class HarmOsc: public Psi{ // 1-dimensional harmonic oscillator

   R1 f;
      // frequency of the oscillator
   R1 alpha; // see Leonhard I. Schiff pp 70,...

public:
   HarmOsc():Psi(){}
   HarmOsc(Z d0, R m0, R f0, R n0);
      // d0  number of space points that make up the space
      // m0 mass of the oscillating particle
      // f0 frequency of oscillation
      // n0 determines the space interval iv in a way that
      //    a state with quantum number n0 is vitually unaffected
      //    by the limitation of space
   R N_n(Z n)const; // normalization factor of eigen functions.
      // Nice recursive definition, not needed since states
      // get normalized anyway.
      // Schiff p.71 (13.17)
   void setQuantumNumber(Z n);
   R getAlpha()const{ return alpha;}
   R_Func eigenFunction(Z n)const;
      // is chosen real
};

} // namespace
```

```
#endif
```

145 cpmpsi.cpp

```
//? cpmpsi.cpp
//? Status of work 2008-10-25.
//?
#include <cpmpsi.h>

using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmArrays;
using namespace CpmFunctions;

using CpmWaves::Psi;
using CpmWaves::HarmOsc;

//////////////// class Psi ////////////////////////////////

Psi::Psi(Z d0, Iv iv0, R m0, R k0, const R_Func& V0):
   d(d0),iv(iv0),m(m0),k(k0),method(2),V(V0),psi(d),psiDot(d),x(d)
{
   dx=iv.absVal()/d;
   R x0=iv.inf()+0.5*dx;
   for (Z i=1;i<=d;i++){
      x[i]=x0;
      x0+=dx;
   }
}

C_Vector Psi::dot(const C_Vector& psi0)
{
   R fac=-hq2/(2*m*dx*dx);
   C_Vector res(d);
   cpmassert(d>1,"Psi::dot");
   for (Z i=1;i<=d;i++){
      C val=fac*(psi0.cyc(i-1)-2*psi0[i]+psi0.cyc(i+1));
      val+=k*V(x[i])*psi0[i]; // up to here this is the Hamiltonian
      val*=ihqi; // times 1/(i*hQuer)
      res[i]=val;
   }
   return res;
}

void Psi::setState(const R_Func& psiRe, const R_Func& psiIm)
{
   for (Z i=1;i<=d;i++){
      C val(psiRe(x[i]),psiIm(x[i]));
      psi[i]=val;
   }
   psiDot=dot(psi);
```

```
   normalize();
}

void Psi::step(R dt)
// intentionally coded rather compactly (following a proposal
// of Thomas Dera)
{
   if (method==0){ // Euler method
      psi.add(psiDot,dt);
      psiDot=dot(psi);
   }
   else if (method==1){ // explicit midpoint method for velocity
      R dth=dt*0.5;
      C_Vector v=dot(psi+psiDot*dth);
      psi.add(v,dt);
      psiDot=v*2-psiDot;
   }
   else { // explicit midpoint method for acceleration
      R dth=dt*0.5;
      psi.add(psiDot,dth);
      C_Vector a=dot(dot(psi));
         // acceleration for midpoint configuration
      psi.add(psiDot.add(a,dt),dth);
   }
}

R_Matrix Psi::rep()const
      // graphical object representing the amplitude distribution
{
   const Z mesLim=3;
   cpmmessage(mesLim,"Psi::rep() started");
   R_Matrix res(4,d);
   for (Z i=1;i<=d;i++){
      res[1][i]=x[i];
      res[2][i]=psi[i][1];
      res[3][i]=psi[i][2];
      res[4][i]=psi[i].absVal();
   }
   cpmmessage(mesLim,"Psi::rep() done");
   return res;
}

R_Matrix Psi::repReal(R n)const
//not OK since creating internal compiler error
{
   const R lambdaFactor=6.123;
   R_Vector re=psi.real();
   R_Vector im=psi.imag();
   R_Func reF(x,re);
   R_Func imF(x,im);
```

```
   Z dn=(Z)(n*d+0.5);
   R_Matrix res(2,dn);
   R xStart=x[1];
   R xEnd=x[d];
   R ds=(xEnd-xStart)/(dn-1);
   R fac=(dn-1.)/(d-1.);
   R s=xStart;
   R lambda=lambdaFactor*ds;
   R omega=2*Pi/lambda;
   for (Z i=1;i<=dn;i++){
      res[1][i]=s;
      R zre=reF(s);
      R zim=imF(s);
//      res[2][i]=(C(zre,zim)*CpmRoot::expI(omega*s))[1];
      s+=ds;
   }
   return res;
}

R Psi::normSquare()const
{
   return psi.absSqr();
}

R Psi::normalize()
{
   R pn=norm();
   R pnInv=1./pn;
   psi*=pnInv;
   psiDot*=pnInv;
   return pn;
}

C Psi::energy()const
{
   return (psi|psiDot)*ihq;
}

R Psi::naturalTime()const
{
   R p2=psi.absSqr();
   R pd2=psiDot.absSqr();
   R tSquare=p2/pd2;
   return sqrt(tSquare);
}

R Psi::naturalTimeStep()const
{
   const R fac=0.5;
   return fac*naturalTime()/d;
```

```
}

void Psi::boost(R v)
{
   C fac=v*m*ihqi;
   for (Z i=1;i<=d;i++){
      psi[i]*=exp(x[i]*fac);
   }
   psiDot=dot(psi);
}

void Psi::dephase(R dPhi)
      // apply to the psi-value at each point a stochastically
      // chosed factor exp(i*dt*phiDot*random) where random
      // varies in [0,1]
{
   Z j=0;
   for (Z i=1;i<=d;i++){
      psi[i]*=C::expi(dPhi*randomR(j++));
   }
}

bool Psi::prnOn(ostream& str)const
{
   cpmp(psi);
   cpmp(psiDot);
   return true;
}

bool Psi::scanFrom(istream& str)
{
   cpms(psi);
   cpms(psiDot);
   return true;
}

//////////////////// class HarmOsc ////////////////////////////

namespace {

   Z fac2(Z n) // (2^n)*(n!)
   {
      if (n==0) return 1;
      else return 2*n*fac2(n-1);
   }

   R hermite(R x, Z n)
   {
      if (n==0) return 1;
      else if (n==1) return 2*x;
```

```
      else return 2*x*hermite(x,n-1)-2*(n-1)*hermite(x,n-2);
   }

   R u(const R& x, const Z& n, const R& alpha)
   {
      R xi=alpha*x;
      return hermite(xi,n)*exp(-0.5*xi*xi);
   }

   R kOsc(R m, R f)
   {
      R omega=2*Pi*f;
      return omega*omega*m;
   }

   R alphaOsc(R m, R f)
   {
      R k=kOsc(m,f);
      R alphaSquare=::sqrt(m*k)/CpmWaves::hq; // Schiff p.67 (13.3)
      return ::sqrt(alphaSquare);
   }

   Iv ivOsc(R m, R f, R n)
   {
      const R saftyFac=1.4;

      using CpmRoot::Pi;
      using CpmWaves::hq;
      R e=4*Pi*(n+0.5)*hq*f;
      R k=kOsc(m,f);
      R x0=sqrt(e/k);
      x0*=saftyFac;
      return Iv(-x0,x0);
   }
}

HarmOsc::HarmOsc(Z d0, R m0, R f0, R n0):
Psi(d0,ivOsc(m0,f0,n0),m0,kOsc(m0,f0),R(0.5)*R_Func::x2),f(f0)
{
   alpha=alphaOsc(m,f);
}

R HarmOsc::N_n(Z n)const
{
   R resSquare=alpha/(sqrt(Pi)*fac2(n));
   return sqrt(resSquare);
}

void HarmOsc::setQuantumNumber(Z n)
{
```

```
   R_Func re=eigenFunction(n);
   setState(re);
}

R_Func HarmOsc::eigenFunction(Z n)const
{
   return F2<R,Z,R,R>(n,alpha)(u);
}
```

146 cpmq.h

```
//? cpmq.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_Q_H_
#define CPM_Q_H_
/*

   Description: Declaration of a class of rational numbers

*/
#include <cpmtypes.h>

namespace CpmRoot{

class Q { // rational numbers
   // Mainly used as descriptors for dimensions of physical quantities
   // (see class CpmPhysics::Si)

   Z qZ, qN;
   void normalize();
   Q(const Word& w, Z a, Z b):qZ(a),qN(b){}
      // not safe, but efficient, w has no influence
public:

   Q(void):qZ(0),qN(1){}

   explicit Q(Z a):qZ(a),qN(1){}

   Q(Z a, Z b):qZ(a),qN(b){normalize();}
      // gives a/b in standard form

   explicit Q(R x, R eps=1e-4, Z nOK=20);
      // constructor for q=m/n such that |x-q|<=eps and n minimal

   typedef Q Type;
   typedef Z ScalarType;
   CPM_IO
   CPM_ORDER
   CPM_TEST_MOD
   CPM_DESCRIPTORS
   CPM_CONJUGATION
   CPM_SUM_M
   CPM_PRODUCT_M
   CPM_SCALAR_M
      // this incudes Type + ScalarType, not only Type * ScalarType
   CPM_DIFFERENCE
   CPM_DIVISION
```

```
   operator R(void) const{ return ((R)qZ)/qN;}
      // automatic cast for elementary functions
      // conversion from Q to R

   Z numerator()const{ return qZ;}
      // evident, natural type is Z (integer)
      // German association: Z = Zähler

   Z denominator()const{ return qN;}
      // natural type is N (natural number)
      // German association: N = Nenner
};

} // namespace

#endif
```

147 cpmq.cpp

```
//? cpmq.cpp
//? Status of work 2008-10-25.
//?
#include <cpmq.h>
#include <cpmsystem.h>

using namespace CpmRoot;

void Q::normalize()
{
   if (qN==0){   // should not happen
      cpmerror("Q::normalize(): denominator = 0, not expected to happen");
   }

   if (qN<0){
      qN=-qN;
      qZ=-qZ;
   } // now qN is a natural number

   if (qZ==0){  // result is 0
      qZ=0;
      qN=1;
      return;
   }

   if (qZ==qN){  // thus result is 1
      qZ=1;
      qN=1;
      return;
   }

   Z pZ, sig;
   if (qZ>0){
      pZ=qZ;
      sig=1;
   }
   else{
      pZ=-qZ;
      sig=-1;
   } // pZ>0 always

// Now pZ,qN are positive and different

   Z a1,a2,a3;
   if (pZ>qN) {
      a1=pZ; a2=qN;
   }
```

```
   else {
      a1=qN; a2=pZ;
   }
   do {
      a3=a1%a2;
      a1=a2;
      a2=a3;
   } while (a3!=0);
   pZ /= a1;   // dividing by the greatest common divisor a1
   qN /=a1;
   qZ=pZ*sig;
}

Q::Q(R x, R eps, Z nOK)
{
   const Z nMax=10000;
   const R epsMin=1./nMax;
   const R epsExact=1e-10;
   using CpmRoot::absVal;
   cpmassert(eps>=epsMin,"Q::Q(R x, R eps, Z nOK)");
   R nr=1./eps;
   Z sig=1;
   if (x<0) {sig=-1; x=-x;}
   Z n_, nRes=(Z)ceil(nr), n=nRes;
   Z m, mRes=(Z)(x*n+0.5); // mRes/nRes meets
      // accuracy requirements

   while (nRes>nOK){ // trying to reduce magnitude of denominator
      m=(Z)(x*n+0.5);
      Q q(m,n);
      n_=q.denominator();
      if (n_<nRes){ // reduction successful
         nRes=n_;
         mRes=q.numerator();
         if ( absVal(x-(R)mRes/nRes)<=epsExact) goto LABLE;
      }
      n++; // enlarge denominator in order to be able to get a smaller
      // one after cencellation
      if (n>nMax){  // should stop
         cpmwarning("Q(R,R,Z)::numerator larger than required");
         goto LABLE;
      }
   }
LABLE:
   qZ=mRes*sig;
   qN=nRes;
}

Word Q::toWord()const
      // for a nice printable representation
```

```
{
   Word sep_i=""; // initial
   Word sep_c="/"; // central
   Word sep_f=""; // final separator
   Word res;
   if (qN==1) res=cpmwrite(qZ);
   else res=cpmwrite(qZ)&sep_c&cpmwrite(qN);
   return sep_i&res&sep_f;
}

Z Q::com(const Q& x) const
// natural order
{
   Z za=qZ * x.qN;
   Z zb=x.qZ * qN;
   if (za<zb) return 1;
   else if (za>zb) return -1;
   else return 0;
}

bool Q::prnOn(ostream& out )const
{ return write(qZ,out)&&write(qN,out);}

bool Q::scanFrom(istream& in ){ return read(qZ,in)&&read(qN,in);}

Word Q::nameOf(void) const{ return Word("Q");}

Q Q::inv(void)const
{
   if (qZ==0)
      cpmerror("Q::inv(): numerator = 0, not expected to happen");
   Word w("intentionally");
   if (qZ>0) return Q(w,qN,qZ); else return Q(w,-qN,-qZ);
      // fast constructor since qZ,qN are already free of common factors
}

Q Q::neg(void)const{ return Q("x",-qZ,qN);}

Q Q::net(Z i)const
{ if (i==1) return Q(Z(1),Z(1)); else return Q(Z(0),Z(1));}

Q Q::absVal(void)const{ return ( qZ>=0 ? *this : Q("x",-qZ,qN) );}

Q Q::absSqr(void)const{  return Q(qZ*qZ,qN*qN);}

Q Q::con(void)const{ return *this;}

Z Q::hashVal()const
{
   R a=qZ;
```

```
   a*=qN;
   return CpmRoot::hashVal(a);
}

namespace{

Z prod(Z a, Z b)
// Product of integers with overflow warning
{
   typedef long double Rld;
   Rld ar=(Rld)a, br=(Rld)b, cr;
   cr=ar*br;
   Z res=a*b;
   Rld indicator=cr-(Rld)res;
   if (indicator!=0.) cpmwarning("Q: Overflow in multiplying integers");
   return res;
}

} // namespace

Q& Q::operator +=(const Q& x)
{
   qZ=prod(qZ,x.qN) + prod(x.qZ,qN);
   qN=prod(qN,x.qN);
   normalize();
   return *this;
}

Q& Q::operator +=(const Z& s){ *this+=Q(s); return *this;}

Q& Q::operator *=(const Q& x)
{
   qZ=prod(qZ,x.qZ);
   qN=prod(qN,x.qN);
   normalize();
   return *this;
}

Q& Q::operator *=(const Z& z)
{
   qZ=prod(qZ,z);
   normalize();
   return *this;
}

Q Q::ranVal(Z j)const
{
   using CpmRoot::ranVal;
   using CpmRoot::absVal;
   using CpmRoot::posVal;
```

```
   const Z fZ=11;
   const Z fN=13;
   Z q1Z=qZ*fZ;
   Z q1N=qN*fN;
   Z j0=1000;
   Z j_=(j==0 ? 0 : ranVal(j0,j));
   Z a=ranVal<Z>(q1Z,j);
   R a1=absVal<Z>(a); a1=sqrt(a1); a=(Z)a1;
   Z b=1+posVal(ranVal<Z>(q1N,j+j_));
   R b1=(R)b; b1=sqrt(b1); b=(Z)b1;
   return Q(a,b);
}

Q Q::testVal(Z cpl)const
{
   Z a,b;
   a=3*7*CpmRoot::testVal<Z>(a,cpl);
   b=5*11;
   return Q(a,b);
}

R Q::disVal(const Q& z)const
{
   R r1=(R)(*this);
   R r2=(R)z;
   return CpmRoot::disVal(r1,r2);
}
```

148 cpmqm.h

```
//? cpmqm.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_QM_H_
#define CPM_QM_H_
/*
   Description: An attempt to code quantum theory more clearly
*/

#include <cpmvl.h>
#include <cpmc.h>
#include <cpmgraph.h>
#include <cpmrecordhandler.h>
#include <cpmclinalg.h>
#include <map>

using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::C;
using CpmRoot::Word;

using CpmRootX::B;
using CpmRootX::RecordHandler;

using CpmGeo::Iv;

using CpmArrays::Vl;
using CpmArrays::V;
using CpmArrays::VV;
using CpmArrays::X2;
using CpmArrays::X3;
using CpmArrays::R_Vector;
using CpmArrays::Outside;

using CpmFunctions::F;

using CpmGraphics::Graph;
using CpmGraphics::Img24;
using CpmGraphics::ColRef;

using CpmLinAlg::R2;
using CpmLinAlg::R3;
using CpmLinAlg::C2;
using CpmLinAlg::C4;
using CpmLinAlg::CC2;
using CpmLinAlg::CC4;
using CpmLinAlg::CCCC2;
```

```
using CpmLinAlg::Z4;

using CpmImaging::Color;

using std::map;
using std::vector;
using std::endl;

// List of classes to be introduced (in this order!) should help not to
// get lost:

// Lat1, DynDat, RunCon, DisCon,
// SysLat1, TrjLat1, IntActDat, SysLat2, SysLat3,
// TrjLat2, TrjLat3,
// Sep2, Sep3, Fun3, Ia, SpinN
// Ky. QregPar, Qreg, TrjQreg

// interfaces as declaration macros

#define CPM_QM_PURE_VECTOR_SPACE_INTERFACE(X,S)\
public:\
   X operator - (void)const;\
   X& operator *=(S);\
   X& operator +=(X const&);\
   X& operator -=(X const&);\
   X operator *(S)const;\
   X operator +(X const&)const;\
   X operator -(X const&)const;

#define CPM_QM_PURE_SCALAR_INTERFACE(X,S)\
public:\
    X& operator *=(S);\
    X operator *(S)const;

#define CPM_QM_SCALAR_PRODUCT_INTERFACE(X,S)\
public:\
   X con(void)const;\
   S operator |(X const&)const;\
   R normSqr(void)const{ return (operator|(*this)).real();}\
   R norm(void)const{ return ::sqrt(normSqr());}\
   R absVal(void)const{ return norm();}\
   R absSqr(void)const{ return normSqr();}

namespace CpmQM{

// Within this namespace we alway use units in which the numerical values
// of hDirac and c are 1. hPlanck thus has value 2*pi (=CpmRoot::Pi2).

typedef X2< V< V<R2> >, V<Word> > x2vv;
```

```
//////////////////////////// class Lat1 ////////////////////////////////////

class Lat1{ // lattice in one dimension
   // Many quantities of interest depend on a mass, which, however, is
   // not part of the concept. However, there is a natural maximum mass
   // associated with the lattice: mMax()

// independent quantities:
   Iv iv_;
      // biotope as an interval
   Z d_;
      // number of points in the biotope
// dependent quantities:
   R size_;
   R ds_;
   R m_;
      // maximum mass
   R p_;
      // maximum momentum

   void ini_()
   {
      cpmassert(d_>0,"Lat1::ini_()");
      size_=iv_.absVal();
      ds_=size_/d_;
      cpmassert(ds_>0,"Lat1::ini_()");
      m_=0.5/ds_;
      p_=CpmRoot::Pi2*m_;
   }
   typedef Lat1 Type;
public:
   CPM_IO
   CPM_ORDER
   CPM_NAM(Lat1)
   Lat1(Iv const& iv, Z d):iv_(iv),d_(cpmpos(d)){ini_();}
   Lat1(R size, Z d):iv_(size*0.5),d_(cpmpos(d)){ini_();}

   Iv iv()const{ return iv_;}
   Z d()const{ return d_;}
   R size()const{ return size_;}
   R ds()const{ return ds_;}
   R m()const{ return m_;}
   R fir()const{ return iv_.inf()+0.5*ds_;}
      //: first
      // returns the first of the discrete positions associated
      // with *this
   R last()const{ return iv_.sup()-0.5*ds_;}
      //: last
      // returns the last of the discrete positions associated
      // with *this
```

```
void next_1(R& x)const{ x+=ds_;}
   //: next
   // shifts the argument x by ds and thus provides a convenient
   // method for iterating over the discrete positions associated
   // with *this

R pos(Z i)const{ return iv_.inf()+ds_*(i+0.5); }
   // for i=0,1,...d_-1, x runs over the discrete positions associated
   // with *this

R frqMax()const{ return m_;}
   //: frequency maximum
   // returns the value of the spatial Nyquist frequency fc=d/(2*size)

R pMax()const{ return p_;}
   //: p maximum
   // pMax=hPlanck*fc

R mMax()const{ return m_;}
   //: m maximum
   // mMax()==frqMax(), due to hDirac=c=1, masses and spacial
   // frequencies have the same dimension.

R vMax(R mass)const{ return p_/mass;}
   //: v maximum
   // Velocity which gives a particle of mass m just the maximum
   // momentum pMax().
   // Notice that a boost operation is essentially a shift operation
   // in momentum space so that state components that get shifted
   // (formally) to values larger than the upper end of the momentum
   // range will enter the momentum range at the other end.
   // Thus applying a boost with vMax to a Gaussian wave packet (at
   // rest) will result in a wave packet with momentum components
   // arround fc and -fc. Under time evolution this packet will split
   // into two beams; for boosts with velocities close to vMax, these
   // two beams are also obvious but their relative strength varies
   // with v. Notice that for mass -> 0 we have vMax -> infinity,
   // so that vMax <c (=1) is not a natural requirement here.
   // Even for the largest allowed mass, vMax(mMax)=2*pi, so that
   // velocities much larger than 1 (=c) are possible.

R eKinMax(R mass)const{ R dsInv=1./ds_; return 2.*dsInv*dsInv/mass;}
   //: e kinetic maximum
   // eKinMax=(2/m)*(hDirac/ds)^2, ds=(size/d); maximum kinetic energy
   // in free motion if the Hamiltonian is realized by the wellknown
   // differencing scheme for the second derivative.
   // If one writes this as p^2/(2*m), we get p=2*hDirac/ds
   // This p, which also is a kind of maximal momentum turns out to be
   // smaller than pMax as defined previously. It is smaller in the way
```

```
      // 2 is smaller than pi. This is the least-surprising direction ---
      // the opposit would say that the fastest possible motion would
      // have momenta exceeding pMax. For the Hamilton operator defined
      // in momentum space (by discrete Fourier transformation) no such
      // discrepency would occur.

   R tRef(R mass)const{ return size_/vMax(mass);}
      //: t reference
      // time needed for traversing the distance size when travelling at
      // speed vMax(m)

   R dtLim(R mass)const{ return ds_/vMax(mass);}
      //: dt limit
      // time needed for traversing the distance ds() when travelling
      // at speed vMax(m)

   R lamOsc(R mass)const
      //: lambda oscillator
   {
      R x=CpmRoot::Pi2*d_/(size_*size_);
      return 0.5*x*x/mass;
   }
      // (hPlanck*d/(size^2))^2 / (2*mass)
      // Consider the Hamiltonian of a harmonic oscillator:
      // Observable p("p",d), x("x",d);
      // Observable h=p*p/(2*mass)+x*x*lambda;
      // Then, for lambda=QMBase::lambdaOsc(d), a state of maximum
      // momentum will just have the kinetic energy lambda*(size/2)^2.
      // A fictitious state having this momentum and being localized near
      // x=0, would oscillate with an amplitude size*0.5.
      // Let us write  mass = xi*mMax, then one gets
      // lambdaOsc(Z d) = ((2*Pi)^2 * c * hDirac)/(xi * size^3),
      // which is independent of d. This is not too surprising since d is
      // inherent in our parameterization of the  mass as xi*mMax.
   };

//////////////////////////// class DynDat ////////////////////////////////

class DynDat{ // dynamical data
   // Data for defining dynamics, i.e. finally a free and an interacting
   // Hamiltonian.
   R m_;
      //: m
      // mass
   R q_;
      //: q
      // charge, influencing interaction between
      // different particles
// data for defining the binding potential pot(R)
   R lam_;
```

```
      //: lambda
      // binding strength
   R x0_;
      //: x0
      // binding position
   R p_;
      //: p
      // power of binding potential.  The particle
      // at place x feels the potential
      // lambda*|x-x0|^p
// nature of boundary conditions
   Z out_; // to be interpreted as of type CpmArrays::Outside
      //: out

   static R beta_;
      //: beta
      // strength of a homogeneous field
      // Corresponds in a sense to the inter-particle
      // coupling constant alpha_ to be introduced in
      // class IntActDat.
   typedef DynDat Type;
public:
   CPM_IO
   CPM_NAM(DynDat)
   DynDat(R m, R q, R lam, R x0, R p, Z out=0 ):
    m_(cpmabs(m)),q_(q),lam_(lam),x0_(x0),p_(p),out_(out){}
   // m_(cpmabs(m)),q_(q),lam_(lam),x0_(x0),p_(p),out_(2){}
      // constructor
   R m()const{ return m_;}
      //: m
   R q()const{ return q_;}
      //: q
   R lam()const{ return lam_;}
      //: lambda
      // value of the coupling constant, see function pot()
   R x0()const{ return x0_;}
      //: x0
   void shf_(R s){ x0_+=s;}
      //: shift
   R pot(R x)const{ return lam_*::pow(cpmabs(x-x0_),p_)+beta_*q_*x;}
      //: potential
   R force(R x)const{ return -( x>=x0_ ? 1 : -1)*lam_*p_*
      ::pow(cpmabs(x-x0_),p_-1)-beta_*q_;}
      // force
      // force(x)=-dpot(x)/dx
   void setOut_(Outside out){ out_= (Z)out;}
      //: set out
      // setting the boundary boundary conditions
   void setLam_(R x){ lam_=x;}
      //: set lambda
```

```
   void mulLam_(R x){ lam_*=x;}
      //: multiply lambda
   void set_q_(R x){ q_=x;}
      //: set q
   void mul_q_(R x){ q_*=x;}
      //: multiply q
   Outside out()const{ return (Outside)out_;}
      //: outside
   static R& beta(){ return beta_;}
      //: beta
      // Access to beta_:
      // reading: R x=DynDat::beta();
      // writing: R y=...; DynDat::beta()=y;
      // Is this mechanism realy preferable to having
      // beta_ public?
};

// two structs for compact input from an ini-file
//////////////////////////////// class RunCon ////////////////////////////////
   struct RunCon{ // run control
      R dt;
      Z nIter;
      R tWaitIter;
      R gammaIter;
      Z nSteps;
      Z viewPeriod;
      Z docuPeriod;
      Z imgPeriod;
      R tWaitIni;
      R tWaitView;
      R tWaitDocu;
      R tWaitDocu2;
      R tWaitDocu3;
      R tWaitRes;
      Z sel;
      bool reversed;
      bool fromDisk;
      V<Z> imgSteps;
   public:
      RunCon(RecordHandler const& rch, Word const& sec);
   };

//////////////////////////////// class DisCon ////////////////////////////////
   struct DisCon{ // display control
      Z ipolMet;
      R gamma;
      R secFac;
      bool autoScl;
      bool showCor;
         //: show correlations
```

```
      bool showCls;
         //: show classical
         // show also corresponding data for classical particles
      bool areaSty;
         //: area style (prefer representation as a color
         // image ove line graphics
      bool wrtOnShow;
         //: write on show
   public:
      DisCon():ipolMet(-1),gamma(1),secFac(-1),autoScl(true){}
      DisCon(RecordHandler const& rch, Word const& sec)
      {
         cpmrh(ipolMet);
         cpmrh(gamma);
         cpmrh(secFac);
         cpmrh(autoScl);
         cpmrhf(showCor);
         cpmrhf(showCls);
         cpmrhf(areaSty);
         cpmrhf(wrtOnShow);
      }
   };

//////////////////////////// class SysLat1 ////////////////////////////////

   class SysLat1{ // quantum system on a one-dimensional lattice
      // The data which define a Hamiltonian are not part of the
      // class data; they enter as an argument of function dot_.
      Lat1 lat_;
         //: lattice
         // lattice in space
      Vl<C> rep_;
         //: representation
         // list of wave-function values on the
         // lattice positions
      void ini_()
         //: initializer
         // completes the construction of the zero-vector
      {
         Z d=lat_.d();
         rep_=Vl<C>(d);
      }
      void ini_(F<R,C> const& f);
         //: initializer
         // completes the construction of a vector getting
         // values from f
      void ini_(F<R,R> const& f);
         //: initializer
         // completes the construction of a vector getting
         // values from f
```

```
   typedef SysLat1 Type;
public:
   CPM_IO
   CPM_NAM(SysLat1)
   CPM_QM_PURE_VECTOR_SPACE_INTERFACE(Type,C)
   CPM_QM_SCALAR_PRODUCT_INTERFACE(Type,C)
   CPM_QM_PURE_SCALAR_INTERFACE(Type,R)

   SysLat1(Lat1 const& lat):lat_(lat){ini_();}
      // Constructs the zero vector.

   SysLat1(Lat1 const& lat, F<R,R> const& f):lat_(lat) {ini_(f);}
      // Constructs a unit vector such that rep_[i] is proportional
      // to f(lat_.pos()[i]) (with an i-independent factor).

   SysLat1(Lat1 const& lat, F<R,C> const& f):lat_(lat){ini_(f);}
      // Constructs a unit vector such that rep_[i] is proportional
      // to f(lat_.pos()[i]) (with an i-independent factor).

   SysLat1(Iv const& iv, Vl<C> const& v):lat_(iv,v.dim()),rep_(v){}
      // construction from biotope and wave function
      // no normalization!

   SysLat1(Lat1 const& lat, DynDat const& dd, Z n=0);
      // creates the n-th eigenstate (n=0 corresponds to
      // the ground state) of the Hamiltonian determined by
      // lat and dd.

   Lat1 lat()const{ return lat_;}
      //: lattice

   Vl<C> rep()const{ return rep_;}
      //: representation

   Z dim()const{ return lat_.d();}
      //: dimension

   C const& cui(Z i)const{ return rep_.cui(i);}
      //: cui (component for unchecked index)
      //       -              -         -
   C& cui(Z i){ return rep_.cui(i);}
      //: cui

   R dtLim(DynDat const& dd)const{ return lat_.dtLim(dd.m());}
      //: dt limit

   R nor_();
      //: normalize

   R s(R x0=0)const
```

```
   //: s (shift relative to x0)
   // expectation value of the particle position
   // relative to point x0. Thus <x-x0>, where
   // x is the particle position operator.
{
   R res=0;
   for (Z i=rep_.b();i<=rep_.e();++i)
      res+=rep_.cui(i).absSqr()*(lat_.pos(i)-x0);
   return res;
}

SysLat1& makeReal_(R p=2);
// returns a SysLat1 with all components of *this
// replaced by their absolute value to power p

R enrHomFil(DynDat const& dd)const;
   //: energy (in) homogeneous field

void dot_(DynDat const& dd);
   //: dot
   // changes *this into its time derivative under the
   // assumption that dynamical data dd are valid

C eTot(DynDat const& dd)const
   {
      SysLat1 v=*this;
      v.dot_(dd);
      return C::I*(*this|v);
   }

void mark(Graph& g, DisCon const& dc)const;
   //: mark

R dtSup(DynDat const& dd, Z p=100, R tWait=-1)const;
//: dt supremum
// upper limit for the time step according to
// the new insights from the preparation of the
// publication. Tries to estimate the largest
// energy eigenvalue (better: absolute value of this)
// if tWait>=0 it shows the process in graphics,
// each of the intermediate states will be shown on
// screen and tWait is the inspection time per screen
// image.

static void stepGam(R& t, SysLat1& x, SysLat1& v, R dt,
   DynDat const& dd)
   //: step gamma
   // the basic dynamical algorithms with all input in the form
   // of arguments (no internal state)
   //  eq. 8 in my paper
```

```
         // 'The direct midpoint method as a  quantum mechanical
         // integrator II'
         // Here it turned out that there are many advantages of such a
         // 'stateless' representation when one wants to use them in
         // various contexts (as in my case where different versions of
         // dynamics are to be compared.
      {
         R tau=dt*0.5;
         t+=tau;
         x+=v*tau;
         SysLat1 vmp(x);
         vmp.dot_(dd);
         v=vmp*2-v;
         t+=tau;
         x+=v*tau;
      }

      static void stepGam2(R& t, SysLat1& x, SysLat1& v, R dt,
         DynDat const& dd)
         //: step gamma 2
         // the basic dynamical algorithms with all input in the form
         // of arguments (no internal state)
         // norm conserving method
         // That two successive steps of stepGam_ are mor
         // natural than a single one was found 2007-11-25.
         // Now the curves of energy and norm have no longer
         // the alternating behavior which they had before.
         // Also the stability limit (according to my preliminary
         // accessment is now the same.)
      {
         R tau=dt*0.5;
         stepGam(t,x,v,tau,dd);
         stepGam(t,x,v,tau,dd);
      }

      static void step(R& t, SysLat1& x, SysLat1& v, R dt,
         DynDat const& dd)
         //: step
      {
          stepGam2(t,x,v,dt,dd);
      }
   };

////////////////////////////// class TrjLat1 ////////////////////////////

   class TrjLat1{ // trajectories of Lat1 systems
      // trajectories of a quantum system on a one-dimensional
      // lattice
      DynDat dd_;
      SysLat1 x_;
```

```
      SysLat1 v_;
      R t_;
      typedef TrjLat1 Type;
   public:
      CPM_IO
      CPM_NAM(TrjLat1)
      TrjLat1(DynDat const& dd, SysLat1 const& x, R t=0):dd_(dd),x_(x),
      v_(x_),t_(t){ v_.dot_(dd_);}
         // version in which the velocity is set from the state

      TrjLat1(DynDat const& dd, SysLat1 const& x,
         SysLat1 const& v, R t=0):dd_(dd),x_(x),v_(v),t_(t){}
         // version which also allows to set the velocity

      void step_(R dt){ SysLat1::step(t_,x_,v_,dt,dd_);}

      R t()const{ return t_;}
      R n()const{ return x_.absVal();}
      C eTot()const{ return C::I*(x_|v_);}
      C hInt()const;
      C eSep()const{ return eTot()-hInt();}
      R e0()const{ return eSep()[1];}
      R eInt()const{ return hInt()[1];}
      R s()const{ return x_.s(dd_.x0());}
      SysLat1 x()const{ return x_;}
      SysLat1 v()const{ return v_;}

      void mark(Graph& g, DisCon const& dc)const{ x_.mark(g,dc);}

      V< V<R2> > run_(Graph& g, RunCon& rc, DisCon const& dc);

   };

//////////////////////////// class IntActDat ////////////////////////////

   class IntActDat{ // interaction data
      R alpha_;
      R mu_;
      R eps_;
      R p_;
      typedef IntActDat Type;
   public:
      CPM_IO
      CPM_NAM(IntActDat)
      IntActDat(RecordHandler const& rch, Word const& sec)
      {
         cpmrh(alpha_);
         cpmrh(mu_);
         cpmrh(eps_);
         cpmrh(p_);
```

```
      }

      IntActDat(R alpha, R mu, R eps, R p):alpha_(alpha),
      mu_(mu),eps_(eps),p_(p){}

      R pot(R x)const
         //: potential
      {
         if (alpha_==0.) return 0;
         else{
            R r=cpmabs(x);
            return alpha_*::exp(-mu_*r)*::pow(eps_+r,-(p_+1));
         }
      }

      R fi(R x)const
         //: interaction force
      {
         if (alpha_==0.) return 0;
         else{
            R r=cpmabs(x);
            R y1=alpha_*::exp(-mu_*r);
            R re=eps_+r;
            R sig=( x>=0 ? 1 : -1);
            return sig*y1*(mu_*::pow(re,-p_-1)+(p_+1)*pow(re,-p_-2));
         }
      }

      void setAlp_(R a){ alpha_=a;}
         //: set alpha

      void free_(){alpha_=0;}
         //: free
         // setting all coupling constants equal to zero
         // switches interaction off

      R dis(R x, R y)const{ return cpmabs(x-y);}
         //: distance

      R& alpha(){ return alpha_;}
      R const& alpha()const{ return alpha_;}
   };

//////////////////////////// class SysLat2 ////////////////////////////////

   class SysLat2{ // quantum system on a two-dimensional lattice
      // The data which define a Hamiltonian are not part of the
      // class data; they enter as an argument of function dot_.
      Lat1 lat1_,lat2_;
         //: lattice
```

```
      // lattice in space
   Vl< Vl<C> > rep_;
      //: representation
      // list of wave-function values on the
      // lattice positions
    typedef SysLat2 Type;
public:
   CPM_IO
   CPM_NAM(SysLat2)
   CPM_QM_PURE_VECTOR_SPACE_INTERFACE(Type,C)
   CPM_QM_SCALAR_PRODUCT_INTERFACE(Type,C)
   CPM_QM_PURE_SCALAR_INTERFACE(Type,R)

   SysLat2(Lat1 const& lat1, Lat1 const& lat2, Vl< Vl<C> > const& rep):
   lat1_(lat1),lat2_(lat2),rep_(rep)
   {
      Word loc("SysLat2(Lat1,Lat1,Vl<Vl<C> >");
      cpmassert(lat1_.d()==rep_.dim(),loc); // lat1_d()>0 always
      cpmassert(rep_[0].dim()==lat2_.d(),loc);
   }

   SysLat2(SysLat1 const& s1, SysLat1 const& s2);
      // not normalized unless s1 and s2 are

   SysLat1 red1(SysLat1 const& s1)const;
      //: reduce 1
      // reduce first index via s1

   SysLat1 red2(SysLat1 const& s2)const;
      //: reduce 2
      // reduce second index via s2

   Z dim1()const{ return lat1_.d();}
      //: dimension 1

   Z dim2()const{ return lat2_.d();}
      //: dimension 2

   Vl<C>& cui(Z i){ return rep_.cui(i);}
   Vl<C> const& cui(Z i)const{ return rep_.cui(i);}

   R nor_();
      //: normalize

   void dot_(DynDat const& dd1, DynDat const& dd2,
      IntActDat const& dd12);
      //: dot
      // changes *this into its time derivative under the
      // assumption that dynamical data dd1, dd2, dd12
      // are in action
```

```
      C eTot(DynDat const& dd1, DynDat const& dd2,
         IntActDat const& dd12)const
         {
            SysLat2 v=*this;
            v.dot_(dd1,dd2,dd12);
            return C::I*(*this|v);
         }

      R_Vector stt(DynDat const& dd1, DynDat const& dd2)const;
         //: statistics

      void mark(Graph& g, Z ipolMeth=0, R gamma=0.5, R secFac=1.2)const;
       //: mark
      // showing the wave function in a way that the x-axis refers to
      // particle 1 and the y-axis ( oriented vertically upward!)
      // If the last argument is >0 it triggers an automatic
      // bright-level adjustment only in the first call and then
      // holds this level so that changes in brightness of individual
      // pixels can't occur as the result of a new level adjustment.

      R dtSup(DynDat const& dd1, DynDat const& dd2,
         IntActDat const& dd12, Z p=100, R tWait=0, R gamma=0.1)const;
      //: dt supremum
      // upper limit for the time step according to
      // the new insights from the preparation of the
      // publication. Tries to estimate the largest
      // energy eigenvalue (better: absolute value of this)
      // if tWait>0 it shows the process in graphics,
      // each of the intermediate states will be shown on
      // screen and tWait is the inspection time per screen
      // image.
   };

///////////////////////////// class SysLat3 ////////////////////////////////

   class SysLat3{ // quantum system on a three-dimensional lattice
      // The data which define a Hamiltonian are not part of the
      // class data; they enter as an argument of function dot_.
      Lat1 lat1_,lat2_,lat3_;
         //: lattice
         // lattice in space
      Vl< Vl< Vl<C> > >rep_;
         //: representation
         // list of wave-function values on the lattice positions
       typedef SysLat3 Type;
   public:
      CPM_IO
      CPM_NAM(SysLat3)
      CPM_QM_PURE_VECTOR_SPACE_INTERFACE(Type,C)
```

```
CPM_QM_SCALAR_PRODUCT_INTERFACE(Type,C)
CPM_QM_PURE_SCALAR_INTERFACE(Type,R)

SysLat3(SysLat1 const& s1, SysLat1 const& s2, SysLat1 const& s3);
   // not normalized unless s1 and s2 are

SysLat2 red3(SysLat1 const& s3)const;
   //: reduce 3
   // reduce last index via s3
   // results in a two dimensional system

SysLat2 red(Z i)const{ return SysLat2(lat2_,lat3_,rep_[i]);}
   // The wave function on the 3-dimensional lattice
   // can be represented by its values on 2-dimensional
   // sub-lattices (slices). Due to the strorage organization
   // of rep_ it is most efficient to let the slices be states
   // of particles 2 and 3 (thus 'slicing particle 1').

SysLat1 red12(SysLat1 const& s1, SysLat1 const& s2)const;
SysLat1 red23(SysLat1 const& s2, SysLat1 const& s3)const;
SysLat1 red31(SysLat1 const& s3, SysLat1 const& s1)const;


SysLat1 red12(SysLat2 const& s12)const;
SysLat1 red23(SysLat2 const& s23)const;
SysLat1 red31(SysLat2 const& s31)const;

Z dim1()const{ return lat1_.d();}
   //: dimension 1

Z dim2()const{ return lat2_.d();}
   //: dimension 2

Z dim3()const{ return lat3_.d();}
   //: dimension 2

R nor_();
   //: normalize

void dot_(DynDat const& dd1, DynDat const& dd2, DynDat const& dd3,
   IntActDat const& dd12);
   //: dot
   // changes *this into its time derivative under the
   // assumption that dynamical data dd1, dd2, dd12
   // are in action

C eTot(DynDat const& dd1, DynDat const& dd2, DynDat const& dd3,
   IntActDat const& dd12)const
   {
      SysLat3 v=*this;
```

```
            v.dot_(dd1,dd2,dd3,dd12);
            return C::I*(*this|v);
         }

      R_Vector stt(DynDat const& dd1, DynDat const& dd2,
         DynDat const& dd3)const;
         //: statistics
         // returns the expectation values <s1>, <s2>, <s3>,
         // <s1*s2>, <s1*s3>, <s2*s3>, <s1*s2*s3>, where si is
         // the position operator of particle i relative to the
         // equilibrium position of this particle.

      void mark(Graph& g, DisCon const& dc)const;
         //: mark
         // showing the wave function in a way that the x-axis refers to
         // particle 1 and the y-axis ( oriented vertically upward!)
         // If the last argument is >0 it triggers an automatic
         // bright-level adjustment only in the first call and then
         // holds this level so that changes in brightness of individual
         // pixels can't occur as the result of a new level adjustment.

      vector< vector<C> > transform(Graph& g, DisCon const& dc)const;
         // returns a version of rep_ which consists of
         // dim1()+dim2()+dim3() C-arrays which allow to build a
         // trajectory by concatenating the state data that the present
         // function provides. Since fast concatenation is essential
         // in this context, one uses std::vector as the data structure
         // of the return value.

      R dtSup(DynDat const& dd1, DynDat const& dd2, DynDat const& dd3,
         IntActDat const& dd12, Z p=100, R tWait=0, R gamma=0.1)const;
         //: dt supremum
         // upper limit for the time step according to
         // the new insights from the preparation of the
         // publication. Tries to estimate the largest
         // energy eigenvalue (better: absolute value of this)
         // if tWait>0 it shows the process in graphics,
         // each of the intermediate states will be shown on
         // screen and tWait is the inspection time per screen
         // image.
   };

////////////////////////////// class TrjLat2 ////////////////////////////

   class TrjLat2{ // trajectories of Lat2 systems
      // trajectories of a quantum system on a two-dimensional
      // lattice
   protected:
      DynDat dd1_,dd2_;
      IntActDat dd12_;
```

```
   SysLat2 x_;
   SysLat2 v_;

   R t_;
   typedef TrjLat2 Type;
public:
   CPM_IO
   CPM_NAM(TrjLat2)

   TrjLat2(DynDat const& dd1, DynDat const& dd2, IntActDat dd12,
      SysLat2 const& x, R t=0): dd1_(dd1),dd2_(dd2),dd12_(dd12),x_(x),
      v_(x_),t_(t){ v_.dot_(dd1_,dd2_,dd12_);}

   TrjLat2(DynDat const& dd1, DynDat const& dd2, IntActDat dd12,
      SysLat1 const& x1, SysLat1 const& x2, R t=0): dd1_(dd1),
      dd2_(dd2),dd12_(dd12),x_(x1,x2), v_(x_),t_(t)
   { v_.dot_(dd1_,dd2_,dd12_);}

   TrjLat2(DynDat const& dd1, DynDat const& dd2, IntActDat dd12,
      SysLat2 const& x, SysLat2 const& v, R t=0):
      dd1_(dd1),dd2_(dd2),dd12_(dd12),x_(x),v_(v),t_(t){}
      // version which also allows to set the velocity

   R t()const{ return t_;}
   R n()const{ return x_.absVal();}
   C xv()const{ return (x_|v_);}
   C eTot()const{ return C::I*(x_|v_);}
   C hInt()const;
      // irrelevant for dynamics
   SysLat2 x()const{ return x_;}
   SysLat2 v()const{ return v_;}
   DynDat dd1()const{ return dd1_;}
   DynDat dd2()const{ return dd2_;}
   IntActDat dd12()const{ return dd12_;}

   R_Vector stt()const{ return x_.stt(dd1_,dd2_);}

   R dtSup( Z p=100, R tWait=0, R gamma=0.1)const
   { return x_.dtSup(dd1_,dd2_,dd12_,p,tWait,gamma);}

   void stepGamTil_(R dt) // direct midpoint integrator
      //: step gamma tilde
   {
      R tau=dt*0.5;
      x_+=v_*tau;
      SysLat2 acc(x_);
      acc.dot_(dd1_,dd2_,dd12_);
      acc.dot_(dd1_,dd2_,dd12_);
      v_+=acc*dt;
      x_+=v_*tau;
```

```
         t_+=dt;
      }

      void stepGam_(R dt)
         //: step gamma
      {
         R tau=dt*0.5;
         x_+=v_*tau;
         SysLat2 vm(x_);
         vm.dot_(dd1_,dd2_,dd12_);
         v_=vm*2-v_;
         x_+=v_*tau;
         t_+=dt;
      }

      virtual void step_(R dt)
      {
         bool dmi=true; // direct midpoint integrator
            // false only for experimentation, TrjLat3
            // always works with direct midpoint method

         if (dmi){
            stepGamTil_(dt);
         }
         else{
            R tau=dt*0.5;
            stepGam_(tau);
            stepGam_(tau);
         }
      }

      virtual void mark(Graph& g, DisCon const& dc)const
      { x_.mark(g,dc.ipolMet,dc.gamma,dc.secFac);}

      V< V<R2> > run_(Graph& g, RunCon& rc, DisCon const& dc);
      //x2vv run_(Graph& g, RunCon& rc, DisCon const& dc);
         //: run
         // Since the first two arguments are non-constant references
         // the C+- conformal name would be run_1_2_(...)
   };

////////////////////////////// class TrjLat3 ////////////////////////////

   class TrjLat3{ // trajectories of Lat3 systems
   // trajectories of a quantum system on a
   // three-dimensional lattice
   protected:
      DynDat dd1_,dd2_,dd3_;
      IntActDat dd12_;
      SysLat3 x_;
```

```
   SysLat3 v_;

   R t_;
   typedef TrjLat3 Type;
public:
   CPM_IO
   CPM_NAM(TrjLat3)

   TrjLat3(DynDat const& dd1, DynDat const& dd2, DynDat const& dd3,
      IntActDat dd12, SysLat3 const& x, R t=0):
      dd1_(dd1),dd2_(dd2),dd3_(dd3),
      dd12_(dd12),x_(x),v_(x_),t_(t){ v_.dot_(dd1_,dd2_,dd3_,dd12_);}

   TrjLat3(DynDat const& dd1, DynDat const& dd2, DynDat const& dd3,
      IntActDat dd12, SysLat3 const& x, SysLat3 const& v, R t=0):
      dd1_(dd1),dd2_(dd2),dd3_(dd3),dd12_(dd12),x_(x),v_(v),t_(t){}
      // version which also allows to set the velocity

   R t()const{ return t_;}
   R n()const{ return x_.absVal();}
   C xv()const{ return (x_|v_);}
   C eTot()const{ return C::I*(x_|v_);}
   C hInt()const;
   SysLat3 x()const{ return x_;}
   SysLat3 v()const{ return v_;}
   DynDat dd1()const{ return dd1_;}
   DynDat dd2()const{ return dd2_;}
   DynDat dd3()const{ return dd3_;}
   IntActDat dd12()const{ return dd12_;}

   R_Vector stt()const{ return x_.stt(dd1_,dd2_,dd3_);}
      //: statistics

   R dtSup( Z p=100, R tWait=0, R gamma=0.1)const
   { return x_.dtSup(dd1_,dd2_,dd3_,dd12_,p,tWait,gamma);}

   virtual void step_(R dt)
      // direct midpoint integrator
   {
      R tau=dt*0.5;
      x_+=v_*tau;
      SysLat3 acc(x_);
      acc.dot_(dd1_,dd2_,dd3_,dd12_);
      acc.dot_(dd1_,dd2_,dd3_,dd12_);
      v_+=acc*dt;
      x_+=v_*tau;
      t_+=dt;
   }

   virtual void mark(Graph& g, DisCon const& dc)const
```

```
      { x_.mark(g,dc);}

      x2vv run_(Graph& g, RunCon& rc, DisCon const& dc);
         //: run
         // Since the first two arguments are non-constant references
         // the C+- conformal name would be run_1_2_(...)

      x2vv run_(Graph& g, RecordHandler const& rch);
         // testing the idea whether one gets flexability in this way
         // One gets! I should have chosen this approach from the
         // beginning. Nevertheless I make no use of it here for
         // staying compatible with previous results.
         // But for later use: Functions which directly control
         // a program run should get all their data from one argument
         // of type RecordHandler. In this way the implementation of new
         // control modalities does not imply the need for new arguments
         // or the definition of new aggregated data types.
   };

////////////////////////////// class Sep2 ////////////////////////////
// separated dynamics works and satisfies energy conservation law with
// high accuracy: 2007-11-27

   class Sep2{ // quantum system made of two separated subsystems
      DynDat dd1_,dd2_;
      IntActDat dd12_;
      SysLat1 x1_, x2_;
      SysLat1 v1_, v2_;
      R t_;
      R s1_,s2_,s3_;
      typedef Sep2 Type;
   public:
      CPM_IO
      CPM_NAM(Sep2)

      static X2<SysLat1, SysLat1> dot(SysLat1 const& x1,
         SysLat1 const& x2, DynDat const& dd1, DynDat const& dd2,
         IntActDat const& dd12);
      // returns the velocities determined by the arguments

      Sep2(DynDat const& dd1, DynDat const& dd2, IntActDat const& dd12,
         SysLat1 const& x1, SysLat1 const& x2, R t=0):
         dd1_(dd1),dd2_(dd2),dd12_(dd12),
         x1_(x1),x2_(x2),v1_(x1_),v2_(x2_),t_(t)
        // {dot_();}
        { X2<SysLat1,SysLat1> vv=dot(x1_,x2_,dd1_,dd2_,dd12_);
         v1_=vv.c1(); v2_=vv.c2();}

      SysLat1 x1()const{ return x1_;}
      SysLat1 x2()const{ return x2_;}
```

```
SysLat1 v1()const{ return v1_;}
SysLat1 v2()const{ return v2_;}
R t()const{ return t_;}
DynDat dd1()const{ return dd1_;}
DynDat dd2()const{ return dd2_;}
IntActDat dd12()const{ return dd12_;}

static SysLat1 vInt(SysLat1 const& y1, SysLat1 const& y2,
   DynDat const& dyn1, DynDat const& dyn2, IntActDat const& ia12);
   //: v interaction
   // primary interaction descriptor. Changing this function carries
   // over to all other interaction related functions of Sep2 and
   // also
   // further to Sep3

static R vIntScl(SysLat1 const& y1, SysLat1 const& y2,
   DynDat const& dyn1, DynDat const& dyn2, IntActDat const& ia12)
   //: v interaction scalar
{ return ((y2|vInt(y1,y2,dyn1,dyn2,ia12))/C::Ibar)[1];}

R n()const{ return x1_.absVal()*x2_.absVal();}

C hInt()const;
   //: h interaction (h for hamiltonian)

C eSepBad()const{ return C::I*((x1_|v1_)+(x2_|v2_));}
C eSep()const
   // heureka! here was the error: C::I*((x1_|v1_) is not only
   // (x1_| H_1 x_1) but it contains the contribution from the
   // other particle. Correction 2007-11-27
   // eSep should not contain the contribution of the homogeneous
   // field since this is included in hInt();
{
   SysLat1 v1(x1_),v2(x2_);
   R betaMem=DynDat::beta();
   DynDat::beta()=0;
   v1.dot_(dd1_); v2.dot_(dd2_); // v1!=v1_; v2!=v2_;
   DynDat::beta()=betaMem;
   return C::I*((x1_|v1)+(x2_|v2));
}
C eTot()const{ return eSep()+hInt();}
   // this is a (approximately) conserved quantity
R e0()const{ return eSep()[1];}
R eInt()const{ return hInt()[1];}
R_Vector stt()const
    //: statistics
{
   R z1=x1_.s(dd1_.x0()), z2=x2_.s(dd2_.x0());
   return R_Vector("",z1,z2,z1*z2);
}
```

```
      void step_(R dt);
         //: step
         // according to (8) of my paper 'The direct midpoint method
         // as a quantum mechanical integrator II.
         // works since 2007-11-21
      void stepGam_(R dt);
      void stepGamTil_(R dt); // not useful
      void mark(Graph& g, DisCon const& dc)const;
         //: mark
      V< V<R2> > run_(Graph& g, RunCon& rc, DisCon const& dc);
         //: run
    };

//////////////////////// class CM3 ////////////////////////////////////////
// classical model 3
// Only those functions are implemented which are needed
// in Sep3::run_
   class CM3{ // classical model of 3 particles
      R x1_,x2_,x3_;
      R v1_,v2_,v3_;
   public:
      CM3(DynDat const& dd1, DynDat const& dd2, DynDat const& dd3):
      x1_(dd1.x0()),x2_(dd2.x0()),x3_(dd3.x0()),
         v1_(0), v2_(0), v3_(0){}
         // construction of particles at rest, at places given by
         // the equilibrium points determined by dd1,dd2,dd3.

      static void step(R dt, R& x1, R& x2, R& x3, R& v1, R& v2, R& v3,
         DynDat const& dd1, DynDat const& dd2, DynDat const& dd3,
         IntActDat const& dd12);
         //: step
         // dynamical law as a stateless function

      R x1()const{ return x1_;}
         //: x1
      R x2()const{ return x2_;}
         //: x2
      R x3()const{ return x3_;}
         //: x3

      void step_(R dt, DynDat const& dd1, DynDat const& dd2,
         DynDat const& dd3, IntActDat const& dd12)
         //: step
         // dynamical law as a state-changing member function,
         // direct midpoint method
      {
         step(dt,x1_,x2_,x3_,v1_,v2_,v3_,dd1,dd2,dd3,dd12);
      }
   };
```

```
////////////////////////////// class Sep3 ////////////////////////////

   class Sep3{ // quantum system made of three separated subsystems
      // no extension to a trajectory class needed for defining  the
      // function run_(...) and thus to define trajectories of the system.
      DynDat dd1_,dd2_,dd3_;
      IntActDat dd12_;
      SysLat1 x1_, x2_, x3_;
      SysLat1 v1_, v2_, v3_;
      R t_;
      CM3 cs_; // classical state
      typedef Sep3 Type;
   public:
      CPM_IO
      CPM_NAM(Sep3)

      static X3<SysLat1, SysLat1, SysLat1> dot(SysLat1 const& x1,
         SysLat1 const& x2, SysLat1 const& x3,
         DynDat const& dd1, DynDat const& dd2, DynDat const& dd3,
         IntActDat const& dd12);
      // returns the velocities determined by the arguments

      Sep3(DynDat const& dd1, DynDat const& dd2, DynDat const& dd3,
         IntActDat dd12,
         SysLat1 const& x1, SysLat1 const& x2, SysLat1 const& x3,
         R t=0): dd1_(dd1),dd2_(dd2),dd3_(dd3),dd12_(dd12),
         x1_(x1),x2_(x2),x3_(x3),v1_(x1_),v2_(x2_),v3_(x3_),t_(t),
         cs_(dd1_,dd2_,dd3_)
         {
            X3<SysLat1,SysLat1,SysLat1> vv=
               dot(x1_,x2_,x3_,dd1_,dd2_,dd3_,dd12_);
            v1_=vv.c1(); v2_=vv.c2(); v3_=vv.c3();
         }

      TrjLat3 toTrlLat3()const
      //: to TrjLat3
      // usefull for comparison with exact dynamics
      {
         return TrjLat3(dd1_,dd2_,dd3_,dd12_, SysLat3(x1_,x2_,x3_),t_);
      }

      R dtSup( Z p=100, R tWait=0, R gamma=0.1)const
      //: dt supremum
      // Obtains an upper limit for a stable time step based on the
      // corresponding Hamiltonian for which there is a good theory
      // (which for separated dynamics is missing)
      {
         return toTrlLat3().dtSup(p,tWait,gamma);
      }
```

```
SysLat1 x1()const{ return x1_;}
SysLat1 x2()const{ return x2_;}
SysLat1 x3()const{ return x3_;}

SysLat1 v1()const{ return v1_;}
SysLat1 v2()const{ return v2_;}
SysLat1 v3()const{ return v3_;}

R t()const{ return t_;}

DynDat dd1()const{ return dd1_;}
DynDat dd2()const{ return dd2_;}
DynDat dd3()const{ return dd3_;}

IntActDat dd12()const{ return dd12_;}
R n()const{ return x1_.absVal()*x2_.absVal()*x3_.absVal();}

C hInt()const;
   //: h interaction (h for hamiltonian)

C eSepBad()const{ return C::I*((x1_|v1_)+(x2_|v2_)+(x3_|v3_));}
C eSep()const
{
   SysLat1 v1(x1_),v2(x2_),v3(x3_);
   R betaMem=DynDat::beta();
   DynDat::beta()=0;
   v1.dot_(dd1_); v2.dot_(dd2_); v3.dot_(dd3_);
   DynDat::beta()=betaMem;
   return C::I*((x1_|v1)+(x2_|v2)+(x3_|v3));
}

C eTot()const{ return eSep()+hInt();}
R e0()const{ return eSep()[1];}
R eInt()const{ return hInt()[1];}
R_Vector stt()const
    //: statistics
    // expectation values of the quantum particle positions
    // relative to the equilibrium points of the particles.
    // No reference to the classical particles.
{
   R z1=x1_.s(dd1_.x0()), z2=x2_.s(dd2_.x0()), z3=x3_.s(dd3_.x0());
   return R_Vector("",z1,z2,z3,z1*z2,z1*z3,z2*z3,z1*z2*z3,z1+z2+z3);
}
R_Vector sttCls()const
   //: statistics classical
{
   R z1=cs_.x1()-dd1_.x0(), z2=cs_.x2()-dd2_.x0(),
   z3=cs_.x3()-dd3_.x0();
   return R_Vector("",z1,z2,z3,z1*z2,z1*z3,z2*z3,z1*z2*z3,z1+z2+z3);
```

```
      }
      void step_(R dt);
         //: step
         // implemented as a double step of stepGam_
      void stepGam_(R dt);
         //: step gamma
         // integrator from equation (8) of my paper 'The direct midpoint
         // method as a quantum mechanical integrator II.
         // step_ and stepGam_ don't propagate cs_. This reflects the
         // status of cs_ as an experimental device for a first comparison
         // between quantum motion and classical motion.
         // However, function run_(...) does propagate cs_.
      void mark(Graph& g, DisCon const& dc)const;
         //: mark
         // presently the vertical lines indicate the equilibrium position
         // of the particle. It would be useful also to mark the
         // expected position of the quantum particle and the position
         // of the classical particle.
      x2vv run_(Graph& g, RunCon& rc, DisCon const& dc);
         //: run
         // The return values are formed only from output values of
         // function stt() and thus don't refer to classical particles.
         // Since the first two arguments are non-constant references
         // the C+- conformal name would be run_1_2_(...)
      x2vv run_(Graph& g, RecordHandler const& rc);
    };

//////////////////////////////// class Fun3 ////////////////////////////

   class Fun3{ // fundamental version of Sep3
      // Dynamics derived from TrjLat3 by means of destruction operators.
      // Sep3 is much more efficient.
      // Fun3 makes no use of the velocities and they thus are not among
      // the data members. This lets the method mis the stability of the
      // serious integrators employed in Sep3 and TrjLat3. It is only
      // by enforced re-normalization that the method allows trajectories
      // long enough for a meaningful comparison with the serious methods.
      // quantum system made of three separated subsystems
      // no extension to a trajectory class needed for defining  the
      // function run_(...) and thus to define trajectories of the system.

      DynDat dd1_,dd2_,dd3_;
      IntActDat dd12_;
      SysLat1 x1_, x2_, x3_;
      R t_;
      CM3 cs_; // classical state
         // not essential but convenient in order to let Fun3 replace
         // Sep3 in all places

      typedef Fun3 Type;
```

```
   static bool simplify_;
      // method parameter in function step_
   static R epsilon_;
      // method parameter in function step_
public:
   CPM_IO
   CPM_NAM(Fun3)


   Fun3(DynDat const& dd1, DynDat const& dd2, DynDat const& dd3,
      IntActDat dd12,
      SysLat1 const& x1, SysLat1 const& x2, SysLat1 const& x3,
      R t=0): dd1_(dd1),dd2_(dd2),dd3_(dd3),dd12_(dd12),
      x1_(x1),x2_(x2),x3_(x3),t_(t),
      cs_(dd1_,dd2_,dd3_){}

   SysLat1 x1()const{ return x1_;}
   SysLat1 x2()const{ return x2_;}
   SysLat1 x3()const{ return x3_;}

   R t()const{ return t_;}

   DynDat dd1()const{ return dd1_;}
   DynDat dd2()const{ return dd2_;}
   DynDat dd3()const{ return dd3_;}

   IntActDat dd12()const{ return dd12_;}
   R n()const{ return x1_.absVal()*x2_.absVal()*x3_.absVal();}


   C eTot()const
   { return SysLat3(x1_,x2_,x3_).eTot(dd1_,dd2_,dd3_,dd12_); }
   C eSep()const
   {
      IntActDat dd12f(dd12_);
      dd12f.free_();
      return SysLat3(x1_,x2_,x3_).eTot(dd1_,dd2_,dd3_,dd12f);
   }
   C hInt()const{ return eTot()-eSep();}
      //: h interaction (h for hamiltonian)

   R e0()const{ return eSep()[1];}
   R eInt()const{ return hInt()[1];}
   R_Vector stt()const
       //: statistics
       // expectation values of the quantum particle positions
       // relative to the equilibrium points of the particles.
       // No reference to the classical particles.
   {
      R z1=x1_.s(dd1_.x0()), z2=x2_.s(dd2_.x0()), z3=x3_.s(dd3_.x0());
```

```
         return R_Vector("",z1,z2,z3,z1*z2,z1*z3,z2*z3,z1*z2*z3,z1+z2+z3);
      }
      R_Vector sttCls()const
         //: statistics classical
      {
         R z1=cs_.x1()-dd1_.x0(), z2=cs_.x2()-dd2_.x0(),
         z3=cs_.x3()-dd3_.x0();
         return R_Vector("",z1,z2,z3,z1*z2,z1*z3,z2*z3,z1*z2*z3,z1+z2+z3);
      }
      void step_(R dt);
         //: step
         // implemented as a double step of stepGam_

      void mark(Graph& g, DisCon const& dc)const;
         //: mark
         // presently the vertical lines indicate the equilibrium position
         // of the particle. It would be useful also to mark the
         // expected position of the quantum particle and the position
         // of the classical particle.
      x2vv run_(Graph& g, RunCon& rc, DisCon const& dc);
         //: run
         // The return values are formed only from output values of
         // function stt() and thus don't refer to classical particles.
         // Since the first two arguments are non-constant references
         // the C+- conformal name would be run_1_2_(...)

      static bool& sim(){ return simplify_;}
      static R& eps(){ return epsilon_;}
    };


//////////////////////////// class Ia ////////////////////////////////////

class Ia{ // interaction descriptor for spin one half
   // Interaction descriptor for spin 1/2
      CC2 h1_;
         // one-particle interaction with external field
      CCCC2 h12_;
         // pair interaction
   public:
      Ia(){}

      Ia(R3 const& h, R q, bool full=true):
         h1_(CC2::sig1*h[1]+CC2::sig2*h[2]+CC2::sig3*h[3]),
         h12_(CCCC2::spinSpin(q,full)){}
         // basic constructor. h homogeneous field
         // q coupling constant. For full==false we have
         // only XY-coupling, else rotation invariant coupling.

      Z dim()const{ return 2;}
```

```
         //: dimension
      C2 operator()(C2 const& x)const{ return h1_*x;}
      C2 operator()(C2 const& x, C2 const& y)const
      { return h12_(x,y).trc1(x);}

      C avr(C2 const& x, C2 const& y)const
         //: average
      { return (h12_(x,y).trc1(x)|y);}

      CC2 const& h1()const{ return h1_;}
      CCCC2 const& h12()const{ return h12_;}

   };

//////////////////////////// class SpinN ////////////////////////////////

   class SpinN{ // spin lattice quantum system
      Z m_,n_;
      Ia ia_; // contains the coupling constants
      Vl<C2> vx_;
      Vl<C2> vv_;
      R t_;
      R eps_;
      R dt_,dt0Inv_;
      Z sc_; // step counter
      R rf_; // reverse factor ( has the same dynamical effect as
         // reversal of dt_ . But running verbally back in time
         // conflicts with my program control, especially when
         // automatic time step adjustment is being used.
      typedef SpinN Type;
      void ini_(){
         C et=eTot();
         cpmdebug(et);
         dt_=cpminv(et.absVal())*sqrt(R(m_*n_));
         dt0Inv_=1./dt_;
         t_=0;eps_=0;sc_=0;
      }
      void dot_();

   public:
      CPM_NAM(SpinN)

      SpinN(Z m, Z n, Ia const& ia): m_(m), n_(n),ia_(ia),
         vx_(m_*n_,C2(C(1,1),C(-1,-1))), rf_(1)
            // rf_ needs to be defined when dot_ gets called
      {
         for (Z i=vx_.b();i<=vx_.e();++i) vx_[i]=vx_[i].ranVal(i);
         dot_();
         ini_();
      }
```

```
SpinN(Z m, Vl<C2> const& vx, Ia const& ia): m_(m),
   n_(vx.dim()/m_),ia_(ia), vx_(vx), rf_(1)
{
   cpmassert(vx_.dim()==m_*n_,"SpinN(Z,Vl<C2>,Ia)");
   dot_();
   ini_();
}

SpinN(Img24 const& im, Ia const& ia):m_(im.h()),
   n_(im.w()),ia_(ia),vx_(m_*n_), rf_(1)
{
   Z k=0;
   VV<ColRef> vvc=im.toVVColRef();
   for (Z i=1;i<=vvc.dim1();++i){
      for (Z j=1;j<=vvc.dim2();++j){
         vx_[k++]=C2(Color(vvc[i][j]).toR3());
      }
   }
   dot_();
   ini_();
}

Z w()const{return n_;}
Z h()const{return m_;}
Z hw()const{return m_*n_;}

Ia const& getIa()const{ return ia_;}
   //: get Ia
R getEps()const{ return eps_;}
   //: get epsilon
void setEps_(R eps){ eps_=eps;}
   //: set epsilon
R get_dt()const{ return dt_;}
   //: get dt
void set_dt_(R dt){dt_=dt;}
   //: set dt
void set_t_(R t){t_=t;}
   //: set t
R t()const{ return t_;}
   //: t
R dtMean()const{ return t_/sc_;}
   //: dt mean
void mod_dt_(R fac){dt_*=fac; cpmdebug(dt_);}
   //: modify dt

Ia ia()const{ return ia_;}

Z parNum()const{ return vx_.dim();}
   //: particle number
```

```
Z parDim()const{ return 2;}
   //: particle dimension

Z4 neg(Z i)const;
   //: neighbors

R n()const
   //: norm
{
   R r=1;
   for (Z i=vx_.b();i<=vx_.e();++i) r*=vx_.cui(i).absVal();
   return r;
}

C hInt()const;
   //: h interaction (h for hamiltonian)

R hInt_i(Z i)const; // only needed for eTot_i
R eSep_i(Z i)const; // only needed for eTot_i
R eTot_i(Z i)const; // only needed for temperature,
   // not yet working

R theta_i(Z i)const{ return vx_[i].blcAng()[1];}
R phi_i(Z i)const{ return vx_[i].blcAng()[2];}
R2 blc_i(Z i)const{ return vx_[i].blcAng();}

R sig3()const
{
   R res=0;
   for (Z i=vx_.b();i<=vx_.e();++i){
      C2 ci=vx_.cui(i);
      res+=(ci|CC2::sig3*ci)[1];
   }
   return res/(m_*n_);
}

R3 sig(Z i)const;
   //: sigma
   // expectation values of the three Pauli matrices
   // at chain position i

Vl<C2> col()const;
   //: color

R2 blcAng()const
   //: Bloch angles
   // returns mean values of Bloch angles over the lattice
{
   R r1=0, r2=0;
```

```
      for (Z i=vx_.b();i<=vx_.e();++i){
         R2 ri=vx_.cui(i).blcAng();
         r1+=ri[1];
         r2+=ri[2];
      }
      R npInv=1./(m_*n_);
      return R2(r1*npInv,r2*npInv);
   }

   C eSep()const;
   C eTot()const{ return eSep()+hInt();}
      //: e total
   R_Vector stt()const{ return R_Vector();}

   void step_();
      //: step
      // implemented as a double step of stepGam_(h)
      // where h=dt_*0.5

   void stepGam_(R dt);
      //: step gamma
      // integrator from equation (8) of my paper 'The direct midpoint
      // method as a quantum mechanical integrator II.
      // Here the argument is active, not dt_

   void rev_()
      //: reversal
   {
      for (Z i=vv_.b();i<=vv_.e();++i) vv_[i]*=-1;
      t_=0;
      rf_*=-1;
   }

   void revDir_(){ dt_*=-1;}
      //: reverse directly

   void mark(Graph& g, DisCon const& dc)const;
      //: mark
      // The connection between C2 and color is given by function

   void run_(Graph& g, RunCon& rc, DisCon const& dc);
      //: run

   void runSim_(Graph& g, RunCon& rc, DisCon const& dc);
      //: run simply

   void runRevTest_(Graph& g, RunCon& rc, DisCon const& dc);
      //: run reversion test
      // This is actually the work hourse for the qid publication
};
```

```
//////////////////////////// class Ky ////////////////////////////////////

    class Ky{ // lean class of keys, for forming map<Ky,C>
       std::vector<bool> v_;
    public:
       explicit Ky(Z d=0): v_(cpmpos(d),false){}
         //: Ky
       bool operator<(Ky const& k)const{ return v_<k.v_;}
         //: <
       void set_(Z i, bool b){ v_[i]=b;}
         //: set
       bool get(Z i)const{ return v_[i];}
         //: get
       bool next_(){ return CpmAlgorithms::next_1(v_);}
         //: next
       void ins_(Z i, bool b){ v_.insert(v_.begin()+i,b);}
         //: insert
    };

    typedef std::map<Ky,C> Mkc; // data type for wave functions

//////////////////////////// struct QrecPar ////////////////////////////////
    struct QregPar{ // parameters needed in TrjQreg
       R h,q;
       bool full;
       QregPar(R hi, R qi, bool fi):h(hi),q(qi),full(fi){}
    };

//////////////////////////////// class Qreg ////////////////////////////////

    class Qreg{ // array of qubits
      // q-register, system made out of a finite number of
      // q-bits
      Z n_; // number of q-bits available
      Mkc m_;
      Qreg(Z n, Mkc const& m):n_(n),m_(m){}
      typedef C ScalarType;

    public:
      CPM_QM_SCALAR_PRODUCT_INTERFACE(Qreg,C)
      CPM_NORMALIZE
      CPM_NAM("Qreg")
      explicit Qreg(Z n, Z kind=0);
         //: Qreg
      // Creating the normalized ground state on a lattice of s points
      // (Fock vacuum state) for kind==1 and the zero-vector on the same
      // lattice else

      explicit Qreg(Vl<C2> const& v);
```

```
   //: Qreg

Z dim()const{ return n_;}
   //: dimension

Z parNum()const{ return n_;}
   //: particle number

Vl<C2> den()const;
   //: density

Vl<C2> col()const;
   //: color

R3 sig(Z i)const;
   //: sigma
   // expectation values of the three Pauli matrices
   // at chain position i

C get(Ky const& k)const
    //: get
{
  Mkc::const_iterator i;
  i=m_.find(k);
  return i!=m_.end() ? i->second : C();
}

void set_(Ky const& k, C z){ m_[k]=z;}
   //: set
   // notice that mutating functions end in _ by convention

void add_(Ky const& k, C z){ m_[k]+=z;}
   //: add
void sub_(Ky const& k, C z){ m_[k]-=z;}
   //: subtract
void mul_(Ky const& k, R r){ m_[k]*=r;}
   //: multiply
void mul_(Ky const& k, C z){ m_[k]*=z;}
   //: multiply
Qreg operator+(Qreg const& q)const;
   //: +
Qreg& operator+=(Qreg const& q);
   //: +=
Qreg& operator-=(Qreg const& q);
   //: -=

Qreg operator*(R r)const;
   //: *
Qreg& operator*=(R r);
   //: *=
```

```
      Qreg& operator*=(C z);
         //: *=

      Qreg app(CC2 const& a, Z i)const;
         //: apply
         // changes *this into the state resulting from it by
         // application of a 'single particle operator' to the
         // subsystem (not subspace!) i

      Qreg app(CCCC2 const& b, Z i, Z j)const;
         //: apply
         // changes *this into the state resulting from it by
         // application of a 'particle pair operator' b
         // on the subsystem {i,j}

      Qreg ham(CC2 const& h0, CCCC2 const& hi)const;
         // : Hamiltonian
         // exact spin chain Hamiltonian
         // returns a Qreg state which results from application of
         // the spin chain Hamiltonian to Qreg state *this.
         // Explanation of parameters ....

      C hSep(CC2 const& h0)const
          //: h separated
       { return (*this|ham(h0,CCCC2()));}

      R dtSup(CC2 const& h0, CCCC2 const& hi, Z p=100, R tWait=-1)const;
         //: dt supremum

      R mag()const
         //: magnetization
         // expectation value of sigma 3
      { return (operator|(ham(CC2::sig3,CCCC2())))[1];}

      R_Vector stt()const;
          //: statistics

      void mark(Graph& g, DisCon const& dc)const;
         //: mark
    };

//////////////////////////// class TrjQreg ////////////////////////////////

   class TrjQreg{ // Trajectory of a Qreg system
      QregPar p_;
      CC2 h0_;
      CCCC2 hi_;
      Qreg x_;
      Qreg v_;
```

```
      R t_;
    public:
      CPM_NAM(TrjQreg)
      TrjQreg(QregPar const& p, Qreg const& x):p_(p),h0_(CC2::sig3*p_.h),
         hi_(CCCC2::spinSpin(p_.q, p_.full)),x_(x),
         v_(x_.ham(h0_,hi_)),t_(0){v_*=C::Ibar;}
         // version in which the velocity is set from the state
      void step_(R dt);
      R t()const{ return t_;}
      R n()const{ return x_.absVal();}
      C eTot()const{ return C::I*(x_|v_);}
      C hInt()const{ return eTot()-eSep();}
      C eSep()const{ return x_.hSep(h0_);}
      R e0()const{ return eSep()[1];}
      R eInt()const{ return hInt()[1];}
      Qreg x()const{ return x_;}
      Qreg v()const{ return v_;}
      R dtSup(Z p=100, R tWait=-1)const
         { return x_.dtSup(h0_,hi_,p,tWait);}
         //: dt supremum
      R_Vector stt()const{ return x_.stt();}
          //: statistics
      V< V<R2> > run_(Graph& g, RunCon& rc, DisCon const& dc);
         //: run
      void mark(Graph& g, DisCon const& dc)const{ x_.mark(g,dc);}
   };

   void testEC(Z d1, Z d2, Z d3);
      // experimental test of a property that should ensure
      // energy conservation. Three particle states

   void testEC4(Z d1, Z d2, Z d3, Z d4);
      // experimental test of a property that should ensure
      // energy conservation. Four-particle states.
      // This should be general enough to be convincing
      // for arbitrary particle number.

} // namespace

#endif
```

149 cpmqm.cpp

```
//? cpmqm.cpp
//? Status of work 2008-10-25.
//?
/*
   cpmqm.cpp
   History: Started 98-10-12

*/

#include <cpmqm.h>
#include <cpmvectormore.h>
#include <cpmobservable.h>
#include <cpmdistribution.h>
#include <cpmmultiimage.h>
#include <cpmframes.h>

using CpmQM::Lat1;
using CpmQM::RunCon;
using CpmQM::DisCon;
using CpmQM::DynDat;
using CpmQM::SysLat1;
using CpmQM::TrjLat1;
using CpmQM::IntActDat;
using CpmQM::SysLat2;
using CpmQM::SysLat3;
using CpmQM::TrjLat2;
using CpmQM::TrjLat3;
using CpmQM::Sep2;
using CpmQM::Sep3;
using CpmQM::Fun3;
using CpmQM::CM3;
using CpmQM::SpinN;
using CpmQM::Ia;
using CpmQM::Ky;
using CpmQM::Qreg;
using CpmQM::QregPar;
using CpmQM::TrjQreg;
using CpmQM::testEC;
using CpmQM::x2vv;

using CpmArrays::VV;
using CpmArrays::VVV;
using CpmArrays::VVVV;
using CpmArrays::R_Vector;
using CpmArrays::DisR;
using CpmLinAlg::CC2;
using CpmLinAlg::CCCC2;
```

```
using CpmImaging::Color;
using CpmImaging::WHITE;
using CpmImaging::BLACK;
using CpmImaging::LIGHTGREEN;
using CpmImaging::ImgR;


///////////////// class Lat1 ////////////////////////////////////////////

bool Lat1::prnOn(ostream& str)const
{
   cpmwt("Lat1");
   cpmp(iv_);
   cpmp(d_);
   return true;
}

bool Lat1::scanFrom(istream& str)
{
   cpms(iv_);
   cpms(d_);
   ini_();
   return true;
}

Z Lat1::com(Lat1 const& lat)const
{
   if (iv_<lat.iv_) return 1;
   if (iv_>lat.iv_) return -1;
   if (d_<lat.d_) return 1;
   if (d_>lat.d_) return -1;
   return 0;
}

//////////////////////////// class DynDat ////////////////////////////
R DynDat::beta_=0;

// CPM_IO

bool DynDat::prnOn(ostream& str)const
{
   cpmwt("DynDat");
   cpmp(m_);
   cpmp(q_);
   cpmp(lam_);
   cpmp(x0_);
   cpmp(p_);
   cpmp(out_);
   return true;
}
```

```
bool DynDat::scanFrom(istream& str)
{
   cpms(m_);
   cpms(q_);
   cpms(lam_);
   cpms(x0_);
   cpms(p_);
   cpms(out_);
   return true;
}

//////////////////////// struct RunCon ////////////////////////////////////
RunCon::RunCon(RecordHandler const& rch, Word const& sec):imgSteps(0),
fromDisk(false),reversed(true),tWaitDocu2(0), tWaitDocu3(0)
{
   cpmrh(dt);
   cpmrhf(nIter);// needs not to be there
   cpmrhf(tWaitIter);// needs not to be there
   cpmrhf(gammaIter); // needs not to be there
   cpmrh(nSteps);
   cpmrh(viewPeriod);
   cpmrh(docuPeriod);
   cpmrhf(imgPeriod); // needs not to be there
   cpmrh(tWaitIni);
   cpmrh(tWaitView);
   cpmrh(tWaitDocu);
   cpmrhf(tWaitDocu2); // needs not to be there
   cpmrhf(tWaitDocu3); // needs not to be there
   cpmrh(tWaitRes);
   cpmrhf(reversed); // needs not to be there
   cpmrhf(fromDisk); // needs not to be there
   cpmrh(sel);
   cpmrhf(imgSteps); // needs not to be there
}

////////////////////////////// class SysLat1 ////////////////////////////
// CPM_IO

bool SysLat1::prnOn(ostream& str)const
{
   cpmwt("SysLat1");
   cpmp(lat_);
   cpmp(rep_);
   return true;
}

bool SysLat1::scanFrom(istream& str)
{
   cpms(lat_);
```

```
   cpms(rep_);
   return true;
}

// construction

namespace{
   R potAux(R const& x, DynDat const& dd, Lat1 const& lat)
   {
      R xs=x+lat.iv().mean(); // shifted
      return dd.pot(xs);
   }
      // the involved argument shift is needed, since
      // QMBase understands the biotope always with the
      // origin at the center
}

SysLat1::SysLat1(Lat1 const& lat, DynDat const& dd, Z n):lat_(lat)
{
   Z mL=2;
   Word loc("SysLat1(Lat1,DynDat,Z)");
   CPM_MA
   using namespace CpmQuantumMechanics;
   QMBase::setMass(dd.m());
   QMBase::setSize(lat.size());
   Z d=lat_.d();
   Observable T1("eKinFd",d);
   R_Func pot=F2<R,DynDat,Lat1,R>(dd,lat_)(potAux);
   Observable V1("x",d,pot);
   Observable H1=T1+V1;
   SpectralObject so;
   {  // encapsulating the heavy initialization of so
      bool enforceSymmetry=true;
         // our H2 is known to be symmetric
      bool test=false;
         // true would consume a lot of time but worked in my test cases
         // perfectly
      Z iterMax=128;
      R eps0=0;
      R eps1=1e-7,eps2=1e-6; // test values from the past
      so=H1.spectralRepresentation(enforceSymmetry,
      test,iterMax,eps0,eps1,eps2); // this is a big beast
      so=so.order();
   }
   rep_=so[n].getRep();
}

#define CPM_SC\
   Z d=lat_.d();\
   rep_=Vl<C>(d);\
```

```
   R xi=lat_.fir();\
   for (Z i=0;i<d;++i){\
      rep_.cui(i)=C(f(xi));\
      lat_.next_1(xi);\
   }\
   nor_();

void SysLat1::ini_(F<R,C> const& f)
{
   CPM_SC
}

void SysLat1::ini_(F<R,R> const& f)
{
   CPM_SC
}

#undef CPM_SC

// arithmetics

namespace{
   void accFunc(C& z, C const& zAdd){ z+=zAdd;}
   C conjFunc(C const& z){ return z.con();}
   C negFunc(C const& z1){  return -z1;}
   C sumFunc(C const& z1, C const& z2){ return z1+z2;}
   C diffFunc(C const& z1, C const& z2){ return z1-z2;}
   C prodFunc(C const& z1, C const& z2){ return z1*z2;}
   C prodFuncConj1(C const& z1, C const& z2){return z1.con()*z2; }
   C prodFuncConj2(C const& z1, C const& z2){ return z1*z2.con();}
   C prodFuncReal(C const& z1, C const& z2){ return z1*z2.real();}
}

C SysLat1::operator |(SysLat1 const& x2)const
{
   return rep_.fAcc2(prodFuncConj1,accFunc,x2.rep_);
}

SysLat1 SysLat1::con(void)const
{
   Vl<C> v=rep_.fa(conjFunc);
   return SysLat1(lat_.iv(),v);
}

SysLat1 SysLat1::operator - (void)const
{
   Vl<C> v=rep_.fa(negFunc);
   return SysLat1(lat_.iv(),v);
}
```

```
namespace{
   C fpFunc(C const& z, R const& p)
   {
      return p==2 ? C(z.absSqr()) : C(::pow(z.absVal(),p));
   }
}

SysLat1& SysLat1::makeReal_(R p)
{
   F<C,C> fp=F1<C,R,C>(p)(fpFunc);
   rep_.fa_(fp,rep_.dom());
   return *this;
}

SysLat1& SysLat1::operator +=(SysLat1 const& x)
{
   rep_.ff_(sumFunc,x.rep_);
   return *this;
}

SysLat1& SysLat1::operator -=(SysLat1 const& x)
{
   rep_.ff_(diffFunc,x.rep_);
   return *this;
}

SysLat1& SysLat1::operator *=(C z)
{
   rep_.fc_(prodFunc,z);
   return *this;
}

SysLat1& SysLat1::operator *=(R r)
{
   rep_.fc_(prodFuncReal,C(r));
   return *this;
}

SysLat1 SysLat1::operator +(SysLat1 const& x)const
{
  SysLat1 res(*this);
  return res+=x;
}

SysLat1 SysLat1::operator -(SysLat1 const& x)const
{
   SysLat1 res(*this);
   return res-=x;
}
```

```
SysLat1 SysLat1::operator *(C x2)const
{
  SysLat1 res(*this);
  return res*=x2;
}

SysLat1 SysLat1::operator *(R x2)const
{
  SysLat1 res(*this);
  return res*=x2;
}

R SysLat1::nor_()
{
   R n=norm();
   if (n>0){
      R nInv=1./n;
      operator*=(nInv);
   }
   return n;
}

// graphics

void SysLat1::mark(Graph& g, DisCon const& dc)const
{
   using CpmLinAlg::R2;
   using CpmArrays::V;
   using namespace CpmImaging;
   V< V<R2> > dat(3);
   Z d=lat_.d();
   R xi=lat_.fir();
   V<R2> re(d);
   V<R2> im(d);
   V<R2> ab(d);
   for (Z i=0;i<d;++i){
      C zi=rep_.cui(i);
      R2 rei(xi,zi.real());
      R2 imi(xi,zi.imag());
      R2 abi(xi,zi.absVal());
      Z ip=i+1;
      re.cui(ip)=rei;
      im.cui(ip)=imi;
      ab.cui(ip)=abi;
      lat_.next_1(xi);
   }
   dat[1]=re;
   dat[2]=im;
   dat[3]=ab;
   V<Color> cl(3);
```

```
   cl[1]=RED;  // color for real part
   cl[2]=GREEN;   // color for imaginary part
   cl[3]=BLACK; // color for absolute value
   g.setAutoScl_(dc.autoScl);
   g.mark(dat,cl,dc.secFac);
}
// dynamics

void SysLat1::dot_(DynDat const& dd)
// mutating function to avoid copying of state data
{
   Z d=lat_.d();
   if (d<2) cpmerror("SysLat1::dot_(DynDat): d<2");

   R ds=lat_.ds();
   R fr=2*dd.m()*ds*ds;
   if (fr==0.) cpmerror("SysLat1::dot_(DynDat): fr==0");

   fr=1./fr;
   C f1=fr*C::I;
   C f2=-C::I;
   Vl<C> mem=rep_;
   R xi=lat_.fir();
   Z dm=d-1;
// handling of boundary conditions
   CpmArrays::Outside out=dd.out();
   C valLeft=mem(-1,out);
   C valRight= mem(d,out);
// first term of the loop
   rep_.cui(0)=(valLeft - mem.cui(0)*2 + mem.cui(1))*f1 +
      mem.cui(0)*f2*dd.pot(xi);
   lat_.next_1(xi); // shifts xi to the next position
// core terms of the loop
   for (Z i=1;i<dm;++i){ // part not affected by boundary conditions
      rep_.cui(i)=(mem.cui(i-1) - mem.cui(i)*2 + mem.cui(i+1))*f1 +
         mem.cui(i)*f2*dd.pot(xi);
      lat_.next_1(xi);
   }
// last term of the loop
// xi is now at the last position
   rep_.cui(dm)=(mem.cui(dm-1) - mem.cui(dm)*2 + valRight)*f1 +
      mem.cui(dm)*f2*dd.pot(xi);
      // up to positve constants we have psi^dot= -i H psi =
      // -i (T + V) psi =
      // (-i) T psi - i V psi = (-i) -psi" - i V psi
      // = i psi" - i V psi . In finite differences:
      // psi" = psi1 -2 psi0 +psi(-1).
 }

R SysLat1::enrHomFil(DynDat const& dd)const
```

```
{
   R b=dd.q()*DynDat::beta();
   R res=0;
   R xi=lat_.fir();
   for (Z i=rep_.b();i<=rep_.e();++i){
      res+=rep_.cui(i).absSqr()*b*xi;
      lat_.next_1(xi); // shifts xi to the next position
   }
   return res;
 }

R SysLat1::dtSup(DynDat const& dd, Z p, R tWait)const
{
   Z mL=1;
   Word loc("SysLat1::dtSup(Z)");
   CPM_MA
   cpmassert(p>0,loc);
   SysLat1 psi=*this;
   psi.nor_();
   R_Vector mem(p);
   Graph gr;
   R pInv=1./p;
   for (Z i=1;i<=p;++i){
      psi.dot_(dd);
      R mi=psi.nor_(); // the value is the norm!
      mem[i]=mi; // the value is the norm!
      cpmvalue("enr",mi,2);
      if (tWait>=0){
         psi.mark(gr,DisCon());
         gr.display();
         cpmwait(tWait);
      }
      if (i==1){
         R n1=mem[i];
         cpmdebug(n1);
      }
      cpmprogress("x",i*pInv,3);
   }
   R_Vector stat=CpmArrays::statistics(mem);
   R sigma=stat[3];
   R maxVal=stat[10];
   cpmdebug(maxVal);
   R epsMax=maxVal+sigma;
   cpmdebug(epsMax);
   cpmvalue("eMax",epsMax,3);
   R res=2./epsMax;
   CPM_MZ
   return res;
}
```

```
//////////////////////////// class TrjLat1 ////////////////////////////////

C TrjLat1::hInt()const
{
   Lat1 lat=x_.lat();
   Z d=lat.d();
   SysLat1 ax(x_);
   ax.makeReal_();
   C res;
   for (Z i=0;i<d;++i){
      R xi=lat.pos(i);
      res+= ax.cui(i)*dd_.pot(xi);
   }
   return res;
}

#define CPM_SC_RUN\
   using CpmArrays::R_Matrix;\
   using CpmArrays::R_Vector;\
   using std::vector;\
   g.setSclMem_(false);\
   R n0=n();\
   R n0Inv=1./n0;\
   C e0=eTot();\
   R eTotR=e0[1];\
   C ei0=hInt();\
   R eInt=ei0[1];\
   cpmvalues2("eTot,eInt",eTotR,eInt,4);\
   cpmdata<<endl<<"eTot="<<e0<<" eInt="<<ei0<<endl;\
   R e0Inv=cpminv(e0.absVal());\
   R e01=e0[1];\
   R e02=e0[2];\
   R dt=rc.dt;\
   Z nSteps=rc.nSteps;\
   R nInv=1./nSteps;\
   R tWaitView=rc.tWaitView;\
   R tWaitDocu=rc.tWaitDocu;\
   R tWaitRes=rc.tWaitRes;\
   CpmAlgorithms::CyclicAlarm img(rc.imgPeriod);\
   CpmAlgorithms::CyclicAlarm view(rc.viewPeriod);\
   CpmAlgorithms::CyclicAlarm docu(rc.docuPeriod);\
   if (rc.tWaitIni>0){\
      mark(g,dc);\
      g.setText("initial state");\
      g.dis();\
      cpmwait(rc.tWaitIni,2);\
   }\
   g.clr_();\
   vector<R> tv(1,0.), nv(1,0.), e1v(1,0.), e2v(1,0.);
```

```
#define CPM_SC_RUN2\
   R ti=t();\
   R ni=n();\
   C ei=eTot();\
   tv.push_back(ti);\
   nv.push_back((ni-n0)*n0Inv);\
   e1v.push_back((ei[1]-e01)*e0Inv);\
   e2v.push_back((ei[2]-e02)*e0Inv);

#define CPM_SC_RUN3\
   R_Matrix doc(4);\
   doc[1]=R_Vector(tv);\
   doc[2]=R_Vector(nv);\
   doc[3]=R_Vector(e1v);\
   doc[4]=R_Vector(e2v);\
   g.clr_();\
   g.setAutoScl_(true);

V< V<R2> > TrjLat1::run_(Graph& g, RunCon& rc, DisCon const& dc)
{
   CPM_SC_RUN
   if (dt<0) dt=-dt; // negative input allows us to use values without
      // energy dependent rescaling
   else{
      R dtMax=x_.dtSup(dd_,rc.nIter,rc.tWaitIter);
      dt*=dtMax;
   }
   vector<R2> s1v;
   for (Z i=1;i<=nSteps;++i){
      step_(dt);
      cpmprogress("run",nInv*i);
      if (tWaitDocu>0){
         if (docu()){
            CPM_SC_RUN2
            R si=s();
            s1v.push_back(R2(ti,si));
         }
      }
      if (view()){
         mark(g,dc);
         g.dis();
         cpmwait(rc.tWaitView);
      }
   }
   if (tWaitDocu>0){
      CPM_SC_RUN3
      g.setText("TrjLat1: run documentation");
      g.show(doc);
      cpmwait(tWaitDocu,2);
   }
```

```
   V< V<R2> > res(1);
   res[1]=V<R2>(s1v);
   if (tWaitRes>0){
      g.clr_();
      g.setText("TrjLat1: res documentation");
      g.mark(res);
      g.dis();
      cpmwait(tWaitRes,2);
   }
   return res;
}

//////////////////////////// class IntActDat ////////////////////////////
// CPM_IO

bool IntActDat::prnOn(ostream& str)const
{
   cpmwt("IntActDat");
   cpmp(alpha_);
   cpmp(mu_);
   cpmp(eps_);
   cpmp(p_);
   return true;
}

bool IntActDat::scanFrom(istream& str)
{
   cpms(alpha_);
   cpms(mu_);
   cpms(eps_);
   cpms(p_);
   return true;
}

//////////////////////// class SysLat2 ////////////////////////////////
// CPM_IO

bool SysLat2::prnOn(ostream& str)const
{
   cpmwt("SysLat2");
   cpmp(lat1_);
   cpmp(lat2_);
   cpmp(rep_);
   return true;
}

bool SysLat2::scanFrom(istream& str)
{
   cpms(lat1_);
   cpms(lat2_);
```

```
   cpms(rep_);
   return true;
}

// construction

SysLat2::SysLat2(SysLat1 const& s1, SysLat1 const& s2):
   lat1_(s1.lat()), lat2_(s2.lat()),rep_(lat1_.d(),Vl<C>(lat2_.d()))
{
   Z d1=lat1_.d(),d2=lat2_.d();
   for (Z i=0;i<d1;++i){
      C si=s1.cui(i);
      for (Z j=0;j<d2;++j){
         C sij=si*s2.cui(j);
         rep_.cui(i).cui(j)=sij;
      }
   }
}

SysLat2& SysLat2::operator *=(C z)
{
   Z d1=lat1_.d();
   for (Z i=0;i<d1;++i){
      rep_.cui(i).fc_(prodFunc,z);
   }
   return *this;
}

SysLat2& SysLat2::operator *=(R r)
{
   Z d1=lat1_.d();
   for (Z i=0;i<d1;++i){
      rep_.cui(i).fc_(prodFuncReal,C(r));
   }
   return *this;
}

SysLat2& SysLat2::operator +=(const SysLat2& x)
{
   Z d1=lat1_.d();
   for (Z i=0;i<d1;i++){
      rep_.cui(i).ff_(sumFunc,x.rep_.cui(i));
   }
   return *this;
}

SysLat2& SysLat2::operator -=(const SysLat2& x)
{
   Z d1=lat1_.d();
   for (Z i=0;i<d1;i++){
```

```
      rep_.cui(i).ff_(diffFunc,x.rep_.cui(i));
   }
   return *this;
}

C SysLat2::operator |(SysLat2 const& x)const
{
   C res;
   Z d1=lat1_.d();
   for (Z i=0;i<d1;i++){
      res+=rep_.cui(i).fAcc2(prodFuncConj1, accFunc, x.rep_.cui(i));
   }
   return res;
}

SysLat2 SysLat2::operator +(SysLat2 const& x)const
{
  SysLat2 res(*this);
  return res+=x;
}

SysLat2 SysLat2::operator -(SysLat2 const& x)const
{
   SysLat2 res(*this);
   return res-=x;
}

SysLat2 SysLat2::operator *(C x2)const
{
  SysLat2 res(*this);
  return res*=x2;
}

SysLat2 SysLat2::operator *(R x2)const
{
  SysLat2 res(*this);
  return res*=x2;
}

R SysLat2::nor_()
{
   R n=norm();
   if (n>0){
      R nInv=1./n;
      operator*=(nInv);
   }
   return n;
}

void SysLat2::dot_(DynDat const& dd1, DynDat const& dd2,
```

```
         IntActDat const& dd12)
{
   Z mL=2;
   Word loc("SysLat2::dot_(DynDat,DynDat,IntActDat)");
   CPM_MA
   R m1=dd1.m(), m2=dd2.m();
   R q1=dd1.q(),q2=dd2.q();
   R q12=q1*q2;
   R ds1=lat1_.ds(), ds2=lat2_.ds();
   R f1=1./ds1;
   f1*=(-f1);
   R f2=1./ds2;
   f2*=(-f2);
   R fac1=f1/(m1*2);
   R fac2=f2/(m2*2);
   R facs=(fac1+fac2)*(-2);
   C fac=-C::I;
   Z d1=lat1_.d(), d2=lat2_.d();
   Outside o1=dd1.out(), o2=dd2.out();
   Vl< Vl<C> > vv=rep_;
   for (Z i=0;i<d1;++i){
      R xi=lat1_.pos(i);
      R v1=dd1.pot(xi);
      for (Z j=0;j<d2;++j){
         R xj=lat2_.pos(j);
         C vvij=vv.cui(i).cui(j);
         C cij=vvij*facs+
         (vv.read(i-1,o1).read(j,o2)+vv.read(i+1,o1).read(j,o2))*fac1+
         (vv.cui(i).read(j-1,o2)+vv.cui(i).read(j+1,o2))*fac2;
         R rij=dd12.dis(xi,xj);
         R v12=q12*dd12.pot(rij);
         R v2=dd2.pot(xj);
         C vij=vvij*(v1+v12+v2);
         cij+=vij;
         cij*=fac;
         rep_.cui(i).cui(j)=cij;
      }
   }
   CPM_MZ
}

SysLat1 SysLat2::red1(SysLat1 const& s)const
{
   Z mL=2;
   Word loc("SysLat2::red1(SysLat1)");
   CPM_MA
   Z d1=dim1();
   Z d2=dim2();
   cpmassert(s.dim()==d1,loc);
   SysLat1 res(lat2_);
```

```
   for (Z j=0;j<d2;++j){
      C cj;
      for (Z i=0;i<d1;++i){
         C si=s.cui(i).con();
         cj+=si*rep_.cui(i).cui(j);
      }
      res.cui(j)=cj;
   }
   CPM_MZ
   return res;
}

SysLat1 SysLat2::red2(SysLat1 const& s)const
{
   Z mL=2;
   Word loc("SysLat2::red2(SysLat1)");
   CPM_MA
   Z d1=dim1();
   Z d2=dim2();
   cpmassert(s.dim()==d2,loc);
   SysLat1 res(lat1_);
   for (Z i=0;i<d1;++i){
      C ci;
      for (Z j=0;j<d2;++j){
         C sj=s.cui(j).con();
         ci+=sj*rep_.cui(i).cui(j);
      }
      res.cui(i)=ci;
   }
   CPM_MZ
   return res;
}

R SysLat2::dtSup(DynDat const& dd1, DynDat const& dd2,
         IntActDat const& dd12, Z p, R tWait, R gamma)const
{
   Z mL=1;
   Word loc("SysLat2::dtSup(DynDat,DynDat,IntActDat,Z,R,R)");
   CPM_MA
   cpmassert(p>0,loc);
   SysLat2 psi=*this;
   psi.nor_();
   R_Vector mem(p);
   Graph gr;
   R pInv=1./p;
   for (Z i=1;i<=p;++i){
      psi.dot_(dd1,dd2,dd12);
      R mi=psi.nor_(); // the value is the norm!
      mem[i]=mi; // the value is the norm!
      cpmvalue("enr",mi,2);
```

```
      if (tWait>0){
         psi.mark(gr,0,gamma);
         gr.dis();
         cpmwait(tWait);
      }
      if (i==1){
         R n1=mem[i];
         cpmdebug(n1);
      }
      cpmprogress("x",i*pInv,3);
   }
   R_Vector stat=CpmArrays::statistics(mem);
   R sigma=stat[3];
   R maxVal=stat[10];
   cpmdebug(maxVal);
   R epsMax=maxVal+sigma;
   cpmdebug(epsMax);
   cpmvalue("eMax",epsMax,3);
   R res=2./epsMax;
   CPM_MZ
   return res;
}

R_Vector SysLat2::stt(DynDat const& dd1, DynDat const& dd2)const
{
   Z mL=2;
   Word loc("SysLat2::stt(DynDat)");
   CPM_MA
   Z d1=dim1();
   Z d2=dim2();
   R res1=0;
   R res2=0;
   R res12=0;
   for (Z i=0;i<d1;++i){
      R s1=lat1_.pos(i)-dd1.x0();
      for (Z j=0;j<d2;++j){
         R s2=lat2_.pos(j)-dd2.x0();
         R wij=rep_.cui(i).cui(j).absSqr();
         res1+=wij*s1;
         res2+=wij*s2;
         res12+=wij*s1*s2;
      }
   }
   CPM_MZ
   return R_Vector("",res1,res2,res12);
}

void SysLat2::mark(Graph& g, Z ipolMeth, R gamma, R secFac)const
{
   CpmImaging::Color::setGamma(gamma);
```

```
   bool aspRatFromState=false;
   bool periodic=true;
   bool matrixstyle=true;
   g.mark(rep_,Iv(),ipolMeth,aspRatFromState,periodic,secFac,matrixstyle);

}
//////////////////// class SysLat3 ////////////////////////////////////
// CPM_IO

bool SysLat3::prnOn(ostream& str)const
{
   cpmwt("SysLat3");
   cpmp(lat1_);
   cpmp(lat2_);
   cpmp(lat3_);
   cpmp(rep_);
   return true;
}

bool SysLat3::scanFrom(istream& str)
{
   cpms(lat1_);
   cpms(lat2_);
   cpms(lat3_);
   cpms(rep_);
   return true;
}

// construction

SysLat3::SysLat3(SysLat1 const& s1, SysLat1 const& s2, SysLat1 const& s3):
   lat1_(s1.lat()), lat2_(s2.lat()), lat3_(s3.lat())
{
   Z d1=lat1_.d(),d2=lat2_.d(),d3=lat3_.d();
   Vl<C> rep3(d3);
   Vl< Vl<C> > rep23(d2, rep3);
   rep_=Vl< Vl < Vl <C> > >(d1,rep23);
   for (Z i=0;i<d1;++i){
      C si=s1.cui(i);
      for (Z j=0;j<d2;++j){
         C sij=si*s2.cui(j);
         for (Z k=0;k<d3;++k){
            C sijk=sij*s3.cui(k);
            rep_.cui(i).cui(j).cui(k)=sijk;
         }
      }
   }
}

SysLat3& SysLat3::operator *=(C z)
```

```
{
   Z d1=lat1_.d();
   Z d2=lat2_.d();
   Z d3=lat3_.d();
   for (Z i=0;i<d1;++i){
      for (Z j=0;j<d2;++j){
         for (Z k=0;k<d3;++k){
            rep_.cui(i).cui(j).cui(k)*=z;
         }
      }
   }
   return *this;
}

SysLat3& SysLat3::operator *=(R r)
{
   Z d1=lat1_.d();
   Z d2=lat2_.d();
   Z d3=lat3_.d();
   for (Z i=0;i<d1;++i){
      for (Z j=0;j<d2;++j){
         for (Z k=0;k<d3;++k){
            rep_.cui(i).cui(j).cui(k)*=r;
         }
      }
   }
   return *this;
}

SysLat3& SysLat3::operator +=(const SysLat3& x)
{
   Z d1=lat1_.d();
   Z d2=lat2_.d();
   Z d3=lat3_.d();
   for (Z i=0;i<d1;++i){
      for (Z j=0;j<d2;++j){
         for (Z k=0;k<d3;++k){
            rep_.cui(i).cui(j).cui(k)+=x.rep_.cui(i).cui(j).cui(k);
         }
      }
   }
   return *this;
}

SysLat3& SysLat3::operator -=(const SysLat3& x)
{
   Z d1=lat1_.d();
   Z d2=lat2_.d();
   Z d3=lat3_.d();
   for (Z i=0;i<d1;++i){
```

```
      for (Z j=0;j<d2;++j){
         for (Z k=0;k<d3;++k){
            rep_.cui(i).cui(j).cui(k)-=x.rep_.cui(i).cui(j).cui(k);
         }
      }
   }
   return *this;
}

C SysLat3::operator |(SysLat3 const& x)const
{
   C res;
   Z d1=lat1_.d();
   Z d2=lat2_.d();
   Z d3=lat3_.d();
   for (Z i=0;i<d1;++i){
      for (Z j=0;j<d2;++j){
         for (Z k=0;k<d3;++k){
            res+=rep_.cui(i).cui(j).cui(k).con()*x.rep_.cui(i).cui(j\
            ).cui\
            (k);
         }
      }
   }
   return res;
}

SysLat3 SysLat3::operator +(SysLat3 const& x)const
{
  SysLat3 res(*this);
  return res+=x;
}

SysLat3 SysLat3::operator -(SysLat3 const& x)const
{
   SysLat3 res(*this);
   return res-=x;
}

SysLat3 SysLat3::operator *(C x2)const
{
  SysLat3 res(*this);
  return res*=x2;
}

SysLat3 SysLat3::operator *(R x2)const
{
  SysLat3 res(*this);
  return res*=x2;
}
```

```
R SysLat3::nor_()
{
   R n=norm();
   if (n>0){
      R nInv=1./n;
      operator*=(nInv);
   }
   return n;
}

void SysLat3::dot_(DynDat const& dd1, DynDat const& dd2,
   DynDat const& dd3, IntActDat const& dd12)
{
   Z mL=2;
   Word loc("SysLat3::dot_(DynDat,DynDat,DynDat,IntActDat)");
   CPM_MA
   R m1=dd1.m(), m2=dd2.m(), m3=dd3.m();
   R q1=dd1.q(), q2=dd2.q(), q3=dd3.q();
   R ds1=lat1_.ds(), ds2=lat2_.ds(), ds3=lat3_.ds();

   R f1=1./ds1;
   f1*=(-f1);
   R f2=1./ds2;
   f2*=(-f2);
   R f3=1./ds3;
   f3*=(-f3);
   R fac1=f1/(m1*2);
   R fac2=f2/(m2*2);
   R fac3=f3/(m3*2);

   R facs=(fac1+fac2+fac3)*(-2);
   C fac=-C::I;
   Z d1=lat1_.d();
   Z d2=lat2_.d();
   Z d3=lat3_.d();
   Outside o1=dd1.out();
   Outside o2=dd2.out();
   Outside o3=dd3.out();
   Vl< Vl< Vl<C> > > vv=rep_;
   for (Z i=0;i<d1;++i){
      R xi=lat1_.pos(i);
      R v1=dd1.pot(xi);
      for (Z j=0;j<d2;++j){
         R xj=lat2_.pos(j);
         R v2=dd2.pot(xj);
         R rij=dd12.dis(xi,xj);
         R pij=q1*q2*dd12.pot(rij);
         for (Z k=0;k<d3;++k){
            R xk=lat3_.pos(k);
```

```
            R v3=dd3.pot(xk);
            R rik=dd12.dis(xi,xk);
            R pik=q1*q3*dd12.pot(rik);
            R rjk=dd12.dis(xj,xk);
            R pjk=q2*q3*dd12.pot(rjk);
            C vvijk=vv.cui(i).cui(j).cui(k);
            C cijk=vvijk*facs
               +(vv.read(i-1,o1).read(j,o2).read(k,o3)
                  +vv.read(i+1,o1).read(j,o2).read(k,o3))*fac1
               +(vv.cui(i).read(j-1,o2).read(k,o3)
                  +vv.cui(i).read(j+1,o2).read(k,o3))*fac2
               +(vv.cui(i).cui(j).read(k-1,o3)
                  +vv.cui(i).cui(j).read(k+1,o3))*fac3;
            C vijk=vvijk*(v1+v2+v3+pij+pik+pjk);
               // here was v2+v2 instead of v2+v3 till 2008-02-11, had bad
               // effects
            cijk+=vijk;
            cijk*=fac;
            rep_.cui(i).cui(j).cui(k)=cijk;
         }
      }
   }
   CPM_MZ
}

SysLat2 SysLat3::red3(SysLat1 const& s)const
{
   Z mL=2;
   Word loc("SysLat3::red1(SysLat1)");
   CPM_MA
   Z d1=dim1();
   Z d2=dim2();
   Z d3=dim3();
   cpmassert(s.dim()==d3,loc);
   SysLat2 res(lat1_,lat2_);
   for (Z i=0;i<d1;++i){
      for (Z j=0;j<d2;++j){
         C ck;
         for (Z k=0;k<d3;++k){
            C sk=s.cui(k).con();
            ck+=sk*rep_.cui(i).cui(j).cui(k);
         }
         res.cui(i).cui(j)=ck;
      }
   }
   CPM_MZ
   return res;
}

SysLat1 SysLat3::red12(SysLat1 const& s1, SysLat1 const& s2)const
```

```
{
   Z mL=3;
   Word loc("SysLat3::red12(SysLat1,SysLat1)");
   CPM_MA
   Z d1=dim1();
   Z d2=dim2();
   Z d3=dim3();
   cpmassert(s1.dim()==d1,loc);
   cpmassert(s2.dim()==d2,loc);
   SysLat1 s3(lat3_);
   for (Z k=0;k<d3;++k){
      C ck;
      for (Z i=0;i<d1;++i){
         C ci=s1.cui(i).con();
         for (Z j=0;j<d2;++j){
            C cj=s2.cui(j).con();
            C cij=ci*cj;
            ck+=cij*rep_.cui(i).cui(j).cui(k);
         }
      }
      s3.cui(k)=ck;
   }
   CPM_MZ
   return s3;
}

SysLat1 SysLat3::red23(SysLat1 const& s2, SysLat1 const& s3)const
{
   Z mL=3;
   Word loc("SysLat3::red23(SysLat1,SysLat1)");
   CPM_MA
   Z d1=dim1();
   Z d2=dim2();
   Z d3=dim3();
   cpmassert(s2.dim()==d2,loc);
   cpmassert(s3.dim()==d3,loc);
   SysLat1 s1(lat1_);
   for (Z k=0;k<d1;++k){
      C ck;
      for (Z i=0;i<d2;++i){
         C ci=s2.cui(i).con();
         for (Z j=0;j<d3;++j){
            C cj=s3.cui(j).con();
            C cij=ci*cj;
            ck+=cij*rep_.cui(k).cui(i).cui(j);
         }
      }
      s1.cui(k)=ck;
   }
   CPM_MZ
```

```
   return s1;
}


SysLat1 SysLat3::red31(SysLat1 const& s3, SysLat1 const& s1)const
{
   Z mL=3;
   Word loc("SysLat3::red31(SysLat1,SysLat1)");
   CPM_MA
   Z d1=dim1();
   Z d2=dim2();
   Z d3=dim3();
   cpmassert(s1.dim()==d1,loc);
   cpmassert(s3.dim()==d3,loc);
   SysLat1 s2(lat2_);
   for (Z k=0;k<d2;++k){
      C ck;
      for (Z i=0;i<d1;++i){
         C ci=s1.cui(i).con();
         for (Z j=0;j<d3;++j){
            C cj=s3.cui(j).con();
            C cij=ci*cj;
            ck+=cij*rep_.cui(i).cui(k).cui(j);
         }
      }
      s2.cui(k)=ck;
   }
   CPM_MZ
   return s2;
}

SysLat1 SysLat3::red12(SysLat2 const& s12)const
{
   Z mL=3;
   Word loc("SysLat3::red12(SysLat1,SysLat1)");
   CPM_MA
   Z d1=dim1();
   Z d2=dim2();
   Z d3=dim3();
   cpmassert(s12.dim1()==d1,loc);
   cpmassert(s12.dim2()==d2,loc);
   SysLat1 s3(lat3_);
   for (Z k=0;k<d3;++k){
      C ck;
      for (Z i=0;i<d1;++i){
       //  C ci=s1.cui(i).con();
         for (Z j=0;j<d2;++j){
        //    C cj=s2.cui(j).con();
            C cij=s12.cui(i).cui(j).con();
            ck+=cij*rep_.cui(i).cui(j).cui(k);
```

```
         }
      }
      s3.cui(k)=ck;
   }
   CPM_MZ
   return s3;
}

SysLat1 SysLat3::red23(SysLat2 const& s23)const
{
   Z mL=3;
   Word loc("SysLat3::red23(SysLat1,SysLat1)");
   CPM_MA
   Z d1=dim1();
   Z d2=dim2();
   Z d3=dim3();
   cpmassert(s23.dim1()==d2,loc);
   cpmassert(s23.dim2()==d3,loc);
   SysLat1 s1(lat1_);
   for (Z k=0;k<d1;++k){
      C ck;
      for (Z i=0;i<d2;++i){
       //  C ci=s2.cui(i).con();
         for (Z j=0;j<d3;++j){
         //   C cj=s3.cui(j).con();
          //  C cij=ci*cj;
            C cij=s23.cui(i).cui(j).con();
            ck+=cij*rep_.cui(k).cui(i).cui(j);
         }
      }
      s1.cui(k)=ck;
   }
   CPM_MZ
   return s1;
}


SysLat1 SysLat3::red31(SysLat2 const& s31)const
{
   Z mL=3;
   Word loc("SysLat3::red31(SysLat1,SysLat1)");
   CPM_MA
   Z d1=dim1();
   Z d2=dim2();
   Z d3=dim3();
   cpmassert(s31.dim1()==d3,loc);
   cpmassert(s31.dim2()==d1,loc);
   SysLat1 s2(lat2_);
   for (Z k=0;k<d2;++k){
      C ck;
```

```
      for (Z i=0;i<d1;++i){
          for (Z j=0;j<d3;++j){
            C cij=s31.cui(i).cui(j).con();
            ck+=cij*rep_.cui(i).cui(k).cui(j);
         }
      }
      s2.cui(k)=ck;
   }
   CPM_MZ
   return s2;
}


R SysLat3::dtSup(DynDat const& dd1, DynDat const& dd2, DynDat const& dd3,
         IntActDat const& dd12, Z p, R tWait, R gamma)const
{
   Z mL=1;
   Word loc("SysLat3::dtSup(DynDat,DynDat,DynDat,IntActDat,Z,R,R)");
   CPM_MA
   cpmassert(p>0,loc);
   SysLat3 psi=*this;
   psi.nor_();
   R_Vector mem(p);
   Graph gr;
   R pInv=1./p;
   cpmdata<<endl<<loc<<": Iterative approach to the norm of the\
    Hamiltonian:";
   for (Z i=1;i<=p;++i){
      psi.dot_(dd1,dd2,dd3,dd12);
      R mi=psi.nor_(); // the value is the norm!
      mem[i]=mi; // the value is the norm!
      cpmdata<<endl<<"mi="<<cpm(mi);
      if (tWait>0){
         DisCon dc; dc.ipolMet=0; dc.gamma=gamma;
         psi.mark(gr,dc);
         gr.display();
         cpmwait(tWait);
         cpmprogress("x",i*pInv,3);
      }
   }
   R_Vector stat=CpmArrays::statistics(mem);
   R sigma=stat[3];
   R maxVal=stat[10];
   cpmnote(sigma);
   cpmnote(maxVal);
   R epsMax=maxVal+sigma;
   cpmdata<<endl<<"Norm as estimated from statistcs: epsMax= "<<epsMax;
   R dt=2./epsMax;
   cpmdata<<endl<<"dt=2./epsMax="<<dt;
   CPM_MZ
```

```
   return dt;
}

R_Vector SysLat3::stt(DynDat const& dd1, DynDat const& dd2,
   DynDat const& dd3)const
{
   Z mL=2;
   Word loc("SysLat3::s123(DynDat,DynDat,DynDat)");
   CPM_MA
   Z d1=dim1();
   Z d2=dim2();
   Z d3=dim2();
   R res1=0;
   R res2=0;
   R res3=0;
   R res12=0;
   R res13=0;
   R res23=0;
   R res123=0;
   for (Z i=0;i<d1;++i){
      R s1=lat1_.pos(i)-dd1.x0();
      for (Z j=0;j<d2;++j){
         R s2=lat2_.pos(j)-dd2.x0();
         for (Z k=0;k<d3;++k){
            R s3=lat3_.pos(k)-dd3.x0();
            R wijk=rep_.cui(i).cui(j).cui(k).absSqr();
            res1+=wijk*s1;
            res2+=wijk*s2;
            res3+=wijk*s3;
            res12+=wijk*s1*s2;
            res13+=wijk*s1*s3;
            R s23=s2*s3;
            res23+=wijk*s23;
            res123+=wijk*s1*s23;
         }
      }
   }
   R rest=res1+res2+res3;
   CPM_MZ
   return R_Vector("",res1,res2,res3,res12,res13,res23,res123,rest);
}

void SysLat3::mark(Graph& g, DisCon const& dc)const
{
   CpmImaging::Color::setGamma(dc.gamma);
   bool aspRatFromState=true;
   bool matrixstyle=true;
   bool periodic=true;
   Vl< Vl<C> > rep2=CpmArrays::tenToMat(rep_);
   g.mark(rep2,Iv(),dc.ipolMet,aspRatFromState,periodic,dc.secFac,
```

```
      matrixstyle);
}

/************* experimental version which makes the data
// stuctures of function transform visible
void SysLat3::mark(Graph& g, DisCon const& dc)const
{
   CpmImaging::Color::setGamma(0);
   Z d1=dim1();
   Z d2=dim2();
   Z d3=dim2();
   Z d12=d1+d2;
   Z d123=d12+d3;
   Z d=d1*d2*d3;
   Z nH=g.dim2()-1;
   Z nV=g.dim1()-1-d123;
   Z i,j,k,l;
   Z p=1;
   static Z ii=0,jj=0;
   static bool firstrun=true;
   if (firstrun){
      g.clr_(BLACK);
      firstrun=false;
   }
   for (i=0;i<d1;++i){
      for (j=0;j<d2;++j){
         for (k=0;k<d3;++k){
            C val=rep_[i][j][k];
            Color c(val);
            R w=val.absSqr();
            for (l=0;l<p;++l){
               R rv=CpmRoot::randomR();
               if (rv<w*d){
                  g.putPixel(jj+i,ii,c);
                  g.putPixel(jj+d1+j,ii,c);
                  g.putPixel(jj+d12+k,ii,c);
               }
               if (ii<nH)
                  ii++;
               else{
                  ii=0;
                  if (jj<nV){
                     jj+=d123;
                  }
                  else{
                     jj=0;
                     g.clr_(BLACK);
                  }
               }
            }
```

```
         }
      }
   }
}
*/

vector< vector<C> > SysLat3::transform(Graph& g, DisCon const& dc)const
/*
This provides an alternative form of the concept of a 3-particle wave
function (3 is the situation in this specific function, generalization
to general n is obvious). For each of the d1+d2+d3 configurations
(not d1*d2*d3 !) there is stream of data which, when taken together,
describe the present state. Notice that the length of each of these
streams is d1*d2*d3. An index running simultaneously for the
d1+d2+d3 streams from 0 to d1*d2*d3-1 is a kind of time parameter
within the 'point in time' associated with the state as a whole.
The state thus can be considered an evolution process in this
new time parameter (which we call 'fast time').
As already mentioned, the state is completely determined by the
the data arrays res[i]. This suggests that the primary data are
the arrays res[i], i=1,2,... d1+d2+d3 which provides a 'local view'.
For each of streams it is normal to 'have value 0' i.e. res[i][j]=C(0)
is normal. For given j, there are always three indexes i1, i2, i3
such that res[i1][j], res[i2][j], res[i3][j] differ from C(0) and
are equal. This shows that retreiving the state information from the
arrays res[i] is possible only if one inspect 'time-slices' in
fast time.
The information which makes up a three-particle wave function is
thus contained in 'coincidences' in parallel streams. This should
be a model for my old project to understand many-particle wave functions
as correlation functions of a fast local 'Funkel-Process'.
There is an obvious paralellism to the concept of the late Fritz Bopp
who studied local appearance and dis-appearance of particles as a
process which may underlies quantum mechanical wave functions.
*/
{
   Z d1=dim1();
   Z d2=dim2();
   Z d3=dim2();
   Z d12=d1+d2;
   Z d123=d12+d3;
   Z i,j,k,l;

   vector< vector<C> > res(d123);

   for (i=0;i<d1;++i){
      for (j=0;j<d2;++j){
         for (k=0;k<d3;++k){
            C val0,val=rep_[i][j][k];
            for (l=0;l<d123;++l){ // each res[l] gets an new
```

```
                // component appended
                if (l==i) res[l].push_back(val);
                else if (l==j+d1) res[l].push_back(val);
                else if (l==k+d12) res[l].push_back(val);
                else res[l].push_back(val0);
             }
          }
       }
    }
    return res;
}
////////////////////////////// class TrjLat2 ////////////////////////////

C TrjLat2::hInt()const
{
    C enr=eTot();
    IntActDat dd120=dd12_;
    dd120.free_();
    R betaMem=DynDat::beta();
    DynDat::beta()=0;
    SysLat2 v0(x_);
    v0.dot_(dd1_,dd2_,dd120);
    C enr0=C::I*(x_|v0);
    DynDat::beta()=betaMem;
    return enr-enr0;
}

V< V<R2> > TrjLat2::run_(Graph& g, RunCon& rc, DisCon const& dc)
{
       CPM_SC_RUN
    if (dt<0) dt=-dt; // negative input allows us to use values without
       // energy dependent rescaling
    else{
       R dtMax=dtSup(rc.nIter,rc.tWaitIter,rc.gammaIter);
       dt*=dtMax;
       rc.dt=-dt; // then call another run function subsequently will use
          // dt
    }
    vector<R2> s1v, s2v, s12v;
    g.clr();
    for (Z i=1;i<=nSteps;++i){
       step_(dt);
       cpmprogress("run",nInv*i);
       if (tWaitDocu>0){
          if (docu()){
             CPM_SC_RUN2
             R_Vector rv=stt();
             s1v.push_back(R2(ti,rv[1]));
             s2v.push_back(R2(ti,rv[2]));
             s12v.push_back(R2(ti,rv[3]));
```

```
         }
      }
      if (view()){
         mark(g,dc);
         g.dis();
         cpmwait(rc.tWaitView);
      }
   }
   if (tWaitDocu>0){
      CPM_SC_RUN3
      g.setText("TrjLat2: run documentation");
      g.show(doc);
      cpmwait(tWaitDocu,2);
   }
   V< V<R2> > res(3);
   res[1]=V<R2>(s1v);
   res[2]=V<R2>(s2v);
   res[3]=V<R2>(s12v);
   if (tWaitRes>0){
      g.clr_();
      g.setAutoScl_(true);
      g.setText("TrjLat2: res documentation");
      g.mark(res);
      g.dis();
      cpmwait(tWaitRes,2);
   }
   return res;
}

////////////////////////////// class TrjLat3 //////////////////////////////

C TrjLat3::hInt()const
{
   C enr=eTot();
   IntActDat dd120=dd12_;
   dd120.free_();
   R betaMem=DynDat::beta();
   DynDat::beta()=0;
   SysLat3 v0(x_);
   v0.dot_(dd1_,dd2_,dd3_,dd120);
   C enr0=C::I*(x_|v0);
   DynDat::beta()=betaMem;
   return enr-enr0;
}

x2vv TrjLat3::run_(Graph& g, RunCon& rc, DisCon const& dc)
{
   CPM_SC_RUN
   if (dt<0) dt=-dt; // negative input allows us to use values without
      // energy dependent rescaling
```

```
else{
   R dtMax=dtSup(rc.nIter,rc.tWaitIter,rc.gammaIter);
   dt*=dtMax;
   rc.dt=-dt; // then calling another run-function (e.g. one for
      // separated QM, i.e. Sep3::run_()) subsequently with
      // argument rc will use dt, and will disable its own time
      // step adjustment method. So far the stability properties
      // of Sep3 allowed this taking over of a  time step determined
      // by TrjLat3::run_.
}
vector<R2> s1v;
vector<R2> s2v;
vector<R2> s3v;
vector<R2> s12v;
vector<R2> s13v;
vector<R2> s23v;
vector<R2> s123v;
vector<R2> stv;
vector<Word> vw;
cpmdebug(rc.imgSteps);
Vo<Z> imgSteps=rc.imgSteps;
cpmmessage("before sorting");
imgSteps.sort_();
cpmmessage("sorting done");
for (Z i=1;i<=nSteps;++i){
   step_(dt);
   cpmprogress("run",nInv*i);
   if (tWaitDocu>0){
      if (docu()){
         CPM_SC_RUN2
         R_Vector rv=stt();
         s1v.push_back(R2(ti,rv[1]));
         s2v.push_back(R2(ti,rv[2]));
         s3v.push_back(R2(ti,rv[3]));
         s12v.push_back(R2(ti,rv[4]));
         s13v.push_back(R2(ti,rv[5]));
         s23v.push_back(R2(ti,rv[6]));
         s123v.push_back(R2(ti,rv[7]));
         stv.push_back(R2(ti,rv[8]));
      }
   }
   if (view()){
      mark(g,dc);
      Word fn=g.dis(dc.wrtOnShow);
      if (fn.dim()>0) vw.push_back(fn);
      cpmwait(rc.tWaitView);
   }
   if (img()||imgSteps.find(i)!=0){
      g.clr_();
      mark(g,dc);
```

```
         g.setText(Word("Lat3: t=")&cpm(t_));
         g.dis(true);
      }
   }
   if (tWaitDocu>0){
      CPM_SC_RUN3
      g.setText("TrjLat3: run documentation");
      g.show(doc);
      doc.toGnuPlot("TrjLat3_run_documentation.txt");
      cpmwait(tWaitDocu,2);
   }
   V< V<R2> > res(8);
   res[1]=V<R2>(s1v);
   res[2]=V<R2>(s2v);
   res[3]=V<R2>(s3v);
   res[4]=V<R2>(s12v);
   res[5]=V<R2>(s13v);
   res[6]=V<R2>(s23v);
   res[7]=V<R2>(s123v);
   res[8]=V<R2>(stv);
   if (tWaitRes>0){
      g.clr_();
      g.setAutoScl_(true);
      Word txt1("TrjLat3: expectation values and correlations");
      Word txt2(" of particle shifts");
      g.setText(txt1&txt2);
      g.mark(res);
      g.dis(dc.wrtOnShow);
      cpmwait(tWaitRes,2);
   }
   V<Word> res2(vw);
   return x2vv(res,res2);
}

x2vv TrjLat3::run_(Graph& g, RecordHandler const& rch)
{
   RunCon rc(rch,"run control");
   DisCon dc(rch,"display control");
   CPM_SC_RUN
   if (dt<0) dt=-dt; // negative input allows us to use values without
      // energy dependent rescaling
   else{
      R dtMax=dtSup(rc.nIter,rc.tWaitIter,rc.gammaIter);
      dt*=dtMax;
   }
   vector<R2> s1v;
   vector<R2> s2v;
   vector<R2> s3v;
   vector<R2> s12v;
   vector<R2> s13v;
```

```
vector<R2> s23v;
vector<R2> s123v;
vector<R2> stv;
vector<Word> vw;
cpmdebug(rc.imgSteps);
Vo<Z> imgSteps=rc.imgSteps;
cpmmessage("before sorting");
imgSteps.sort_();
cpmmessage("sorting done");
for (Z i=1;i<=nSteps;++i){
   step_(dt);
   cpmprogress("run",nInv*i);
   if (tWaitDocu>0){
      if (docu()){
         CPM_SC_RUN2
         R_Vector rv=stt();
         s1v.push_back(R2(ti,rv[1]));
         s2v.push_back(R2(ti,rv[2]));
         s3v.push_back(R2(ti,rv[3]));
         s12v.push_back(R2(ti,rv[4]));
         s13v.push_back(R2(ti,rv[5]));
         s23v.push_back(R2(ti,rv[6]));
         s123v.push_back(R2(ti,rv[7]));
         stv.push_back(R2(ti,rv[8]));
      }
   }
   if (view()){
      mark(g,dc);
      Word fn=g.dis(dc.wrtOnShow);
      if (fn.dim()>0) vw.push_back(fn);
      cpmwait(rc.tWaitView);
   }
   if (img()||imgSteps.find(i)!=0){
      g.clr_();
      mark(g,dc);
      g.setText(Word("Lat3: t=")&cpm(t_));
      g.dis(true);
  }
}
if (tWaitDocu>0){
   CPM_SC_RUN3
   g.setText("TrjLat3: run documentation");
   g.show(doc);
   doc.toGnuPlot("TrjLat3_run_documentation.txt");
   cpmwait(tWaitDocu,2);
}
V< V<R2> > res(8);
res[1]=V<R2>(s1v);
res[2]=V<R2>(s2v);
res[3]=V<R2>(s3v);
```

```
   res[4]=V<R2>(s12v);
   res[5]=V<R2>(s13v);
   res[6]=V<R2>(s23v);
   res[7]=V<R2>(s123v);
   res[8]=V<R2>(stv);
   if (tWaitRes>0){
      g.clr_();
      g.setAutoScl_(true);
      Word txt1("TrjLat3: expectation values and correlations");
      Word txt2(" of particle shifts");
      g.setText(txt1&txt2);
      g.mark(res);
      g.dis(dc.wrtOnShow);
      cpmwait(tWaitRes,2);
   }
   V<Word> res2(vw);
   return x2vv(res,res2);
}

//////////////////////////////////////// class Sep2 ////////////////////////


SysLat1 Sep2::vInt(SysLat1 const& y1, SysLat1 const& y2,
   DynDat const& dyn1, DynDat const& dyn2, IntActDat const& ia12)
{
   Lat1 lat1=y1.lat(), lat2=y2.lat();
   Z d1=lat1.d(), d2=lat2.d();
   R q1=dyn1.q(),q2=dyn2.q();
   R q12=q1*q2;
   SysLat1 res(lat2); // zero vector
   C fac=C::Ibar;
   for (Z j=0;j<d2;++j){
      R xj=lat2.pos(j);
      R resj=0;
      for (Z i=0;i<d1;++i){
         R xi=lat1.pos(i);
         R rij=ia12.dis(xi,xj);
         R pij=y1.cui(i).absSqr()*q12*ia12.pot(rij);
         resj+=pij;
      }
      res.cui(j)=fac*y2.cui(j)*resj;
   }
   return res;
}

X2<SysLat1, SysLat1> Sep2::dot(SysLat1 const& x1, SysLat1 const& x2,
   DynDat const& dd1, DynDat const& dd2,IntActDat const& dd12)
      // returns the velocities determined by the arguments
      // very transparent form for generalisation to more particles.
{
```

```
   SysLat1 v1(x1),v2(x2);
   v1.dot_(dd1);
   v2.dot_(dd2);
   SysLat1 dv1=vInt(x2,x1,dd1,dd2,dd12);
   SysLat1 dv2=vInt(x1,x2,dd1,dd2,dd12);
   v1+=dv1;
   v2+=dv2;
   return X2<SysLat1, SysLat1>(v1,v2);
}

C Sep2::hInt()const
//: h interaction (h for hamiltonian)
{
   R res=vIntScl(x1_,x2_,dd1_,dd2_,dd12_);
   res+=x1_.enrHomFil(dd1_);
   res+=x2_.enrHomFil(dd2_);
   return C(res);
}

void Sep2::stepGam_(R dt)
{
   R tau=dt*0.5;
   t_+=tau;
   x1_+=v1_*tau;
   x2_+=v2_*tau;
   X2<SysLat1,SysLat1> vv=dot(x1_,x2_,dd1_,dd2_,dd12_);
   v1_=vv.c1()*2-v1_;
   v2_=vv.c2()*2-v2_;
   t_+=tau;
   x1_+=v1_*tau;
   x2_+=v2_*tau;
}

void Sep2::step_(R dt)
{
   // fine, avoids oscillatory behavior
   // of the diagnostic quantities norm and energy
      R tau=dt*0.5;
      stepGam_(tau);
      stepGam_(tau);
}

// master implementation of run_ functions
V< V<R2> > Sep2::run_(Graph& g, RunCon& rc, DisCon const& dc)
{
   CPM_SC_RUN
   if (dt<0) dt=-dt; // negative input allows us to use values without
      // energy dependent rescaling
   else{
      R dtMax=SysLat2(x1_,x2_).
```

```
         dtSup(dd1_,dd2_,dd12_,rc.nIter,rc.tWaitIter,rc.gammaIter);
      dt*=dtMax;
   }
   vector<R2> s1v, s2v, s12v;

   for (Z i=1;i<=nSteps;++i){
      step_(dt);
      cpmprogress("run",nInv*i);
      if (tWaitDocu>0){
         if (docu()){
            CPM_SC_RUN2
            R_Vector rv=stt();
            s1v.push_back(R2(ti,rv[1]));
            s2v.push_back(R2(ti,rv[2]));
            s12v.push_back(R2(ti,rv[3]));
         }
      }
      if (view()){
         mark(g,dc);
         g.dis();
         cpmwait(rc.tWaitView);
      }
   }
   if (tWaitDocu>0){
      CPM_SC_RUN3
      g.setText("Sep2: run documentation");
      g.show(doc);
      cpmwait(tWaitDocu,2);
   }
   V< V<R2> > res(3);
   res[1]=V<R2>(s1v);
   res[2]=V<R2>(s2v);
   res[3]=V<R2>(s12v);
   if (tWaitRes>0){
      g.clr_();
      g.setAutoScl_(true);
      g.setText("Sep2: res documentation");
      g.mark(res);
      g.dis();
      cpmwait(tWaitRes,2);
   }
   return res;
}

void Sep2::mark(Graph& g, DisCon const& dc)const
{
   using CpmLinAlg::R2;
   using CpmArrays::V;
   using namespace CpmImaging;
```

```
V< V<R2> > dat(6);
Lat1 lat1=x1_.lat();
Z d1=lat1.d();
R xi=lat1.fir();
V<R2> re1(d1);
V<R2> im1(d1);
V<R2> ab1(d1);
Z i;
for (i=0;i<d1;++i){
   C zi=x1_.cui(i);
   R2 rei(xi,zi.real());
   R2 imi(xi,zi.imag());
   R2 abi(xi,zi.absVal());
   Z ip=i+1;
   re1.cui(ip)=rei;
   im1.cui(ip)=imi;
   ab1.cui(ip)=abi;
   lat1.next_1(xi);
}
dat[1]=re1;
dat[2]=im1;
dat[3]=ab1;

Lat1 lat2=x2_.lat();
Z d2=lat2.d();
xi=lat2.fir();
V<R2> re2(d2);
V<R2> im2(d2);
V<R2> ab2(d2);
for (i=0;i<d2;++i){
   C zi=x2_.cui(i);
   R2 rei(xi,zi.real());
   R2 imi(xi,zi.imag());
   R2 abi(xi,zi.absVal());
   Z ip=i+1;
   re2.cui(ip)=rei;
   im2.cui(ip)=imi;
   ab2.cui(ip)=abi;
   lat2.next_1(xi);
}
dat[4]=re2;
dat[5]=im2;
dat[6]=ab2;

V<Color> cl(6);
cl[1]=RED;  // color for real part
cl[2]=GREEN;   // color for imaginary part
cl[3]=BLACK; // color for absolute value
cl[4]=LIGHTRED;  // color for real part
cl[5]=LIGHTGREEN;   // color for imaginary part
```

```
   cl[6]=LIGHTGRAY; // color for absolute value
   g.setAutoScl_(dc.autoScl);
   g.mark(dat,cl,dc.secFac);
   g.addText(Word("t=")&cpm(t_));
}

//////////////////////////////////////// class Sep3 ////////////////////////

X3<SysLat1, SysLat1, SysLat1> Sep3::dot(SysLat1 const& x1,
   SysLat1 const& x2, SysLat1 const& x3,
   DynDat const& dd1, DynDat const& dd2, DynDat const& dd3,
   IntActDat const& dd12)
{
   SysLat1 v1(x1),v2(x2),v3(x3);
   v1.dot_(dd1);
   v2.dot_(dd2);
   v3.dot_(dd3);
   SysLat1 dv1=Sep2::vInt(x2,x1,dd1,dd2,dd12)
      +Sep2::vInt(x3,x1,dd1,dd3,dd12);
   SysLat1 dv2=Sep2::vInt(x1,x2,dd1,dd2,dd12)
      +Sep2::vInt(x3,x2,dd2,dd3,dd12);
   SysLat1 dv3=Sep2::vInt(x1,x3,dd1,dd3,dd12)
      +Sep2::vInt(x2,x3,dd2,dd3,dd12);
   v1+=dv1;
   v2+=dv2;
   v3+=dv3;
   return X3<SysLat1, SysLat1,SysLat1>(v1,v2,v3);
}

C Sep3::hInt()const
{
   R res=0;
   res+=Sep2::vIntScl(x1_,x2_,dd1_,dd2_,dd12_);
   res+=Sep2::vIntScl(x1_,x3_,dd1_,dd3_,dd12_);
   res+=Sep2::vIntScl(x2_,x3_,dd2_,dd3_,dd12_);
   res+=x1_.enrHomFil(dd1_);
   res+=x2_.enrHomFil(dd2_);
   res+=x3_.enrHomFil(dd3_);
   return C(res);
}

void Sep3::stepGam_(R dt)
{
   R tau=dt*0.5;
   t_+=tau;
   x1_+=v1_*tau;
   x2_+=v2_*tau;
   x3_+=v3_*tau;
   X3<SysLat1,SysLat1,SysLat1> vvv=dot(x1_,x2_,x3_,dd1_,dd2_,dd3_,dd12_);
   v1_=vvv.c1()*2-v1_;
```

```
   v2_=vvv.c2()*2-v2_;
   v3_=vvv.c3()*2-v3_;
   t_+=tau;
   x1_+=v1_*tau;
   x2_+=v2_*tau;
   x3_+=v3_*tau;
}

void Sep3::step_(R dt)
{
   const bool correctNorm=false;
      // we choose false so that norm conservation of the method
      // can not arise as an artifact of implementation

   const bool twoSteps=true; // true is proper
      // clean way of verifying the oscillatory behavior of 2 steps
      // period if the combination of two steps int one are missing

   if (twoSteps){
      R tau=dt*0.5;
      stepGam_(tau);
      stepGam_(tau);
   }
   else stepGam_(dt);

   if (correctNorm){
      R eps=1e-5; // controls the rigidity of control
      R n1=x1_.absVal(), n2=x2_.absVal(), n3=x3_.absVal();
      R d1=cpmabs(n1-1), d2=cpmabs(n2-1), d3=cpmabs(n3-1);
      if (d1>eps){
         R corr=1./n1; x1_*=corr; v1_*=corr;
      }
      if (d2>eps){
         R corr=1./n2; x2_*=corr; v2_*=corr;
      }
      if (d3>eps){
         R corr=1./n3; x3_*=corr; v3_*=corr;
      }
   }
}

x2vv Sep3::run_(Graph& g, RunCon& rc, DisCon const& dc)
{
   CPM_SC_RUN
   if (dt<0) dt=-dt; // negative input allows us to use values without
      // energy dependent rescaling
   else{
      dt*=dtSup( rc.nIter,rc.tWaitIter,rc.gammaIter);
         // now based on SysLat3 as a whole
   }
```

```
Vo<Z> imgSteps=rc.imgSteps;
imgSteps.sort_();
cpmdebug(imgSteps);
vector<Word> vw;
vector<R2> s1v, s2v, s3v, s12v, s13v, s23v, s123v,stv; //st = s total
   // s1+s2+s3, 't' is used similar to '123'
vector<R2> s1vc, s2vc, s3vc, s12vc, s13vc, s23vc, s123vc, stvc;
   // means for storing output from stt()
Z i;
Color cGridMem=g.getGridColor();
for (i=1;i<=nSteps;++i){
   step_(dt); // promotes the wave functions
   cs_.step_(dt,dd1_,dd2_,dd3_,dd12_);
      // promotes the classical particles
   cpmprogress("run",nInv*i);
   if (tWaitDocu>0){
      if (docu()){
        CPM_SC_RUN2
        R_Vector rv=stt();
        s1v.push_back(R2(ti,rv[1]));
        s2v.push_back(R2(ti,rv[2]));
        s3v.push_back(R2(ti,rv[3]));
        s12v.push_back(R2(ti,rv[4]));
        s13v.push_back(R2(ti,rv[5]));
        s23v.push_back(R2(ti,rv[6]));
        s123v.push_back(R2(ti,rv[7]));
        stv.push_back(R2(ti,rv[8]));

        R_Vector rvc=sttCls(); // classical motion
        s1vc.push_back(R2(ti,rvc[1]));
        s2vc.push_back(R2(ti,rvc[2]));
        s3vc.push_back(R2(ti,rvc[3]));
        s12vc.push_back(R2(ti,rvc[4]));
        s13vc.push_back(R2(ti,rvc[5]));
        s23vc.push_back(R2(ti,rvc[6]));
        s123vc.push_back(R2(ti,rvc[7]));
        stvc.push_back(R2(ti,rvc[8]));
      }
   }
   if (view()){
      mark(g,dc);
      Word fn=g.dis(dc.wrtOnShow);
      if (fn.dim()>0) vw.push_back(fn);
      cpmwait(rc.tWaitView);
   }
   if (img()||imgSteps.find(i)!=0){
      g.clr_(WHITE);
      DisCon dcMod=dc;
      dcMod.areaSty=false;
      g.setThcLin_(true);
```

```
         mark(g,dcMod);
         g.dis(true);
         g.clr_();
         g.setThcLin_(false);
         dcMod.areaSty=true;
         mark(g,dcMod);
         g.setText(Word("Sep3: t=")&cpm(t_));
         g.dis(true);
      }
   }
   if (tWaitDocu>0){
      CPM_SC_RUN3
      Word txt1("Sep3: relative changes of norm(black), real energy");
      Word txt2("(violet), imaginary energy(blue)");
      g.setText(txt1&txt2);
      g.show(doc);
      doc.toGnuPlot("Sep3_run_documentation.txt");
      cpmwait(tWaitDocu,2);
   }
   V< V<R2> > res(16);
// quantum data
   res[1]=V<R2>(s1v);
   res[2]=V<R2>(s2v);
   res[3]=V<R2>(s3v);
   res[4]=V<R2>(s12v);
   res[5]=V<R2>(s13v);
   res[6]=V<R2>(s23v);
   res[7]=V<R2>(s123v);
   res[8]=V<R2>(stv);
// classical data
   res[9]=V<R2>(s1vc);
   res[10]=V<R2>(s2vc);
   res[11]=V<R2>(s3vc);
   res[12]=V<R2>(s12vc);
   res[13]=V<R2>(s13vc);
   res[14]=V<R2>(s23vc);
   res[15]=V<R2>(s123vc);
   res[16]=V<R2>(stvc);
   if (tWaitRes>0){
      g.clr();
      g.setAutoScl_(true);
      if (dc.showCor){
         Word txt1("Sep3: expectation values and correlations");
         Word txt2(" of particle shifts");
         g.setText(txt1&txt2);
         g.mark(res);
      }
      else{
         V< V<R2> > resSmall(2);
         resSmall[1]=res[8];
```

```
         resSmall[2]=res[16];
         Word txt1("Sep3: sum of particle shifts, QM and classical");
         g.setText(txt1);
         g.mark(resSmall);
      }
      g.dis(dc.wrtOnShow);
      cpmwait(tWaitRes,2);
   }
   if (!dc.showCls) res=res.resize(8);
      // then TrjLat3 and Sep3 return data of equal format which in some
      // applications is required
   V<Word> res2(vw);
   return x2vv(res,res2);
}

x2vv Sep3::run_(Graph& g, RecordHandler const& rch)
{
   RunCon rc(rch,"run control");
   DisCon dc(rch,"display control");
   CPM_SC_RUN
   if (dt<0) dt=-dt; // negative input allows us to use values without
      // energy dependent rescaling
   else{
      dt*=dtSup( rc.nIter,rc.tWaitIter,rc.gammaIter);
         // now based on SysLat3 as a whole
   }
   Vo<Z> imgSteps=rc.imgSteps;
   imgSteps.sort_();
   cpmdebug(imgSteps);
   vector<Word> vw;
   vector<R2> s1v, s2v, s3v, s12v, s13v, s23v, s123v,stv; //st = s total
      // s1+s2+s3, 't' is used similar to '123'
   vector<R2> s1vc, s2vc, s3vc, s12vc, s13vc, s23vc, s123vc, stvc;
      // means for storing output from stt()
   Z i;
   Color cGridMem=g.getGridColor();
   for (i=1;i<=nSteps;++i){
      step_(dt); // promotes the wave functions
      cs_.step_(dt,dd1_,dd2_,dd3_,dd12_);
         // promotes the classical particles
      cpmprogress("run",nInv*i);
      if (tWaitDocu>0){
         if (docu()){
           CPM_SC_RUN2
           R_Vector rv=stt();
           s1v.push_back(R2(ti,rv[1]));
           s2v.push_back(R2(ti,rv[2]));
           s3v.push_back(R2(ti,rv[3]));
           s12v.push_back(R2(ti,rv[4]));
           s13v.push_back(R2(ti,rv[5]));
```

```
            s23v.push_back(R2(ti,rv[6]));
            s123v.push_back(R2(ti,rv[7]));
            stv.push_back(R2(ti,rv[8]));

            R_Vector rvc=sttCls(); // classical motion
            s1vc.push_back(R2(ti,rvc[1]));
            s2vc.push_back(R2(ti,rvc[2]));
            s3vc.push_back(R2(ti,rvc[3]));
            s12vc.push_back(R2(ti,rvc[4]));
            s13vc.push_back(R2(ti,rvc[5]));
            s23vc.push_back(R2(ti,rvc[6]));
            s123vc.push_back(R2(ti,rvc[7]));
            stvc.push_back(R2(ti,rvc[8]));
         }
      }
      if (view()){
         mark(g,dc);
         Word fn=g.dis(dc.wrtOnShow);
         if (fn.dim()>0) vw.push_back(fn);
         cpmwait(rc.tWaitView);
      }
      if (img()||imgSteps.find(i)!=0){
         if (imgSteps.find(i)!=0){
            g.clr_(WHITE);
            DisCon dcMod=dc;
            dcMod.areaSty=false;
            g.setThcLin_(true);
            mark(g,dcMod);
            g.dis(true);
            g.clr_();
            g.setThcLin_(false);
           // dcMod.areaSty=true;
         }
    //     mark(g,dcMod);
         mark(g,dc);
         g.setText(Word("Sep3: t=")&cpm(t_));
         g.dis(true);
      }
   }
   if (tWaitDocu>0){
      CPM_SC_RUN3
      Word txt1("Sep3: relative changes of norm(black), real energy");
      Word txt2("(violet), imaginary energy(blue)");
      g.setText(txt1&txt2);
      g.show(doc);
      doc.toGnuPlot("Sep3_run_documentation.txt");
      cpmwait(tWaitDocu,2);
   }
   V< V<R2> > res(16);
// quantum data
```

```
   res[1]=V<R2>(s1v);
   res[2]=V<R2>(s2v);
   res[3]=V<R2>(s3v);
   res[4]=V<R2>(s12v);
   res[5]=V<R2>(s13v);
   res[6]=V<R2>(s23v);
   res[7]=V<R2>(s123v);
   res[8]=V<R2>(stv);
// classical data
   res[9]=V<R2>(s1vc);
   res[10]=V<R2>(s2vc);
   res[11]=V<R2>(s3vc);
   res[12]=V<R2>(s12vc);
   res[13]=V<R2>(s13vc);
   res[14]=V<R2>(s23vc);
   res[15]=V<R2>(s123vc);
   res[16]=V<R2>(stvc);
   if (tWaitRes>0){
      g.clr();
      g.setAutoScl_(true);
      if (dc.showCor){
         Word txt1("Sep3: expectation values and correlations");
         Word txt2(" of particle shifts");
         g.setText(txt1&txt2);
         g.mark(res);
      }
      else{
         V< V<R2> > resSmall(2);
         resSmall[1]=res[8];
         resSmall[2]=res[16];
         Word txt1("Sep3: sum of particle shifts, QM and classical");
         g.setText(txt1);
         g.mark(resSmall);
      }
      g.dis(dc.wrtOnShow);
      cpmwait(tWaitRes,2);
   }
   if (!dc.showCls) res=res.resize(8);
      // then TrjLat3 and Sep3 return data of equal format which in some
      // applications is required
   V<Word> res2(vw);
   return x2vv(res,res2);
}


void Sep3::mark(Graph& g, DisCon const& dc)const
{
   if (dc.areaSty){
      SysLat3(x1_,x2_,x3_).mark(g,dc);
      return;
```

```
}
using CpmLinAlg::R2;
using CpmArrays::V;
using namespace CpmImaging;

V< V<R2> > dat(9);
Lat1 lat1=x1_.lat();
Lat1 lat2=x2_.lat();
Lat1 lat3=x3_.lat();

Z d1=lat1.d();
Z d2=lat2.d();
Z d3=lat3.d();

R xi=lat1.fir();
V<R2> re1(d1);
V<R2> im1(d1);
V<R2> ab1(d1);
Z i;
for (i=0;i<d1;++i){
   C zi=x1_.cui(i);
   R2 rei(xi,zi.real());
   R2 imi(xi,zi.imag());
   R2 abi(xi,zi.absVal());
   Z ip=i+1;
   re1.cui(ip)=rei;
   im1.cui(ip)=imi;
   ab1.cui(ip)=abi;
   lat1.next_1(xi);
}
dat[1]=re1;
dat[2]=im1;
dat[3]=ab1;

xi=lat2.fir();
V<R2> re2(d2);
V<R2> im2(d2);
V<R2> ab2(d2);
for (i=0;i<d2;++i){
   C zi=x2_.cui(i);
   R2 rei(xi,zi.real());
   R2 imi(xi,zi.imag());
   R2 abi(xi,zi.absVal());
   Z ip=i+1;
   re2.cui(ip)=rei;
   im2.cui(ip)=imi;
   ab2.cui(ip)=abi;
   lat2.next_1(xi);
}
dat[4]=re2;
```

```
   dat[5]=im2;
   dat[6]=ab2;

   xi=lat3.fir();
   V<R2> re3(d3);
   V<R2> im3(d3);
   V<R2> ab3(d3);
   for (i=0;i<d3;++i){
      C zi=x3_.cui(i);
      R2 rei(xi,zi.real());
      R2 imi(xi,zi.imag());
      R2 abi(xi,zi.absVal());
      Z ip=i+1;
      re3.cui(ip)=rei;
      im3.cui(ip)=imi;
      ab3.cui(ip)=abi;
      lat3.next_1(xi);
   }
   dat[7]=re3;
   dat[8]=im3;
   dat[9]=ab3;

   V<Color> cl(9);
   cl[1]=RED;  // color for real part
   cl[2]=GREEN;   // color for imaginary part
   cl[3]=BLACK; // color for absolute value
   cl[4]=LIGHTRED;  // color for real part
   cl[5]=LIGHTGREEN;   // color for imaginary part
   cl[6]=LIGHTGRAY; // color for absolute value
   cl[7]=ORANGE;  // color for real part
   cl[8]=CYAN;   // color for imaginary part
   cl[9]=DARKGRAY; // color for absolute value
   g.setAutoScl_(dc.autoScl);
   g.mark(dat,cl,dc.secFac);

   R x1=cs_.x1(), x2=cs_.x2(), x3=cs_.x3();
   V<R> xList("",x1,x2,x3); // classical positions
   V<Color> xColors("",RED,GREEN,BLUE);
   R y1=dd1_.x0(), y2=dd2_.x0(), y3=dd3_.x0();
   V<R> yList("",y1,y2,y3); // equilibrium positions
   V<Color> yColors("",DARKGRAY,DARKGRAY,DARKGRAY);
   R z1=x1_.s(), z2=x2_.s(), z3=x3_.s();
   V<R> zList("",z1,z2,z3); // quantum positions
   V<Color> zColors("",LIGHTRED,LIGHTGREEN,LIGHTBLUE);
   g.mark(xList,xColors);
   g.mark(zList,zColors);
   g.mark(yList,yColors);
// show the time on the graph
   Word tText=Word("t=")&cpm(t_);
   R xT=g.getGridTextPos()[1], yT=0, hT=g.getGridTextHeg();
```

```
      // yT=0 for automatic y
   g.addText(tText,xT,yT,BLACK,hT);
}

////////////////////////////////////// class Fun3 ////////////////////////
bool Fun3::simplify_=false;
R Fun3::epsilon_=0;

void Fun3::step_(R dt)
{
   bool correctNorm=true;
   bool simplification=false;
   // no simplification should be allowed
   Z mL=2;
   Word loc("Fun3::step_(R)");
   CPM_MA

   SysLat3 x(x1_,x2_,x3_);
   TrjLat3 tx(dd1_,dd2_,dd3_,dd12_,x,t_);
   tx.step_(dt);
   SysLat3 xt=tx.x();

   TrjLat2 tx12(dd1_,dd2_,dd12_,x1_,x2_,t_);
   TrjLat2 tx23(dd2_,dd3_,dd12_,x2_,x3_,t_);
   TrjLat2 tx31(dd3_,dd1_,dd12_,x3_,x1_,t_);

   if (!simplify_){
      // surprising that this step has nearly no influence
      tx12.step_(dt);
      tx23.step_(dt);
      tx31.step_(dt);
   }

   SysLat2 x12=tx12.x();
   SysLat2 x23=tx23.x();
   SysLat2 x31=tx31.x();

   x1_=xt.red23(x23);
   x2_=xt.red31(x31);
   x3_=xt.red12(x12);

   if (epsilon_>=0){
      R eps=epsilon_; // controls the rigidity of control
      // to do the correction always seems to be not worse
      // So one should have a small value, e.g. 1e-9 here
      R n1=x1_.absVal(), n2=x2_.absVal(), n3=x3_.absVal();
      R d1=cpmabs(n1-1), d2=cpmabs(n2-1), d3=cpmabs(n3-1);
      if (d1>eps){
         R corr=1./n1; x1_*=corr;
      }
```

```
      if (d2>eps){
         R corr=1./n2; x2_*=corr;
      }
      if (d3>eps){
         R corr=1./n3; x3_*=corr;
      }
   }
   t_+=dt;
   CPM_MZ
}

x2vv Fun3::run_(Graph& g, RunCon& rc, DisCon const& dc)
// many additions and modifications made to Sep3::run_
// are not carried out here (2008-03-19)
{
   CPM_SC_RUN
   if (dt<0) dt=-dt; // negative input allows us to use values without
      // energy dependent rescaling
   else{
      R dt12=SysLat2(x1_,x2_).
         dtSup(dd1_,dd2_,dd12_,rc.nIter,rc.tWaitIter,rc.gammaIter);
      R dt13=SysLat2(x1_,x3_).
         dtSup(dd1_,dd3_,dd12_,rc.nIter,rc.tWaitIter,rc.gammaIter);
      R dt23=SysLat2(x2_,x3_).
         dtSup(dd2_,dd3_,dd12_,rc.nIter,rc.tWaitIter,rc.gammaIter);
      R dtMax=CpmRootX::inf<R>(dt12,dt13,dt23);
      dt*=dtMax;
      cpmdata<<"dt12="<<dt12<<", dt13="<<dt13<<", dt23="<<dt23<<
         ", dtMax="<<dtMax<<endl;
   }
   vector<Word> vw;
   vector<R2> s1v, s2v, s3v, s12v, s13v, s23v, s123v, stv;
   vector<R2> s1vc, s2vc, s3vc, s12vc, s13vc, s23vc, s123vc, stvc;
      // means for storing output from stt()
   Z i;
   for (i=1;i<=nSteps;++i){
      step_(dt); // promotes the wave functions
      cs_.step_(dt,dd1_,dd2_,dd3_,dd12_);
         // promotes the classical particles
      cpmprogress("run",nInv*i);
      if (tWaitDocu>0){
         if (docu()){
           CPM_SC_RUN2
           R_Vector rv=stt();
           s1v.push_back(R2(ti,rv[1]));
           s2v.push_back(R2(ti,rv[2]));
           s3v.push_back(R2(ti,rv[3]));
           s12v.push_back(R2(ti,rv[4]));
           s13v.push_back(R2(ti,rv[5]));
           s23v.push_back(R2(ti,rv[6]));
```

```
            s123v.push_back(R2(ti,rv[7]));
            stv.push_back(R2(ti,rv[8]));

            R_Vector rvc=sttCls(); // classical motion
            s1vc.push_back(R2(ti,rvc[1]));
            s2vc.push_back(R2(ti,rvc[2]));
            s3vc.push_back(R2(ti,rvc[3]));
            s12vc.push_back(R2(ti,rvc[4]));
            s13vc.push_back(R2(ti,rvc[5]));
            s23vc.push_back(R2(ti,rvc[6]));
            s123vc.push_back(R2(ti,rvc[7]));
            stvc.push_back(R2(ti,rvc[8]));
         }
      }
      if (view()){
         stt();
         mark(g,dc);
         Word fn=g.dis(dc.wrtOnShow);
         if (fn.dim()>0) vw.push_back(fn);
         cpmwait(rc.tWaitView);
      }
   }
   if (tWaitDocu>0){
      CPM_SC_RUN3
      Word txt1("Fun3: relative changes of norm(black), real energy");
      Word txt2("(violet), imaginary energy(blue)");
      g.setText(txt1&txt2);
      g.show(doc);
      cpmwait(tWaitDocu,2);
   }
   V< V<R2> > res(16);
// quantum data
   res[1]=V<R2>(s1v);
   res[2]=V<R2>(s2v);
   res[3]=V<R2>(s3v);
   res[4]=V<R2>(s12v);
   res[5]=V<R2>(s13v);
   res[6]=V<R2>(s23v);
   res[7]=V<R2>(s123v);
   res[8]=V<R2>(stv);
// classical data
   res[9]=V<R2>(s1vc);
   res[10]=V<R2>(s2vc);
   res[11]=V<R2>(s3vc);
   res[12]=V<R2>(s12vc);
   res[13]=V<R2>(s13vc);
   res[14]=V<R2>(s23vc);
   res[15]=V<R2>(s123vc);
   res[16]=V<R2>(stvc);
   if (tWaitRes>0){
```

```
      g.clr();
      g.setAutoScl_(true);
      if (dc.showCor){
         Word txt1("Fun3: expectation values and correlations");
         Word txt2(" of particle shifts");
         g.setText(txt1&txt2);
         g.mark(res);
      }
      else{
         V< V<R2> > resSmall(2);
         resSmall[1]=res[8];
         resSmall[2]=res[16];
         Word txt1("Fun3: sum of particle shifts, QM and classical");
         g.setText(txt1);
         g.mark(resSmall);
      }
      g.dis();
      cpmwait(tWaitRes,2);
   }
   if (!dc.showCls) res=res.resize(8);
      // then TrjLat3 and Fun3 return data of equal format which in some
      // applications is required
   V<Word> res2(vw);
   return x2vv(res,res2);
}

void Fun3::mark(Graph& g, DisCon const& dc)const
{
   if (dc.areaSty){
    SysLat3(x1_,x2_,x3_).mark(g,dc);
    return;
   }
   using CpmLinAlg::R2;
   using CpmArrays::V;
   using namespace CpmImaging;

   V< V<R2> > dat(9);
   Lat1 lat1=x1_.lat();
   Lat1 lat2=x2_.lat();
   Lat1 lat3=x3_.lat();

   Z d1=lat1.d();
   Z d2=lat2.d();
   Z d3=lat3.d();

   R xi=lat1.fir();
   V<R2> re1(d1);
   V<R2> im1(d1);
   V<R2> ab1(d1);
   Z i;
```

```
for (i=0;i<d1;++i){
   C zi=x1_.cui(i);
   R2 rei(xi,zi.real());
   R2 imi(xi,zi.imag());
   R2 abi(xi,zi.absVal());
   Z ip=i+1;
   re1.cui(ip)=rei;
   im1.cui(ip)=imi;
   ab1.cui(ip)=abi;
   lat1.next_1(xi);
}
dat[1]=re1;
dat[2]=im1;
dat[3]=ab1;

xi=lat2.fir();
V<R2> re2(d2);
V<R2> im2(d2);
V<R2> ab2(d2);
for (i=0;i<d2;++i){
   C zi=x2_.cui(i);
   R2 rei(xi,zi.real());
   R2 imi(xi,zi.imag());
   R2 abi(xi,zi.absVal());
   Z ip=i+1;
   re2.cui(ip)=rei;
   im2.cui(ip)=imi;
   ab2.cui(ip)=abi;
   lat2.next_1(xi);
}
dat[4]=re2;
dat[5]=im2;
dat[6]=ab2;

xi=lat3.fir();
V<R2> re3(d3);
V<R2> im3(d3);
V<R2> ab3(d3);
for (i=0;i<d3;++i){
   C zi=x3_.cui(i);
   R2 rei(xi,zi.real());
   R2 imi(xi,zi.imag());
   R2 abi(xi,zi.absVal());
   Z ip=i+1;
   re3.cui(ip)=rei;
   im3.cui(ip)=imi;
   ab3.cui(ip)=abi;
   lat3.next_1(xi);
}
dat[7]=re3;
```

```
   dat[8]=im3;
   dat[9]=ab3;

   V<Color> cl(9);
   cl[1]=RED;  // color for real part
   cl[2]=GREEN;   // color for imaginary part
   cl[3]=BLACK; // color for absolute value
   cl[4]=LIGHTRED;  // color for real part
   cl[5]=LIGHTGREEN;   // color for imaginary part
   cl[6]=LIGHTGRAY; // color for absolute value
   cl[7]=ORANGE;  // color for real part
   cl[8]=CYAN;   // color for imaginary part
   cl[9]=DARKGRAY; // color for absolute value
   g.setAutoScl_(dc.autoScl);
   g.mark(dat,cl,dc.secFac);

   R x1=cs_.x1(), x2=cs_.x2(), x3=cs_.x3();
   V<R> xList("",x1,x2,x3); // classical positions
   V<Color> xColors("",RED,GREEN,BLUE);
   R y1=dd1_.x0(), y2=dd2_.x0(), y3=dd3_.x0();
   V<R> yList("",y1,y2,y3); // equilibrium positions
   V<Color> yColors("",DARKGRAY,DARKGRAY,DARKGRAY);
   R z1=x1_.s(), z2=x2_.s(), z3=x3_.s();
   V<R> zList("",z1,z2,z3); // quantum positions
   V<Color> zColors("",LIGHTRED,LIGHTGREEN,LIGHTBLUE);
   g.mark(xList,xColors);
   g.mark(zList,zColors);
   g.mark(yList,yColors);
}

//////////////////////////////////// class CM3 ////////////////////////////

void CM3::step(R dt, R& x1, R& x2, R& x3, R& v1, R& v2, R& v3,
   DynDat const& dd1, DynDat const& dd2, DynDat const& dd3,
   IntActDat const& dd12)
{
   R q1=dd1.q(), q2=dd2.q(), q3=dd3.q();
   R m1=dd1.m(), m2=dd2.m(), m3=dd3.m();
   R tau=dt*0.5;
   x1+=v1*tau; x2+=v2*tau; x3+=v3*tau;
   R f1=dd1.force(x1) + q1*q2*dd12.fi(x1-x2) + q1*q3*dd12.fi(x1-x3);
   R f2=dd2.force(x2) + q2*q1*dd12.fi(x2-x1) + q2*q3*dd12.fi(x2-x3);
   R f3=dd3.force(x3) + q3*q1*dd12.fi(x3-x1) + q3*q2*dd12.fi(x3-x2);
   R a1=f1/m1, a2=f2/m2, a3=f3/m3;
   v1+=a1*dt; v2+=a2*dt; v3+=a3*dt;
   x1+=v1*tau; x2+=v2*tau; x3+=v3*tau;
}

//////////////////////////// class SpinN ////////////////////////////////
```

```
Z4 SpinN::neg(Z i)const
{
   Z i1=i/n_, i2=i%n_; // 0 <= i1 < m_, 0 <= i2 < n_
      // i=i1*n_+i2
   Z i1p = i1==m_-1 ? 0 : i1+1;
   Z i1m = i1==0 ? m_-1 : i1-1;

   Z i2p = i2==n_-1 ? 0 : i2+1;
   Z i2m = i2==0 ? n_-1 : i2-1;

   Z k1=i1p*n_+i2;
   Z k2=i1m*n_+i2;
   Z k3=i1*n_+i2p;
   Z k4=i1*n_+i2m;
   return Z4(k1,k2,k3,k4);
}

void SpinN::dot_()
{
   Word loc("SpinN::dot_()");
   Z mL=2;
   CPM_MA
   Outside os=CYCLIC;
   Z np=vx_.dim();
   vv_=Vl<C2>(np);
   for (Z i=0;i<np;++i){
      C2 ci=vx_.cui(i);
      Z4 ni=neg(i);
      C2 ri=ia_(ci) + ia_(vx_(ni.x1),ci)+ia_(vx_(ni.x2),ci)
         +ia_(vx_(ni.x3),ci)+ia_(vx_(ni.x4),ci);
      vv_.cui(i)=ri*C::Ibar*rf_;
   }
   CPM_MZ
}

C SpinN::hInt()const
{
   C res;
   Outside os=CYCLIC;
   Z np=vx_.dim();
   for (Z i=0;i<np;++i){
      Z4 ni=neg(i);
      C2 ci=vx_.cui(i);
      res+= (ia_.avr(vx_(ni.x1),ci)+ia_.avr(vx_(ni.x3),ci)
        +ia_.avr(vx_(ni.x2),ci)+ia_.avr(vx_(ni.x4),ci));
   }
   return res*0.5;
}

R SpinN::hInt_i(Z i)const
```

```
{
   Outside os=CYCLIC;
   C2 ci=vx_[i];
   C z=ia_.avr(vx_(i-1,os),ci)
      + ia_.avr(vx_(i+1,os),ci);
   return z[1];
}

R SpinN::eSep_i(Z i)const
{
   C2 ci=vx_[i];
   return (ci|ia_(ci))[1];
}

R SpinN::eTot_i(Z i)const
{
   return eSep_i(i)+hInt_i(i);
}

C SpinN::eSep()const
{
   C res;
   Z np=vx_.dim();
   for (Z i=0;i<np;++i){
      C2 ci=vx_.cui(i);
      res+=(ci|ia_(ci));
   }
   return res;
}

R3 SpinN::sig(Z i)const
{
   C2 ci=vx_.cui(i);
   C v1=(ci|CC2::sig1*ci);
   C v2=(ci|CC2::sig2*ci);
   C v3=(ci|CC2::sig3*ci);
   return R3(v1[1],v2[1],v3[1]);
}

namespace{ // anonymous
   const R xg=-0.5;
   const R yg=0.5*sqrt(3.);
   const R2 er(1,0);
   const R2 eg(xg,yg);
   const R2 eb(xg,-yg);
      // er+eg+eb=0,
      // er.absVal()==eg.absVal()==eb.absVal()==1
   C transform(R3 const& r){
      R xr=(r[1]+1);
      R xg=(r[2]+1);
```

```
      R xb=(r[3]+1);
      R2 res=er*xr+eg*xg+eb*xb;
      return C(res[1],res[2]);
   }
}

Vl<C2> SpinN::col()const
{
   Z np=vx_.dim();
   if (np==0) return Vl<C2>(0);
   Vl<C2> res(np);
   for (Z i=0;i<np;++i){
      R3 r3=sig(i);
      C z=transform(r3);
      res[i]=C2(z,z);
   }
   return res;
}

void SpinN::stepGam_(R dt)
{
   Z i,np=parNum();
   R tau=dt*0.5;

   t_+=tau;
   for (i=0;i<np;++i) vx_.cui(i)+=vv_.cui(i)*tau;

   Vl<C2> vMem=vv_;
   dot_();
   for (i=0;i<np;++i) vv_.cui(i)=vv_.cui(i)*2-vMem.cui(i);

   for (i=0;i<np;++i) vx_.cui(i)+=vv_.cui(i)*tau;
   t_+=tau;
}


void SpinN::step_()
{
   const R facRed=0.75;
   const R facInc=1.1;
   const R epsLow=0.1;
   Vl<C2> xMem=vx_;
   R tau=dt_*0.5;
   stepGam_(tau);
   stepGam_(tau);
   if (eps_>0){ // should not be active in the normal case ???
      // seems to be needed nevertheless since otherwise even harmless
      // configurations explode after having evolved rather regularly over
      // a long time. Together with time step adjustment this is much more
      // acceptable than before.
```

```
      // Conserved reversibility for eps_=1e-6 for the helicopter image
      // and seemed to fail reversibility for eps_=1e-4. In this case
      // the initial image was nevertheless recovered, but already in the
      // first third of the run.
      Z i;
      R sup=0;
      for (i=vx_.b();i<=vx_.e();++i){
         R ni=vx_[i].absVal();
         R si=cpmabs(ni-1);
         if (si>sup) sup=si;
      }
      if (sup>eps_){ // do the step again with smaller dt
         t_-=dt_;
         dt_*=facRed;
         vx_=xMem;
         for (i=vx_.b();i<=vx_.e();++i) vx_[i].nor_();
         dot_();
         tau=dt_*0.5;
         stepGam_(tau);
         stepGam_(tau);
      }
      else if (sup<epsLow*eps_){ dt_*=facInc; }
      else ;
   } // eps_>0
   sc_++;
}

namespace{

void lineMark_1(Graph& g, Vl<C2> const& vx, DisCon const& dc, Z met=1)
{
   using CpmImaging::Color;
   static Z jShift=1;
   Z gw=g.w();
   Z gh=g.h();
   Z np=vx.dim();
   Z pix=gh/np;
   Z mult_i=pix;
   Z mult_j=pix;
   Z js=pix,is=pix;
   Z i,j;
   Z d_=np*mult_i;
   jShift+=js;

   Z iState=0;
   Z iCounter=0;
   Z iEff=is;
   Color c;

   for (i=0;i<d_;i++){
```

```
      iEff++;
      if (iCounter==0){
         C val;
         if (met==1){
            val=vx.cui(iState).blc();
         }
         else{
            val=vx.cui(iState)[1];
         }
         c=Color(val);
      }
      iCounter++;
      if (iCounter==mult_i){
         iCounter=0; // triggers new evaluation of c in the next loop
         iState++;
      }
      for (j=0;j<mult_j;j++){
         g.put(Z2(iEff,(j+jShift)%gw),c);
      }
   }
}

}

void SpinN::mark(Graph& g, DisCon const& dc)const
{
   ImgR im(3,m_,n_);
   for (Z i=0;i<m_;++i){
      for (Z j=0;j<n_;++j){
         C2 cij=vx_[i*n_+j];
         R3 yij=cij.toR3();
         im.putPix(i+1,j+1,Color(yij));
      }
   }
   im.markDirect(g,dc.ipolMet);
}

void SpinN::run_(Graph& g, RunCon& rc, DisCon const& dc)
{
   using CpmArrays::R_Matrix;
   using CpmArrays::R_Vector;
   using std::vector;
   Word loc("SpinN::run_(Graph,RunCon,DisCon)");
   Z mL=1;
   CPM_MA
   g.setSclMem_(false);
   R n0=n();
   R n0Inv=cpminv(n0);
   C e0=eTot();
   R sig30=sig3();
```

```
R2 blc0=blcAng();
R theta0=blc0[1];
R phi0=blc0[2];
R eTotR=e0[1];
R eInt=hInt()[1];
cpmvalues2("eTot,eInt",eTotR,eInt,4);
R e0Inv=cpminv(e0.absVal());
R e01=e0[1];
R e02=e0[2];
cpmvalue("dtRel",dt_*dt0Inv_,3);

Z nSteps=rc.nSteps;
R tTot=nSteps*dt_;
R tTotInv=1./tTot;
R tWaitView=rc.tWaitView;
R tWaitDocu=rc.tWaitDocu;
R tWaitBoltzmann=rc.tWaitDocu2;
R tWaitDocu3=rc.tWaitDocu3;
R tWaitRes=rc.tWaitRes;
CpmAlgorithms::PeriodicAlarm view(rc.viewPeriod*dt_);
CpmAlgorithms::PeriodicAlarm docu(rc.docuPeriod*dt_);
if (rc.tWaitIni>0){
   mark(g,dc);
   g.setText("initial state");
   g.dis();
   cpmwait(rc.tWaitIni,2);
}
g.clr_();
vector<R> tv(1,0.), nv(1,0.), e1v(1,0.), e2v(1,0.),
   thetav(1,theta0), phiv(1,phi0), sig3v(1,0);
Z np=vx_.dim();
bool dtVar=getEps()>0;
R ti=0;
while (ti<tTot){
   step_();
   ti=t();
   if (dtVar){
      cpmvalue("dtRel",dt_*dt0Inv_,3);
   }
   cpmprogress("run",tTotInv*ti);
   if (docu(ti)){
      R ni=n();
      R sig3i=sig3();
      C ei=eTot();
      R2 blc=blcAng();
      R thetai=blc[1];
      R phii=blc[2];
      tv.push_back(ti);
      nv.push_back((ni-n0)*n0Inv);
      e1v.push_back((ei[1]-e01)*e0Inv);
```

```
         e2v.push_back((ei[2]-e02)*e0Inv);
         thetav.push_back(thetai);
         phiv.push_back(phii);
         sig3v.push_back(sig3i-sig30);
      }
      if (view(ti)||ti>=tTot){
         mark(g,dc);
         if (ti>=tTot) g.dis(true);
         else g.dis(dc.wrtOnShow);
         cpmwait(rc.tWaitView);
      }
   }
   R fac=10;
   R facInv=1./fac;
   if (tWaitDocu>0){
      R_Matrix doc(4);
      doc[1]=R_Vector(tv);
      doc[2]=R_Vector(nv);
      doc[3]=R_Vector(e1v);
      doc[4]=R_Vector(e2v);
      Graph g1; // it is neither important nor always useful
      // if diagnostic graphics is of the same size as
      // the evolving image. Hence we use a new frame here.
      g1.clr_();
      g1.setText("SpinN: evolution of norm and energy errors");
      g1.show(doc);
      cpmwait(tWaitDocu,2);
   }
   if (tWaitRes>0){
      R_Matrix doc(4);
      doc[1]=R_Vector(tv);
      doc[2]=R_Vector(thetav);
      doc[3]=R_Vector(phiv);
      doc[4]=R_Vector(sig3v);
      Graph g2;
      g2.clr_();
      g2.setColorActive(LIGHTGREEN);
      g2.setText("SpinN: evolution of mean Bloch angles and sig3");
      g2.show(doc);
      cpmwait(tWaitRes,2);
   }
   CPM_MZ
}

void SpinN::runSim_(Graph& g, RunCon& rc, DisCon const& dc)
{
   using CpmArrays::R_Matrix;
   using CpmArrays::R_Vector;
   using std::vector;
   Word loc("SpinN::runSim_(Graph,RunCon,DisCon)");
```

```
   Z mL=1;
   CPM_MA
   g.setSclMem_(false);
   Z nSteps=rc.nSteps;
   R tTot=nSteps*dt_;
   R tTotInv=1./tTot;
   R tWaitView=rc.tWaitView;
   R tWaitRes=rc.tWaitRes;
   CpmAlgorithms::PeriodicAlarm view(rc.viewPeriod*dt_);
   if (rc.tWaitIni>0){
      mark(g,dc);
      g.setText("initial state");
      g.dis();
      cpmwait(rc.tWaitIni,2);
   }
   g.clr_();
   R ti=0;
   while (ti<tTot){
      step_();
      ti=t();
      cpmprogress("run",tTotInv*ti);
      if (view(ti)||ti>=tTot){
         mark(g,dc);
         if (ti>=tTot) g.dis(true);
         else g.dis(dc.wrtOnShow);
         cpmwait(rc.tWaitView);
      }
   }
   CPM_MZ
}
// clean version for publication. Basing reversal on making dt negative
// and not reversing v
void SpinN::runRevTest_(Graph& g, RunCon& rc, DisCon const& dc)
{
   Word loc("SpinN::runRevTest_(Graph,RunCon,DisCon)");
   Z mL=1;
   CPM_MA
   g.setSclMem_(false);
   Z nSteps=rc.nSteps;
   bool rev=rc.reversed;
   bool fd=rc.fromDisk;
   Z nStepsAll=rev ? nSteps*2 : nSteps;
   R tWaitView=rc.tWaitView;
   R tWaitRes=rc.tWaitRes;
   Z vp=rc.viewPeriod;
   cpmassert(nSteps%vp==0,loc);
   R nInv=1./nStepsAll;
   Word prog("progress");
   Word imgName;
   Z progPane=3;
```

```
   Z i=0;
   cpmprogress(prog,nInv*i,progPane);
   mark(g,dc);
   imgName=g.dis(dc.wrtOnShow);
   cpmcerr<<endl<<"time of "<<imgName<<": "<<t();
   cpmwait(rc.tWaitView);
   for (i=1;i<=nStepsAll;++i){
      step_();
      if (i%vp==0){
         mark(g,dc);
         imgName=g.dis(dc.wrtOnShow);
         cpmcerr<<endl<<"time of "<<imgName<<": "<<t();
         cpmwait(rc.tWaitView);
      }
      if (rev && i==nSteps){
         if (fd){
            Img24 img=g.getImg();
            SpinN sn(img,getIa());
            sn.revDir_(); // changes the sign of dt_
            *this=sn;
         }
         else revDir_(); // changes the sign of dt_
      }
      cpmprogress(prog,nInv*i,progPane);
   }
   CPM_MZ
}

//////////////////////////////// class Qreg ////////////////////////////////

Qreg::Qreg(Z n, Z kind):n_(cpmpos(n)),m_()
{
   if (kind==1) m_[Ky(n_)]=C::one;
}

Qreg::Qreg(Vl<C2> const& v):n_(v.dim()),m_()
{
   if (n_==0) return;
   Ky k(n_);
   bool work=true;
   while (work){
      C val(1.);
      for (Z i=0;i<n_;++i){
         val*=v[i].sel(k.get(i));
      }
      m_[k]=val;
      work=k.next_();
   }
}
```

```
R3 Qreg::sig(Z i)const
{
   Mkc::const_iterator it;
   C r1,r2,r3;
   for (it=m_.begin();it!=m_.end();++it){
      Ky k=it->first;
      C val=it->second;
      bool bi=k.get(i);
      Ky k0=k; k0.set_(i,0);
      Ky k1=k; k1.set_(i,1);
      C2 ci(get(k0),get(k1));
      C val1=CC2::sig1.app(ci,bi);
      C val2=CC2::sig2.app(ci,bi);
      C val3=CC2::sig3.app(ci,bi);
      r1+=(val|val1);
      r2+=(val|val2);
      r3+=(val|val3);
   }
   return R3(r1[1],r2[1],r3[1]);
}

Qreg Qreg::app(CC2 const& a, Z i)const
{
   Qreg res(*this);
   Mkc::const_iterator it;
   for (it=m_.begin();it!=m_.end();++it){
      Ky k=it->first;
      bool bi=k.get(i);
      Ky k0=k; k0.set_(i,0);
      Ky k1=k; k1.set_(i,1);
      C2 ci(get(k0),get(k1));
      C val=a.app(ci,bi);
      res.set_(k,val);
   }
   return res;
}

Qreg Qreg::app(CCCC2 const& a, Z i, Z j)const
{
   Qreg res(*this);
   Mkc::const_iterator it;
   for (it=m_.begin();it!=m_.end();++it){
      Ky k=it->first;
      bool bi=k.get(i);
      bool bj=k.get(j);
      Ky k00=k; k00.set_(i,0); k00.set_(j,0);
      Ky k01=k; k01.set_(i,0); k01.set_(j,1);
      Ky k10=k; k10.set_(i,1); k10.set_(j,0);
      Ky k11=k; k11.set_(i,1); k11.set_(j,1);
      CC2 cij(get(k00),get(k01),get(k10),get(k11));
```

```
      C val=a.app(cij,bi,bj);
      res.set_(k,val);
   }
   return res;
}

Qreg Qreg::ham(CC2 const& ci, CCCC2 const& cij)const
{
   Z mL=3;
   Word loc("Qreg::ham(QregPar)");
   CPM_MA
   Qreg res(n_,0); // zero vector
   for (Z i=0;i<n_;++i){ // loop over nearest neighbors
      res+=app(ci,i);
      Z j=i+1;
      if (j==n_) j=0;
      res+=app(cij,i,j);
   }
   CPM_MZ
   return res;
}

Qreg Qreg::operator+(Qreg const& q)const
{
   Qreg res(*this);
   Mkc::const_iterator i;
   for (i=q.m_.begin();i!=q.m_.end();++i) res.add_(i->first,i->second);
   return res;
}

Qreg& Qreg::operator+=(Qreg const& q)
{
   Mkc::const_iterator i;
   for (i=q.m_.begin();i!=q.m_.end();++i) add_(i->first,i->second);
   return *this;
}

Qreg& Qreg::operator-=(Qreg const& q)
{
   Mkc::const_iterator i;
   for (i=q.m_.begin();i!=q.m_.end();++i) sub_(i->first,i->second);
   return *this;
}

Qreg Qreg::operator*(R r)const
{
   Qreg res(*this);
   Mkc::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i)  res.mul_(i->first,r);
   return res;
```

```
}

Qreg& Qreg::operator*=(R r)
{
   Mkc::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i) mul_(i->first,r);
   return *this;
}

Qreg& Qreg::operator*=(C z)
{
   Mkc::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i) mul_(i->first,z);
   return *this;
}

C Qreg::operator|(Qreg const& q)const
// see Musser,Saini: STL Tutorial and Reference Guide,
// Addison-Wesley 1996, Example 7-6, p. 169,170
// for the pattern of this implementation.
{
   C res;
   Mkc::const_iterator i,i2;
   for (i=m_.begin();i!=m_.end();++i){
      Ky k=i->first;
      i2=q.m_.find(k);
      if (i2!=q.m_.end()){
         C z=i->second;
         C z2=i2->second;
         res+=z.con()*z2;
      }
   }
   return res;
}

Vl<C2> Qreg::den()const
{
   if (n_==0) return Vl<C2>(0);
   Vl<C2> res(n_);
   if (n_==1){
      Ky k(n_);
      C c1=get(k);
      k.next_();
      C c2=get(k);
      res[0]=C2(c1,c2);
      return res;
   }
   Z nm=n_-1;
   for (Z i=0;i<n_;++i){
      R rif=0,rit=0;
```

```
      Ky k(nm);
      bool work=true;
      while (work){
         Ky kif=k;
         kif.ins_(i,false);
         Ky kit=k;
         kit.ins_(i,true);
         rif+=get(kif).absSqr();
         rit+=get(kit).absSqr();
         work=k.next_();
      }
      res[i]=C2(C(rif),C(rit));
   }
   return res;
}

Vl<C2> Qreg::col()const
{
   if (n_==0) return Vl<C2>(0);
   Vl<C2> res(n_);
   for (Z i=0;i<n_;++i){
      R3 r3=sig(i);
      C z=transform(r3);
      res[i]=C2(z,z);
   }
   return res;
}

Qreg Qreg::con(void)const
{
   Qreg res(n_,0);
   Mkc::const_iterator i;
   for (i=m_.begin();i!=m_.end();++i){
      C val=(i->second).con();
      res.set_(i->first,val);
   }
   return res;
}

void Qreg::mark(Graph& g, DisCon const& dc)const
{
   //lineMark_1(g,den(),dc);
   lineMark_1(g,col(),dc,2);
}

R Qreg::dtSup(CC2 const& h0, CCCC2 const& hi, Z p, R tWait)const
{
   Z mL=1;
   Word loc("Qreg::dtSup(QregPar,Z,R)");
   CPM_MA
```

```
   cpmassert(p>0,loc);
   Qreg psi=*this;
   psi.nor_();
   R_Vector mem(p);
   Graph gr;
   R pInv=1./p;
   for (Z i=1;i<=p;++i){
      psi=psi.ham(h0,hi); // no matter of hamiltonian or dot
      R mi=psi.nor_(); // the value is the norm!
      mem[i]=mi; // the value is the norm!
      cpmvalue("enr",mi,2);
      if (tWait>=0){
         psi.mark(gr,DisCon());
         gr.display();
         cpmwait(tWait);
      }
      cpmprogress("dtSup",i*pInv,3);
   }
   R_Vector stat=CpmArrays::statistics(mem);
   R sigma=stat[3];
   R maxVal=stat[10];
   R epsMax=maxVal+sigma;
   cpmvalue("eMax",epsMax,3);
   R res=2./epsMax;
   CPM_MZ
   return res;
}

R_Vector Qreg::stt()const
{
   R_Vector res(1);
   res[1]=mag();
   return res;
}

void TrjQreg::step_(R dt)
{
   R tau=dt*0.5;
   x_+=v_*tau;
   v_-=x_.ham(h0_,hi_).ham(h0_,hi_)*dt;
   x_+=v_*tau;
   t_+=dt;
}

V< V<R2> > TrjQreg::run_(Graph& g, RunCon& rc, DisCon const& dc)
{
   Word loc("TrjQreg::run_(Graph,RunCon,DisCon)");
   Z mL=1;
   CPM_MA
   CPM_SC_RUN
```

```
   if (dt<0) dt=-dt; // negative input allows us to use values without
      // energy dependent rescaling
   else{
      R dtMax=dtSup(rc.nIter,rc.tWaitIter);
      dt*=dtMax;
      rc.dt=-dt; // then call another run function subsequently will use
         // dt
   }
   cpmdebug(rc.dt);
   vector<R2> s1v;
   g.clr();
   for (Z i=1;i<=nSteps;++i){
      step_(dt);
      cpmprogress("run",nInv*i);
      if (tWaitDocu>0){
         if (docu()){
            CPM_SC_RUN2
            R_Vector rv=stt();
            s1v.push_back(R2(ti,rv[1]));
           // s2v.push_back(R2(ti,rv[2]));
          //  s12v.push_back(R2(ti,rv[3]));
         }
      }
      if (view()){
         mark(g,dc);
         g.dis();
         cpmwait(rc.tWaitView);
      }
   }
   if (tWaitDocu>0){
      CPM_SC_RUN3
      g.setText("TrjQreg: run documentation");
      g.show(doc);
      cpmwait(tWaitDocu,2);
   }
   V< V<R2> > res(1);

   res[1]=V<R2>(s1v);
/*
   res[2]=V<R2>(s2v);
   res[3]=V<R2>(s12v);
*/
   if (tWaitRes>0){
      g.clr_();
      g.setAutoScl_(true);
      g.setText("TrjQreg: res documentation");
      g.mark(res);
      g.dis();
      cpmwait(tWaitRes,2);
   }
```

```
   CPM_MZ
   return res;
}

#undef CPM_SC_RUN
#undef CPM_SC_RUN2
#undef CPM_SC_RUN3

// Studying matrix elements of general random-generated pair potentials
// to see whether they are real. Actually those combinations get tested
// the reality of which guarantees conservation of energy. Since
// rather complicated combinatorics is involved, I decided to convince
// myself of the correctness my conjecture before I try to complete my
// proof on paper.
// Result: the conjecture is correct, energy conservation holds for
// arbitrary pair potentials, not only those that are multiplication
// operators in position space.
// The proof on paper was completed 2008-02-26
// So this part of the exercise is no longer important, but it was
// helpful.
//////////////////// class PairStt ////////////////////////////////////
class PairStt{ // pair state
   Z d1_,d2_;
   VV<C> x_;
public:
   PairStt(Z d1, Z d2):d1_(cpmpos(d1)),d2_(cpmpos(d2)),x_(d1_,d2_){}
   Z dim1()const{ return d1_;}
   Z dim2()const{ return d2_;}
   C& operator()(Z i, Z j){ return x_[i][j];}
   C const& operator()(Z i, Z j)const{ return x_[i][j];}
   V<C> trc1(V<C> const& s1)const;
      //: trace 1
      // getting a one-particle state by tracing one state out
   V<C> trc2(V<C> const& s2)const;
      //: trace 2
      // getting a one-particle state by tracing one state out
};

V<C> PairStt::trc1(V<C> const& s1)const
{
   Word loc("PairStt::trc1(V<C>)");
   cpmassert(s1.dim()==d1_,loc);
   V<C> res(d2_);
   Z i,j;
   for (i=1;i<=d2_;++i){
      C ri;
      for (j=1;j<=d1_;++j){
         ri+=s1[j].con()*x_[j][i];
      }
      res[i]=ri;
```

```
   }
   return res;
}

V<C> PairStt::trc2(V<C> const& s2)const
{
   Word loc("PairStt::trc2(V<C>)");
   cpmassert(s2.dim()==d2_,loc);
   V<C> res(d1_);
   Z i,j;
   for (i=1;i<=d1_;++i){
      C ri;
      for (j=1;j<=d2_;++j){
         ri+=x_[i][j]*s2[j].con();
      }
      res[i]=ri;
   }
   return res;
}

//////////////////////////// class Stt ////////////////////////////////////
class Stt{ // state
   // here state of a three-particle system, later we get also
   // Stt4 for 4-particle states which should be effectively the generic
   // case since all relations between 2-particle subsets are no
   // possible: equality, overlap, disjointness
   Z d1_,d2_,d3_;
   VVV<C> x_;
public:
   Stt(Z d1,Z d2, Z d3):d1_(cpmpos(d1)),d2_(cpmpos(d2)),d3_(cpmpos(d3)),
      x_(d1_,d2_,d3_){}

   Stt(V<C> const& p1, V<C> const& p2, V<C> const& p3):d1_(p1.dim()),
      d2_(p2.dim()),d3_(p3.dim()),x_(d1_,d2_,d3_)
   {
      Z i,j,k;
      for (i=1;i<=d1_;++i){
         C pi=p1[i];
         for (j=1;j<=d2_;++j){
            C pij=pi*p2[j];
            for (k=1;k<=d3_;++k){
               x_[i][j][k]=pij*p3[k];
            }
         }
      }
   }

   Stt(PairStt const& pp, V<C> const& s, Word const& w);
      // construction from interacting pair state
      // and a free state, important!
```

```
   Z dim1()const{ return d1_;}
   Z dim2()const{ return d2_;}
   Z dim3()const{ return d3_;}
   C& operator()(Z i, Z j, Z k){ return x_[i][j][k];}
   C const& operator()(Z i, Z j, Z k)const{ return x_[i][j][k];}

   Stt operator+(Stt const& stt)const;
   C operator|(Stt const& stt)const;

};

Stt Stt::operator+(Stt const& stt)const
{
   Word loc("Stt::operator+(Stt)");
   cpmassert(d1_==stt.dim1(),loc);
   cpmassert(d2_==stt.dim2(),loc);
   cpmassert(d3_==stt.dim3(),loc);
   Stt res(d1_,d2_,d3_);
   {
      Z i,j,k;
      for (i=1;i<=d1_;++i){
        for (j=1;j<=d2_;++j){
            for (k=1;k<=d3_;++k){
               res(i,j,k)=x_[i][j][k]+stt(i,j,k);
            }
         }
      }
   }
   return res;
}

C Stt::operator|(Stt const& stt)const
{
   Word loc("Stt::operator+(Stt)");
   cpmassert(d1_==stt.dim1(),loc);
   cpmassert(d2_==stt.dim2(),loc);
   cpmassert(d3_==stt.dim3(),loc);
   C res;
   {
      Z i,j,k;
      for (i=1;i<=d1_;++i){
        for (j=1;j<=d2_;++j){
            for (k=1;k<=d3_;++k){
               res+=x_[i][j][k].con()*stt(i,j,k);
            }
         }
      }
   }
   return res;
```

```
}

Stt::Stt(PairStt const& pp, V<C> const& s, Word const& w)
{
   Word loc("Stt::Stt(PairStt,V<C>,Word)");
   if (w=="12_3"){
      d1_=pp.dim1();
      d2_=pp.dim2();
      d3_=s.dim();
      x_=VVV<C>(d1_,d2_,d3_);
      Z i,j,k;
      for (i=1;i<=d1_;++i){
         for (j=1;j<=d2_;++j){
            C pij=pp(i,j);
            for (k=1;k<=d3_;++k){
               x_[i][j][k]=pij*s[k];
            }
         }
      }
   }
   else if (w=="31_2"){
      d3_=pp.dim1();
      d1_=pp.dim2();
      d2_=s.dim();
      x_=VVV<C>(d1_,d2_,d3_);
      Z i,j,k;
      for (i=1;i<=d3_;++i){
         for (j=1;j<=d1_;++j){
            C pij=pp(i,j);
            for (k=1;k<=d2_;++k){
               x_[j][k][i]=pij*s[k];
            }
         }
      }
   }
   else if (w=="23_1"){
      d2_=pp.dim1();
      d3_=pp.dim2();
      d1_=s.dim();
      x_=VVV<C>(d1_,d2_,d3_);
      Z i,j,k;
      for (i=1;i<=d2_;++i){
        for (j=1;j<=d3_;++j){
           C pij=pp(i,j);
           for (k=1;k<=d1_;++k){
              x_[k][i][j]=pij*s[k];
           }
         }
      }
   }
```

```
   else{
      cpmerror(loc&": bad Word argument");
   }
}

//////////////////////////// class Stt4 ////////////////////////////////////
class Stt4{ // state
   // here state of a four-particle system
   Z d1_,d2_,d3_,d4_;
   VVVV<C> x_;
public:
   Stt4(Z d1,Z d2, Z d3, Z d4):d1_(cpmpos(d1)),d2_(cpmpos(d2)),
      d3_(cpmpos(d3)), d4_(cpmpos(d4)), x_(d1_,d2_,d3_,d4_){}

   Stt4(V<C> const& p1, V<C> const& p2, V<C> const& p3, V<C> const& p4):
      d1_(p1.dim()),d2_(p2.dim()),d3_(p3.dim()),d4_(p4.dim()),
      x_(d1_,d2_,d3_,d4_)
   {
      Z i,j,k,l;
      for (i=1;i<=d1_;++i){
         C pi=p1[i];
         for (j=1;j<=d2_;++j){
            C pij=pi*p2[j];
            for (k=1;k<=d3_;++k){
               C pijk=pij*p3[k];
               for (l=1;l<=d4_;++l){
                  x_[i][j][k][l]=pijk*p4[l];
               }
            }
         }
      }
   }

   Stt4(PairStt const& pp, V<C> const& p1, V<C> const& p2, Word const& w);
   // construction from interacting pair state
   // and two free states, important!

   Z dim1()const{ return d1_;}
   Z dim2()const{ return d2_;}
   Z dim3()const{ return d3_;}
   Z dim4()const{ return d4_;}
   C& operator()(Z i, Z j, Z k, Z l){ return x_[i][j][k][l];}
   C const& operator()(Z i, Z j, Z k, Z l)const{ return x_[i][j][k][l];}

   Stt4 operator+(Stt4 const& stt)const;
   C operator|(Stt4 const& stt)const;

};

Stt4 Stt4::operator+(Stt4 const& stt)const
```

```
{
   Word loc("Stt4::operator+(Stt4)");
   cpmassert(d1_==stt.dim1(),loc);
   cpmassert(d2_==stt.dim2(),loc);
   cpmassert(d3_==stt.dim3(),loc);
   cpmassert(d4_==stt.dim4(),loc);
   Stt4 res(d1_,d2_,d3_,d4_);
   {
      Z i,j,k,l;
      for (i=1;i<=d1_;++i){
        for (j=1;j<=d2_;++j){
            for (k=1;k<=d3_;++k){
               for (l=1;l<=d4_;++l){
                  res(i,j,k,l)=x_[i][j][k][l]+stt(i,j,k,l);
               }
            }
         }
      }
   }
   return res;
}

C Stt4::operator|(Stt4 const& stt)const
{
   Word loc("Stt4::operator+(Stt4)");
   cpmassert(d1_==stt.dim1(),loc);
   cpmassert(d2_==stt.dim2(),loc);
   cpmassert(d3_==stt.dim3(),loc);
   cpmassert(d4_==stt.dim4(),loc);
   C res;
   {
      Z i,j,k,l;
      for (i=1;i<=d1_;++i){
        for (j=1;j<=d2_;++j){
            for (k=1;k<=d3_;++k){
               for (l=1;l<=d4_;++l){
                  res+=x_[i][j][k][l].con()*stt(i,j,k,l);
               }
            }
         }
      }
   }
   return res;
}

Stt4::Stt4(PairStt const& pp, V<C> const& p1,
   V<C> const& p2, Word const& w)
{
   Word loc("Stt4::Stt4(PairStt,V<C>,V<C>,Word)");
   if (w=="12_3_4"){
```

```
      d1_=pp.dim1();
      d2_=pp.dim2();
      d3_=p1.dim();
      d4_=p2.dim();
      x_=VVVV<C>(d1_,d2_,d3_,d4_);
      Z i,j,k,l;
      for (i=1;i<=d1_;++i){
         for (j=1;j<=d2_;++j){
            C pij=pp(i,j);
            for (k=1;k<=d3_;++k){
               for (l=1;l<=d4_;++l){
                  x_[i][j][k][l]=pij*p1[k]*p2[l];
               }
            }
         }
      }
   }
   else if (w=="13_2_4"){
      d1_=pp.dim1();
      d3_=pp.dim2();
      d2_=p1.dim();
      d4_=p2.dim();
      x_=VVVV<C>(d1_,d2_,d3_,d4_);
      Z i,j,k,l;
      for (i=1;i<=d1_;++i){
         for (j=1;j<=d3_;++j){
            C pij=pp(i,j);
            for (k=1;k<=d2_;++k){
               for (l=1;l<=d4_;++l){
                  x_[i][k][j][l]=pij*p1[k]*p2[l];
               }
            }
         }
      }
   }
   else if (w=="14_2_3"){
      d1_=pp.dim1();
      d4_=pp.dim2();
      d2_=p1.dim();
      d3_=p2.dim();
      x_=VVVV<C>(d1_,d2_,d3_,d4_);
      Z i,j,k,l;
      for (i=1;i<=d1_;++i){
         for (j=1;j<=d4_;++j){
            C pij=pp(i,j);
            for (k=1;k<=d2_;++k){
               for (l=1;l<=d3_;++l){
                  x_[i][k][l][j]=pij*p1[k]*p2[l];
               }
            }
```

```
      }
   }
}
else if (w=="23_1_4"){
   d2_=pp.dim1();
   d3_=pp.dim2();
   d1_=p1.dim();
   d4_=p2.dim();
   x_=VVVV<C>(d1_,d2_,d3_,d4_);
   Z i,j,k,l;
   for (i=1;i<=d2_;++i){
      for (j=1;j<=d3_;++j){
         C pij=pp(i,j);
         for (k=1;k<=d1_;++k){
            for (l=1;l<=d4_;++l){
               x_[k][i][j][l]=pij*p1[k]*p2[l];
            }
         }
      }
   }
}
else if (w=="24_1_3"){
   d2_=pp.dim1();
   d4_=pp.dim2();
   d1_=p1.dim();
   d3_=p2.dim();
   x_=VVVV<C>(d1_,d2_,d3_,d4_);
   Z i,j,k,l;
   for (i=1;i<=d2_;++i){
      for (j=1;j<=d4_;++j){
         C pij=pp(i,j);
         for (k=1;k<=d1_;++k){
            for (l=1;l<=d3_;++l){
               x_[k][i][l][j]=pij*p1[k]*p2[l];
            }
         }
      }
   }
}
else if (w=="34_1_2"){
   d3_=pp.dim1();
   d4_=pp.dim2();
   d1_=p1.dim();
   d2_=p2.dim();
   x_=VVVV<C>(d1_,d2_,d3_,d4_);
   Z i,j,k,l;
   for (i=1;i<=d3_;++i){
      for (j=1;j<=d4_;++j){
         C pij=pp(i,j);
         for (k=1;k<=d1_;++k){
```

```
                for (l=1;l<=d2_;++l){
                   x_[k][l][i][j]=pij*p1[k]*p2[l];
                }
             }
          }
       }
    }
    else{
       cpmerror(loc&": bad Word argument");
    }
}

//////////////////////// class PairInt ////////////////////////////////////
class PairInt{ // pair interaction
   Z d1_,d2_;
   VVVV<C> x_;
public:
   PairInt(Z d1, Z d2):d1_(cpmpos(d1)),d2_(cpmpos(d2)),x_(d1_,d1_,d2_,d2_)
      {} // trivial matrix elements but right format
   Z dim1()const{ return d1_;}
   Z dim2()const{ return d2_;}
   PairStt evl(V<C> const& p1, V<C> const& p2)const;
      // returns the pair state which results from the
      // free state p1 \otimes p2 by applying the
      // pair interaction operator *this on it
   PairInt ranVal(Z i=0)const;
      // returns an hermitan pair interaction operator
      // with randomly selected matrix elements
   C& operator()(Z i, Z j, Z k, Z l){ return x_[i][j][k][l];}
   C const& operator()(Z i, Z j, Z k, Z l)const{ return x_[i][j][k][l];}
};

PairStt PairInt::evl(V<C> const& p1, V<C> const& p2)const
{
   Word loc("PairInt::evl(V<C>,V<C>)");
   Z d1=p1.dim(), d2=p2.dim();
   cpmassert(d1==d1_,loc);
   cpmassert(d2==d2_,loc);
   PairStt res(d1_,d2_);
   Z i,j,k,l;
   for (i=1;i<=d1_;++i){
      for (j=1;j<=d2_;++j){
         C rij;
         for (k=1;k<=d1_;++k){
            for (l=1;l<=d2_;++l){
               rij+=x_[i][k][j][l]*p1[k]*p2[l];
            }
         }
         res(i,j)=rij;
      }
```

```
   }
   return res;
}

PairInt PairInt::ranVal(Z p)const
{
   PairInt temp(d1_,d2_);
      // not hermitean
   C z(1.,1.);
   Z i,j,k,l;
   Z ir=p;
   if (ir!=0){
      for (i=1;i<=d1_;++i){
         for (j=1;j<=d1_;++j){
            for (k=1;k<=d2_;++k){
               for (l=1;l<=d2_;++l){
                  temp(i,j,k,l)=z.ranVal(ir++);
               }
            }
         }
      }
   }
   else{
      for (i=1;i<=d1_;++i){
         for (j=1;j<=d1_;++j){
            for (k=1;k<=d2_;++k){
               for (l=1;l<=d2_;++l){
                  temp(i,j,k,l)=z.ranVal();
               }
            }
         }
      }
   }
   PairInt res(d1_,d2_);
      // hermitean result
   for (i=1;i<=d1_;++i){
      for (j=1;j<=d1_;++j){
         for (k=1;k<=d2_;++k){
            for (l=1;l<=d2_;++l){
               res(i,j,k,l)=temp(i,j,k,l)+temp(j,i,l,k).con();
            }
         }
      }
   }
   return res;
}

void CpmQM::testEC(Z d1, Z d2, Z d3)
{
   PairInt v12(d1,d2);
```

```
PairInt v31(d3,d1);
PairInt v23(d2,d3);

v12=v12.ranVal();
v31=v31.ranVal();
v23=v23.ranVal();

C z1(1.,1.);
C z2=z1.ranVal();
C z3=z1.ranVal();

Vr<C> p1(d1,z1);
Vr<C> p2(d2,z2);
Vr<C> p3(d3,z3);
p1=p1.ranVal();
p2=p2.ranVal();
p3=p3.ranVal();
p1.nor_();
p2.nor_();
p3.nor_();

PairStt p12=v12.evl(p1,p2);
PairStt p31=v31.evl(p3,p1);
PairStt p23=v23.evl(p2,p3);
Stt p12_3(p12,p3,"12_3");
Stt p31_2(p31,p2,"31_2");
Stt p23_1(p23,p1,"23_1");
Stt pL=p12_3+p31_2+p23_1;

Stt q12(v12.evl(p1,p2).trc2(p2),p2,p3);
Stt q13(v31.evl(p3,p1).trc1(p3),p2,p3);

Stt q23(p1,v23.evl(p2,p3).trc2(p3),p3);
Stt q21(p1,v12.evl(p1,p2).trc1(p1),p3);

Stt q31(p1,p2,v31.evl(p3,p1).trc2(p1));
Stt q32(p1,p2,v23.evl(p2,p3).trc1(p2));

Stt pR=q12+q13+q23+q21+q31+q32;
cpmdebug((pL|q12));
cpmdebug((pL|q13));
cpmdebug((pL|q21));
cpmdebug((pL|q23));
cpmdebug((pL|q31));
cpmdebug((pL|q32));

cpmdebug((p12_3|q12));
cpmdebug((p12_3|q13));

cpmdebug((p31_2|q12));
```

```
   cpmdebug((p31_2|q13));

   cpmdebug((p23_1|q12));
   cpmdebug((p23_1|q13));

   C res=(pL|pR);
   cpmnote(res);
   cpmdebug(res);
}

void CpmQM::testEC4(Z d1, Z d2, Z d3, Z d4)
{
   PairInt v12(d1,d2);
   PairInt v13(d1,d3);
   PairInt v14(d1,d4);
   PairInt v23(d2,d3);
   PairInt v24(d2,d4);
   PairInt v34(d3,d4);

   v12=v12.ranVal();
   v13=v13.ranVal();
   v14=v14.ranVal();
   v23=v23.ranVal();
   v24=v24.ranVal();
   v34=v34.ranVal();

   C z1(1.,1.);
   C z2=z1.ranVal();
   C z3=z1.ranVal();
   C z4=z1.ranVal();

   Vr<C> p1(d1,z1);
   Vr<C> p2(d2,z2);
   Vr<C> p3(d3,z3);
   Vr<C> p4(d4,z4);

   p1=p1.ranVal();
   p2=p2.ranVal();
   p3=p3.ranVal();
   p4=p4.ranVal();

   p1.nor_();
   p2.nor_();
   p3.nor_();
   p4.nor_();

   PairStt p12=v12.evl(p1,p2);
   PairStt p13=v13.evl(p1,p3);
   PairStt p14=v14.evl(p1,p4);
   PairStt p23=v23.evl(p2,p3);
```

```
PairStt p24=v24.evl(p2,p4);
PairStt p34=v34.evl(p3,p4);

Stt4 p12_3_4(p12,p3,p4,"12_3_4");
Stt4 p13_2_4(p13,p2,p4,"13_2_4");
Stt4 p14_2_3(p14,p2,p3,"14_2_3");

Stt4 p23_1_4(p23,p1,p4,"23_1_4");
Stt4 p24_1_3(p24,p1,p3,"24_1_3");

Stt4 p34_1_2(p34,p1,p2,"34_1_2");

Stt4 pL=p12_3_4 + p13_2_4 + p14_2_3 + p23_1_4 + p24_1_3 + p34_1_2;

Stt4 q12(v12.evl(p1,p2).trc2(p2),p2,p3,p4);
Stt4 q13(v13.evl(p1,p3).trc2(p3),p2,p3,p4);
Stt4 q14(v14.evl(p1,p4).trc2(p4),p2,p3,p4);

Stt4 q21(p1,v12.evl(p1,p2).trc1(p1),p3,p4);
Stt4 q23(p1,v23.evl(p2,p3).trc2(p3),p3,p4);
Stt4 q24(p1,v24.evl(p2,p4).trc2(p4),p3,p4);

Stt4 q31(p1,p2,v13.evl(p1,p3).trc1(p1),p4);
Stt4 q32(p1,p2,v23.evl(p2,p3).trc1(p2),p4);
Stt4 q34(p1,p2,v34.evl(p3,p4).trc2(p4),p4);

Stt4 q41(p1,p2,p3,v14.evl(p1,p4).trc1(p1));
Stt4 q42(p1,p2,p3,v24.evl(p2,p4).trc1(p2));
Stt4 q43(p1,p2,p3,v34.evl(p3,p4).trc1(p3));


Stt4 pR=q12+q13+q14+q21+q23+q24+q31+q32+q34
   +q41+q42+q43;

C res=(pL|pR);
cpmnote(res);
cpmdebug(res);
   // real up to numerical noise
cpmdebug((pL|q12+q13+q14));
   // real up to numerical noise
cpmdebug((pL|q21+q23+q24));
   // real up to numerical noise
cpmdebug((pL|q31+q32+q34));
   // real up to numerical noise
cpmdebug((pL|q41+q42+q43));
   // real up to numerical noise

cpmdebug((p12_3_4|q12));
cpmdebug((p12_3_4|q13));
cpmdebug((p12_3_4|q14));
```

```
    cpmdebug((p13_2_4|q12));
    cpmdebug((p13_2_4|q13));
    cpmdebug((p13_2_4|q14));

    cpmdebug((p14_2_3|q12));
    cpmdebug((p14_2_3|q13));
    cpmdebug((p14_2_3|q14));

    cpmdebug((p23_1_4|q12));
    cpmdebug((p23_1_4|q13));
    cpmdebug((p23_1_4|q14));

    cpmdebug((p24_1_3|q12));
    cpmdebug((p24_1_3|q13));
    cpmdebug((p24_1_3|q14));

    cpmdebug((p34_1_2|q12));
    cpmdebug((p34_1_2|q13));
    cpmdebug((p34_1_2|q14));
}
```

150 cpmqmtemplates.h

```
//? cpmqmtemplates.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_QMTEMPLATES_H_
#define CPM_QMTEMPLATES_H_
/*
   Description: Template classes for constructing quantum systems
      in a manner illustrating the midpoint integrator publication
*/
#include <cpmrvector.h>
#include <cpmc.h>
#include <cpmf.h>
namespace CpmQuantumMechanics{

   using namespace CpmRoot;
   using namespace CpmRootX;
   using CpmArrays::R_Vector;

   template <class Op, class St>
   // Op and St need to implement some of the functions, which
   // classes Observable and State implement. Inspection of the simple
   // code says precisely what these few functions are.
   class Chebyshev{ //The Chebyshev one-step integraton method
      // The Chebyshev one-step integraton method as advocated by
      // Hans De Raedt.
      // As Hans stated: very simple to implement.
      Op a_; // skew-symmetric operator satisfying |a_|<1.
         // Made from the operator under primary consideration
         // by multiplication with a real factor.
      St s_;
         // State, the evolution of which is under consideration
      R norm_;
         // The operator under consideration is a_*norm_
      R eps_; // epsilon
      R sec_; // security factor
      void ini_();
         // initializes norm_ and tests integrity of a_ and s_
   public:
      Chebyshev(Op const& A, St const& s, R sec=1.2, R eps=1e-3):
         a_(A),s_(s),sec_(sec),eps_(eps){ini_();}
         // A should be a skew-symmetric linear operator,
         // and s the state vector on which exp(tA) is to operate.
         // sec and eps are method parametes which control the
         // determination
         // of the operator norm i.e. norm_=|A|. Notice that for the
         // properly initialized operator a_ we have |a_| < 1.
      St step(R t)const;
```

```
         // returns the evolved state exp(t*a_*norm_)*s_
      void step_(R t);
         // changes the state into the evolved one
      St getState()const{ return s_;}
         // returns the state, e.g. for analysis
   };

template <class Op, class St>
void Chebyshev<Op,St>::ini_(){
   static Word loc("Chebyshev<>::ini_()");
   Z mL=3;
   CPM_MA
   const Z iterMax=200;
   Z n=s_.dim();
   St st0(n);
   St st=st0.ranVal();
   R res=-1e20;
   bool work=true;
   Z iter=0;
   while (work){
      iter++;
      st=a_*st;
      R resNew=st.normalize_();
      R diff=CpmRoot::disVal<R>(res,resNew);
      if (diff<eps_){
         work=false;
         res=0.5*(res+resNew);
      }
      else if (iter>iterMax){
         work=false;
         cpmerror(loc&": stopped after "&cpm(iter)&" iterations");
         res=resNew;
      }
      else{
         res=resNew;
      }
   }
   norm_=res*sec_;
   R normInv=cpminv(norm_);
   a_*=normInv;
   CPM_MZ
}

template <class Op, class St>
St Chebyshev<Op,St>::step(R t)const
// needs a fast and accurate implementation of
// the Bessel functions of integer order.
{
   static Word loc("Chebyshev<>::step(R)");
   Z mL=3;
```

```
   CPM_MA
   R tRel=t*norm_;
   R fac=1.4;
     // factor arrises as e/2, bit larger does not hurt
   Z kMax=1+(Z)(fac*cpmabs(tRel));
   St res=s_*besselJ(0,tRel);
   St s0=s_, s1=a_*s_;
   for (Z k=1;k<=kMax;++k){
      res+=s1*(besselJ(k,tRel)*2);
      St temp=s1;
      s1=(a_*temp)*2+s0;
      s0=temp;
   }
   CPM_MZ
   return res;
}

template <class Op, class St>
void Chebyshev<Op,St>::step_(R t)
{
   s_=step(t);
}

//////////////////////////// class MidPoint ////////////////////////////

template < class Op, class St>
   class MidPoint{ //  direct midPoint integraton method
      Op D;
      St psi;
      R t;
      St phi;
   public:
      MidPoint(Op const& D0, St const& psi0, R t0):
         D(D0),psi(psi0),t(t0),phi(D(psi,t)){}

      void step(R dt)
         // changes the state into the evolved one over a small time span
      {
         R tau=dt*0.5;
         t+=tau;
         psi+=phi*tau;
         phi+=D(D(psi,t),t)*dt;
         if (!D.timeIndependent()){
            phi+=D(psi,t+tau)-D(psi,t-tau);
         }
         psi+=phi*tau;
         t+=tau;
         if (D.enforceNormalization()){
            R fac=psi.normalize_();
            R facInv=1./fac;
```

```
      phi*=facInv;
   }
}

St getPsi()const{ return psi;}
St getPhi()const{ return phi;}
   // returns the state, e.g. for analysis
R getTime()const{ return t;}
R getNorm()const{ return psi.absVal();}
C getEnergy()const{ return (psi|phi);}

void revert(){phi*=-1;}

void evolve(R t1, Z n){
   R dt=(t1-t)/n;
   for (Z i=1;i<=n;++i) step(dt);
}

static R testOfComposition(Op const& D0, St const& psi0, R t0,
   R t1, R t2, Z n1, Z n2)
{
   MidPoint m0(D0,psi0,t0);
   m0.evolve(t1,n1);
   St psi1=m0.getPsi();

   MidPoint m1(D0,psi1,t1);
   m1.evolve(t2,n2);
   St psi2= m1.getPsi();

   MidPoint m12(D0,psi0,t0);
   m12.evolve(t2,n1+n2);
   St psi12=m12.getPsi();

   return (psi2-psi12).norm();
}

static R testOfReversibility(Op const& D0, St const& psi0, R t0,
   R t1, Z n)
{
   MidPoint m0(D0,psi0,t0);
   St phi0=m0.getPhi();
   m0.evolve(t1,n);
   m0.evolve(t0,n);
   St psif=m0.getPsi();
   St phif=m0.getPhi();
   return psi0.disVal(psif)+phi0.disVal(phif);
}

static R testOfNormConservation(Op const& D0, St const& psi0, R t0,
 R t1, Z n)
```

```
      {
         MidPoint m0(D0,psi0,t0);
         R n0=psi0.norm();
         m0.evolve(t1,n);
         R nf=m0.getPsi().norm();
         return (nf-n0)/(n0+nf);
      }

      static R testOfConservation(Op const& D0, St const& psi0, R t0,
         R t1, Z n)
      {
         R_Vector epsReal(n);
         R_Vector epsImag(n);
         R_Vector nu(n);
         R_Vector y(n);
         R dt=(t1-t0)/n;
         MidPoint m0(D0,psi0,t0);
         for (Z i=1;i<=n;++i){
            m0.step(dt);
            C vc=m0.getEnergy();
            R vr=m0.getNorm();
            epsReal[i]=vc[1];
            epsImag[i]=vc[2];
            nu[i]=vr*vr;
            y[i]=cpmabs(vc[1])/vc.absVal();
         }
         R diff1=nu.der(dt).disVal(epsReal*2);
         R diff2=nu.disVal(R_Vector(n,1.));
         R diff3=y.sup();
         R diff4=n*epsImag.variation()[2]/epsImag.mean();
         R error=diff1+diff2+diff3;
         if (D0.timeIndependent()) error+=diff4;
         return error;
      }
   };

////////////////////////////// class GMPM ///////////////////////////

template < class Op, class In, class St>
   class GMPM{ // general midpoint method
      // The .
      Op D;
      In f;
      St psi;
      R t;
      St phi;
      static Z method;
   public:
      GMPM(Op const& D0, In const& f0, St const& psi0, R t0):
         D(D0),f(f0),psi(psi0),t(t0),phi(D(psi,t)+f(t))
```

```
   {cpmmessage("GMPM constructed");}

void step(R dt)
   // changes the state into the evolved one over a small time span
{
   R ti=t, tf=t+dt;
   R tau=dt*0.5;
   t+=tau;
   psi+=phi*tau;
   if (method==0)
      phi=(D(psi,t)+f(t))*2-phi; // Leap-frog, not too bad
   else
       phi+=(D(D(psi,t),t)+D(f(t),t))*dt+D(psi,tf)+
         f(tf)-D(psi,ti)-f(ti);
         // full method
      // phi+=D(phi,t)*dt+D(psi,tf)+f(tf)-D(psi,ti)-f(ti);
      // not approved
   psi+=phi*tau;
   t+=tau;
}

void setMethod(Z m){ method=m;}

St getPsi()const{ return psi;}
St getPhi()const{ return phi;}
   // returns the state, e.g. for analysis
R getTime()const{ return t;}
R getNorm()const{ return psi.absVal();}
C getEnergy()const{ return (psi|phi);}

void revert(){phi*=-1;}

void evolve(R t1, Z n){
   R dt=(t1-t)/n;
   for (Z i=1;i<=n;++i) step(dt);
}

static R testOfComposition(Op const& D0, In const& f0,
   St const& psi0, R t0, R t1, R t2, Z n1, Z n2)
{
   GMPM m0(D0,f0,psi0,t0);
   m0.evolve(t1,n1);
   St psi1=m0.getPsi();

   GMPM m1(D0,f0,psi1,t1);
   m1.evolve(t2,n2);
   St psi2= m1.getPsi();

   GMPM m12(D0,f0,psi0,t0);
   m12.evolve(t2,n1+n2);
```

```
   St psi12=m12.getPsi();

   return (psi2-psi12).norm();
}

static R testOfReversibility(Op const& D0, In const& f0,
   St const& psi0, R t0, R t1, Z n)
{
   GMPM m0(D0,f0,psi0,t0);
   St phi0=m0.getPhi();
   m0.evolve(t1,n);
   m0.evolve(t0,n);
   St psif=m0.getPsi();
   St phif=m0.getPhi();
   return psi0.disVal(psif)+phi0.disVal(phif);
}

static R testOfNormConservation(Op const& D0, In const& f0,
   St const& psi0, R t0, R t1, Z n)
{
   GMPM m0(D0,f0,psi0,t0);
   R n0=psi0.norm();
   m0.evolve(t1,n);
   R nf=m0.getPsi().norm();
   return (nf-n0)/(n0+nf);
}

static R testOfConservation(Op const& D0, In const& f0,
   St const& psi0, R t0, R t1, Z n)
{
   R_Vector epsReal(n);
   R_Vector epsImag(n);
   R_Vector nu(n);
   R_Vector y(n);
   R dt=(t1-t0)/n;
   GMPM m0(D0,f0,psi0,t0);
   for (Z i=1;i<=n;++i){
      m0.step(dt);
      C vc=m0.getEnergy();
      R vr=m0.getNorm();
      epsReal[i]=vc[1];
      epsImag[i]=vc[2];
      nu[i]=vr*vr;
      y[i]=cpmabs(vc[1])/vc.absVal();
   }
   R diff1=nu.der(dt).disVal(epsReal*2);
   R diff2=nu.disVal(R_Vector(n,1.));
   R diff3=y.sup();
   R diff4=n*epsImag.variation()[2]/epsImag.mean();
   R error=diff1+diff2+diff3;
```

```
      if (D0.timeIndependent()) error+=diff4;
      return error;
   }

static R testOfOrder(Op const& D0, In const& f0, St const& psi0,
   R t0, R t1, Z n , Z nRuns)
   {
      V< V<St> > res(nRuns, V<St>(n));
      R dt=(t1-t0)/n;
      CyclicAlarm readAlarm(1);
      Z nMem=n, readPeriod=1;
      Z i,j;
      for (Z k=1;k<=nRuns;++k){
         readAlarm.restartWithNewPeriod(readPeriod);
         cpmdebug(readPeriod);
         readPeriod*=2;
         Z ia=1;
         GMPM m0(D0,f0,psi0,t0);
         for (i=1;i<=n;++i){
            m0.step(dt);
            if (readAlarm()){
               res[k][ia]=m0.getPsi();
               ia++;
            }
         }
         n*=2;
         dt*=0.5;
      }
      R_Vector dis(nRuns-1);
      R nMemInv=1./nMem;
      R fac=1;
      for (i=1;i<nRuns;++i){
         R sum=0;
         for (j=1;j<=nMem;++j){
            sum+=(res[nRuns][j]-res[i][j]).absVal();
         }
         dis[i]=sum*nMemInv*fac;
       //  fac*=2;
      }
      Graph gr;
      gr.show(dis);
      cpmwait(20);

      return 1;
   }
};

template < class Op, class In, class St>
Z GMPM<Op,In,St>::method=1;
```

```
} // namespace
#endif
```

151 cpmr2func.h

```
//? cpmr2func.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_R2FUNC_H_
#define CPM_R2FUNC_H_
/*

Description:

   Specialicing the 'function template' Fa<X,Y> to X=RR (RR=C) and Y=R
   'real functions of two variables'. Beyond mere specialization,
   interpolation and visalization is treated

*/

#include <cpmrfunc.h>
#include <cpmc.h>

namespace CpmFunctions{

   using CpmRoot::C;
   using CpmRoot::R;
   using CpmRoot::Rh;
   using CpmRoot::Z;
   using CpmRoot::L;

   typedef C RR;

   // C seems to be the most general and most comfortable description of
   // the Euclidian plane. Notice that angles and rotations can easily be
   // handled via the complex member functions.

//////////////////////////////////// class R2_Func ///////////////////////

class R2_Func: public Fa<RR,R>{
//R-valued functions on RR with arithmetics
   // R-valued functions on RR with arithmetics and trivially defined
   // order properties, and construction from discrete data via
   // interpolation.

public:

// carrying over the constructors from Fa<R,R>

   R2_Func(const F<RR,R>& g):Fa<RR,R>(g){}
      // upcast constructor
```

```
   R2_Func(const Fa<RR,R>& g):Fa<RR,R>(g){}
      // upcast constructor

   R2_Func(const R2_Func& g):Fa<RR,R>(g){}
      // copy constructor

   explicit R2_Func(const R_Func& g);
      // constructor for a function of two variables
      // that is constant with respect to the second
      // argument

   R2_Func(void):Fa<RR,R>(){}
      // default constructor

   explicit   R2_Func( R (*f)(const RR&)):Fa<RR,R>(f){};
      // constructor from native function pointer

   explicit   R2_Func( R (*f)(RR)):Fa<RR,R>(f){};
      // constructor from native function pointer

   explicit   R2_Func( const R& r):Fa<RR,R>(r){};
      // constructor for constant function

// Construction from function pointers

explicit   R2_Func( R (*f)(const R&, const R& ));

explicit   R2_Func( R (*f)(R,R));

// construction from two R_Functions; meaning is tensor product
   R2_Func(const R_Func& f1, const R_Func& f2);

// construction from the data structures used to define images

explicit R2_Func(const CpmArrays::VV<L>& grayImage,
   Z method=1, const Iv& ivx=Iv(0,1), const Iv& ivy=Iv(0,1) );

friend F<RR, CpmArrays::V<R> > toFRRVR(const CpmArrays::VVV<L>& image,
   Z method=1, const Iv& ivx=Iv(0,1), const Iv& ivy=Iv(0,1) );
   // returns a vector-valued function

friend CpmArrays::V<R2_Func> toVR2_Func(const CpmArrays::VVV<L>& image,
   Z method=1, const Iv& ivx=Iv(0,1), const Iv& ivy=Iv(0,1) );
   // returns a vector of real-valued functions

// construction from real matrices

explicit R2_Func(const CpmArrays::VV<R>& matrix,
   Z method=1, const Iv& ivx=Iv(0,1), const Iv& ivy=Iv(0,1),
   bool periodic=true);
```

```
// The resulting function is periodic with the rectangle ivx times ivy
// in RR being the repeating cell for periodic==true. If
// periodic==false, the function is set to 0 outside the rectangle
// ivx times ivy.
// method=0 means pixel-wise representation
// method=1 is a fast simplex interpolation.
// Other values give error.
// The method VV<R> CpmGraphics::Img24::toVVR(R,R,R,Iv) generates a
// suitable real-valued image matrix from an instance of the
// basic C+- image class Img24
// Let matrix be of frame type (m,n). The relation of the function values
// to position is given by the matrix indexing scheme. This means that
// matrix[1][1] is the function value at the upper-left corner,
// matrix[m][n] at the lower-right corner. The matrix values are
// mapped to the grid ivy.centers(m) x ivx.centers(n) in the natural
// manner. So, if the geometry is to remain undistorted (for
// assumed square pixels) one should have e.g. ivx=Iv(0,ar),
// ivy=Iv(0,1) for ar=((R)matrix.dim2())/matrix.dim1().
// If matrix=img.toVV() for some img from Image<>, one has
// ar=img.getAspRat(). For graphically representing the R2_Func on a
// Graph gr, one should put gr.setAspRatHoldHeight(ar) in order to
// see img undistorted.

};

    R2_Func twoDimInt(const CpmArrays::V<R>& ySeq,
        const CpmArrays::V<R_Matrix>& fVal,
        const IpolControl& c=IpolControl());

// We get a function of two real variables by interpolation from a
// discrete set.
// We employ successive one-dimensional interpolation and assume an
// arrangement of the discrete data which allows to do that.
// The assumption is that there exists a finite set of parallel lines
// to the x-axis on which all the points are located for which a function
// value is prescribed. As long as theses lines are parallel, they could
// form any angle with any axis without creating an essentially new
// situation. A completely scattered arrangement of points is something
// considerably different and is not treated here.
// The interpolated function is the return value.
// The arguments of the function describe the discrete interpolation data
// as follows.
// fVal[i][2] is the function value of f at point
// (x,y)=(fVal[i][1],ySeq[i]).
// Thus fVal[i] describes the behaviour of the function on the line, the
// y-coordinate of which is ySeq[i].

   R2_Func indfuncRectangle(const RR& cornerLL, const RR& cornerUR);
      // Creates the geometrical indicator function of a rectangle
      // with lower-left corner cornerLL and upper-right corner cornerUR
```

```
      // Function value is negative in the rectangle area.
      // The absolute value of the function value is the distance from
      // the boundary. So the gradient of the function is the
      // outwards-directed normal vector.

   R2_Func indfuncCircle(const RR& center, R radius);
      // Creates the geometrical indicator function of a circle
      // with given center and radius

   } // namespace

#endif
```

152 cpmr2func.cpp

```
//? cpmr2func.cpp
//? Status of work 2008-10-25.
//?
#include <cpmr2func.h>
#include <cpmvo.h>

using namespace CpmArrays;
using namespace CpmFunctions;

// constructors from functions of two real variables

// auxiliary function objects

namespace{
class R2_FuncAux1: public FncObj<RR,R>{

protected:

   R (*fptr)(const R&, const R& );

public:
   R operator()(const RR& x)const{ return (*fptr)(x[1],x[2]);}

   R2_FuncAux1( R (*g)(const R&, const R&)):fptr(g){}
};

class R2_FuncAux2: public FncObj<RR,R>{

protected:

   R (*fptr)(R,R);

public:
   R operator()(const RR& x)const{ return (*fptr)(x[1],x[2]);}

   R2_FuncAux2( R (*g)(R,R)):fptr(g){}
};

class R2_FuncAux3: public FncObj<RR,R>{

   R_Func f1,f2;

public:
   R operator()(const RR& x)const{ return f1(x[1])*f2(x[2]);}

   R2_FuncAux3(const R_Func f, const R_Func& g):f1(f),f2(g){}
};
```

```
class R2_FuncAux4: public FncObj<RR,R>{

   R_Func f1;

public:
   R operator()(const RR& x)const{ return f1(x[1]);}

   R2_FuncAux4(const R_Func f):f1(f){}
};

}
// main thing

R2_Func::R2_Func( R (*f)(const R&, const R&) ):
Fa<RR,R>(new R2_FuncAux1(f)){}

R2_Func::R2_Func( R (*f)(R,R) ):
Fa<RR,R>(new R2_FuncAux2(f)){}

R2_Func::R2_Func(const R_Func& f1, const R_Func& f2):
Fa<RR,R>(new R2_FuncAux3(f1,f2)){}

R2_Func::R2_Func(const R_Func& f1):
Fa<RR,R>(new R2_FuncAux4(f1)){}


// constructors from matrices and tensors by interpolation, to be
// applied to images

namespace{ // anon

template <class T>
   // automatic conversion from T to R, and R to T has to be available
class FunctionFromMatrix: public CpmFunctions::FncObj<RR,R>{
   //
   VV<T> mat;
   Z m,n,method;
      // important to declare the dependent members m,n later than mat
   Iv ivx,ivy;
   bool periodic;
   static void dec(Z& i){ if (i>1) i--;}
   void inc_i(Z& i)const{ if (i<m) i++;}
   void inc_j(Z& j)const{ if (j<n) j++;}
public:
   R operator()(const RR& x)const;
      // defines values on the whole area. The unit square 0<=x[1]<=1,
      // 0<=x[2]<=1 corresponds directly to the matrix as pixel values.
      // On the rest of the plane the function is defined by periodic
      // repetition
```

```
   FunctionFromMatrix(const VV<T>& matI, Z methodI,
      const Iv& ivxI, const Iv& ivyI,bool per):mat(matI),
      m(mat.dim1()),n(mat.dim2()),method(methodI),
      ivx(ivxI),ivy(ivyI),periodic(per)
   {
      cpmassert( method==1 || method==0,"FunctionFromMatrix(...)");
   }
      // method=1: pixelation
      // method=2: bilinear interpolation
};

template <class T>
R FunctionFromMatrix<T>::operator()(const RR& xx)const
{
   Z mL=4;
   Word loc("FunctionFromMatrix<T>::operator()(RR)");
   cpmmessage(mL,loc&" started, exit not recorded since multiple");
   R x=xx[1];
   R y=xx[2];
   if (!periodic){
      if (!ivx.contains(x) || !ivy.contains(y)) return 0;
   }
   x=ivx.periodic(x);
   y=ivy.periodic(y);
   x*=n;
   y*=m;
      // 0<=x<n, 0<=y<m,
   Z xd=(Z)floor(x);
      // varies from 0 to n-1
   Z yd=(Z)floor(y);
      // varies from 0 to m-1
   Z i=m-yd;   // varies from m to 1
   Z j=xd+1;   // varies from 1 to n
   if (method==0){
       return mat[i][j];
   }
   else if (method==1){
      // unfortunately every case seems to need special consideration:
      // see my pages UM/CTL 98-3-6 in the Folder UM 98, CTL, KAGUM
      x-=xd;
      y-=yd;
      R p1=0,p2=0,f00=0,f01=0,f10=0,f11=0;
      Z iNext=i;
      Z jNext=j;
         // these will be changed by reference by dec and incUM
         // this will happen only once
      if (x==0.5){
         if (y==0.5) return mat[i][j] ;

         if (y>0.5) {
```

```
      f00=mat[i][j];
      dec(iNext);
      f01=mat[iNext][j];
      p2=y-0.5;
   }
   else{ // y<0.5
      inc_i(iNext);
      f00=mat[iNext][j];;
      f01=mat[i][j];
      p2=y+0.5;
   }
   return f00*(1-p2)+f01*p2;
}

if (x>0.5){
   if (y>0.5){  // OK
      dec(iNext);
      inc_j(jNext);
      f00=mat[i][j];
      f10=mat[i][jNext];
      f01=mat[iNext][j];
      f11=mat[iNext][jNext];
      p1=x-0.5;
      p2=y-0.5;
   }
   else if (y<0.5){   // OK
      inc_i(iNext);
      inc_j(jNext);
      f00=mat[iNext][j];
      f10=mat[iNext][jNext];
      f01=mat[i][j];
      f11=mat[i][jNext];
      p1=x-0.5;
      p2=y+0.5;
   }
   else if (y==0){
      inc_j(jNext);
      f00=mat[i][j];
      f10=mat[i][jNext];
      p1=x-0.5;
      return f00*(1-p1)+f10*p1;
   }
}
else{ // then x<0.5, since == was carried to return earlier
   if (y>0.5){  // OK
      dec(iNext);
      dec(jNext);
      f00=mat[i][jNext];
      f10=mat[i][j];
      f01=mat[iNext][jNext];
```

```
            f11=mat[iNext][j];
            p1=x+0.5;
            p2=y-0.5;
         }
         else if (y<0.5){
            inc_i(iNext);
            dec(jNext);
            f00=mat[iNext][jNext];
            f10=mat[iNext][j];
            f01=mat[i][jNext];
            f11=mat[i][j];
            p1=x+0.5;
            p2=y+0.5;
         }
         else if (y==0){
            inc_j(jNext);
            f00=mat[i][j];
            f10=mat[i][jNext];
            p1=x-0.5;
            return f00*(1-p1)+f10*p1;
         }
      }
      R b=f10-f00;
      R c=f01-f00;
      R d=f11-f00-b-c;
      return f00+b*p1+(c+d*p1)*p2;
         // f(x,y):=f00+b*x+c*y+d*x*y has to obey
         // f(0,0)=f00, f(0,1)=f01, f(1,0)=f10, f(1,1)=f11
         // indexes and arguments refer to the (x,y)-system for which
         // lower left corner is (0,0) and right upper corner is (1,1)
   }
   else{
      cpmerror(
         "FunctionFromMatrix<T>::operator(): unvalid parameter method");
      return 0;
   }
}

template <class T>
   // automatic conversion from T to R, and R to T has to be available
class FunctionFromTensor: public FncObj< RR, V<R> >{
   //
protected:
   VVV<T> ten;
   Z l,m,n;
   Iv ivx, ivy;
   Z method;
public:
   V<R> operator()(const RR& x)const;
      // defines values on the whole area. The unit square 0<=x[1]<=1,
```

```
      // 0<=x[2]<=1 corresponds directly to the matrix as pixel values.
      // On the rest of the plane the function is defined by periodic
      // repetition
      FunctionFromTensor(const VVV<T>& tenI,Z methodI,
         const Iv& ivxI, const Iv& ivyI):
         ten(tenI),l(ten.dim1()),m(ten.dim2()), n(ten.dim3()),
         method(methodI), ivx(ivxI), ivy(ivyI){}
      // method=1: pixelation
      // method=2: bilinear interpolation
};

template <class T>
V<R> FunctionFromTensor<T>::operator()(const RR& x)const
{
   V<R> res(l);
   for (Z h=1;h<=l;h++) res[h]=FunctionFromMatrix<L>
      (ten[h],method,ivx,ivy,true)(x);
   return res;
}

} // anon

R2_Func::R2_Func(const VV<L>& mat, Z method,
   const Iv& ivx, const Iv& ivy):
   Fa<RR,R>(new FunctionFromMatrix<L>(mat,method,ivx,ivy,true)){}


R2_Func::R2_Func(const VV<R>& mat, Z method,
   const Iv& ivx, const Iv& ivy, bool per):
Fa<RR,R>(new FunctionFromMatrix<R>(mat,method,ivx,ivy,per)){}

F<RR,V<R> > CpmFunctions::toFRRVR(const VVV<L>& ten, Z method,
   const Iv& ivx, const Iv& ivy)
{
   return
   F<RR,V<R> >(new FunctionFromTensor<L>(ten,method,ivx,ivy));
}

V<R2_Func> CpmFunctions::toVR2_Func(const VVV<L>& ten, Z method,
   const Iv& ivx, const Iv& ivy)
{
   Z mL=3;
   Word loc("CpmFunctions::toVR2_Func(VVV<L>& ...)");
   cpmmessage(mL,loc&" started");
   Z l=ten.dim1();
   V<R2_Func> res(l);
   for (Z h=1;h<=l;h++){
      res[h]=R2_Func(ten[h],method,ivx,ivy);
   }
   cpmmessage(mL,loc&" done");
```

```
   return res;
}

namespace{

R twoDimensiolalInterpolation(R x, R y, const V<R>& ySeq,
    const V<R_Func>& fSeq, const IpolControl& c)
// Write ySeq as y1,...yn. Then we consider the functions fSeq obtained
// from a two-variable function f by
// fi(x,y) = f(x,yi). (Restriction of f on lines which are parallel to
// the x-axis). The present function tries to reconstruct f as
// f(x,y) = twoDimensiolalInterpolation(x, y, (y1,...yn), (f1,...fn)).
// The parameters m and method specify the one-dimensional interpolation
// method used in the implementation.
{
    Z n=ySeq.dim();
    V<R> fVal(n);
    for (Z i=1;i<=n;i++) fVal[i]=(fSeq[i])(x);
    return R_Func(ySeq,fVal,c)(y);
        // interpolation in y-direction
}

R twoDimIntFunc(const RR& xy, const V<R>& ySeq, const V<R_Matrix>& fVal,
    const IpolControl& c)
// Transforming value arrays into function arrays.

{
    R x=xy[1];
    R y=xy[2];
    Z i,n=ySeq.dim();
    V<R_Func> fSeq(n);
    R_Vector xv,yv;
    for (i=1;i<=n;i++){
        xv=fVal[i][1];
        yv=fVal[i][2];
        fSeq[i]=R_Func(xv,yv,c);
            // function list from data list
    }
    return twoDimensiolalInterpolation(x, y, ySeq, fSeq, c);
}

R  indRectangle(const RR& z, const R& xL, const R& xU, const R& yL,
   const R& yU )
// the return value is the distance from z to the boundary of the
// rectangle positive if the point is outside and negative if it is
// inside.
{
   RR center(0.5*(xL+xU),0.5*(yL+yU));
   R a=0.5*(xU-xL);
   R b=0.5*(yU-yL);
```

```
   cpmassert(a>0 && b>0,"indRectangle(...)");
   RR zRel=z-center;
   R x=zRel[1];
   R y=zRel[2];
// We go through the various cases with imediate return values. This
// avoids nested conditions
   if (x>=a){
      if (y>b) return (R)(zRel-RR(a,b)).absVal();
         // then the corner RR(a,b) is the boundary point nearest to z
      if (y<-b) return (R)(zRel-RR(a,-b)).absVal();
         // then the corner RR(a,-b) is the boundary point nearest to z
      return x-a;
         // then the line x=a is the boundary segment nearest to z
   }
   if (x<=-a){
      if (y>b) return (R)(zRel-RR(-a,b)).absVal();
      if (y<-b) return (R)(zRel-RR(-a,-b)).absVal();
      return -x-a;
   }
   // now -a<x<a
   if (y>=b) return y-b;
   if (y<=-b) return -y-b;
   // now -b<y<b
// Now (x,y) fulfills -a<x<a, -b<y<b. It is inside the rectangle
// all distances of (x,y) from the walls of the rectangle (all positive)
   Vo<R> dxy(4,0);
   dxy[1]=a-x;
   dxy[2]=x+a;
   dxy[3]=b-y;
   dxy[4]=y+b;
   dxy.order();
// The distance to the wall is the smallest of these values and returned
// with a minus sign since we are inside the area
   return -dxy[1];
}

R indCircle(const RR& z, const R& xC, const R& yC, const R& r )
{
   return (R)((z-RR(xC,yC)).absVal()-r);
}


} // namespace

R2_Func CpmFunctions::indfuncRectangle(const RR& cornerLL,
   const RR& cornerUR)
{
   R xL=cornerLL[1];
   R xU=cornerUR[1];
   R yL=cornerLL[2];
```

```
   R yU=cornerUR[2];
   return F4<RR,R,R,R,R,R>(xL,xU,yL,yU)(indRectangle);
}


R2_Func CpmFunctions::indfuncCircle(const RR& center, R radius)
{
   R xC=center[1];
   R yC=center[2];
   return F3<RR,R,R,R,R>(xC,yC,radius)(indCircle);
}

R2_Func CpmFunctions::twoDimInt(const V<R>& ySeq,
   const V<R_Matrix>& fVal, const IpolControl& control)
{
    Z i,d,n=ySeq.dim();
    cpmassert(n==fVal.dim(),"twoDimInt(...)");
    for (i=1;i<=n;i++){
        d=(fVal[i]).dim1();
        if (d!=2) cpmerror(
        "twoDimensiolalInterpolation(): dimension mismatch in value\
 list");
    }
    return F3<RR,V<R>,V<R_Matrix>,IpolControl,R>
       (ySeq,fVal,control)(twoDimIntFunc);
}
```

153 cpmrecord.h

```
//? cpmrecord.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_RECORD_H_
#define CPM_RECORD_H_
/*
Record is a data structure capable of holding data of types bool, Z, R,
Word and accessing those data via a section and an item.
The main constructor for this class is one which reads all
the data from file. This file has to follow formatting
rules very similar to those for Windows ini-files.
The main difference is that we need an explicit type
indicator at the beginning of each value defining line.
Also, a section is a single word (that may contain spaces)
on a line. No '[' and ']' is needed.
*/
#include <cpmm.h>
   // powerful tool to achieve most of the functionality
#include <cpmv.h>
#include <cpmword.h>
#include <cpmx.h>
#include <cpmtypes.h>

namespace CpmRootX{

   using namespace CpmStd;

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::Word;

   using CpmArrays::V;
   using CpmArrays::X2;
   using CpmArrays::M;
   using CpmArrays::S;

//////////////////////////// class SecCol ////////////////////////////
// 'Section Collector'
// Collects lines from various files and places them consistently
// in sections which are defined by headlines. (There are no
// headlines of different hierarchical level). See comment to
// SecCol::str_ for details.
// Tool for reading ('parsing') the most recent version of ini-Files
// which allow as data entries the names of other ini-files
// which will be used as data sources just as the one appearing as an
// argument of the main constructor.
// assume a file we want to use is named ".\\iniFiles\\aux.ini"
```

```
// then the corresponding entry in the basic ini file is
//
/*
F ./iniFiles/aux.ini

  or

include ./iniFiles/aux.ini
*/
// where F tries to suggest the usage of File as a data type
// and the word 'include' is familiar from C programming.
// File names that include whitespace cannot be used.

// The result of parsing is the well organized single string
// str_, which can be accessed via function str()
// and finally by Record::readFormatted(...)

// Used to implement
// bool Record::readFromFile(const Word& fileName, bool binary);
// which in turn implements
// RecordHandler::RecordHandler(const Word& file=Word(), bool
// verbose=true);
// and
// IniFileBasedApplication::IniFileBasedApplication(const Word& prog).

class Record;
namespace aux{
   class SecCol{// section collector,

   B binary;
      // file open mode, true and false seem to work equally well
      // on my present Windows system

   string el;
      // line terminator ('end of line'), may need to depend on the
      // previous parameter

   S<Word> fns;
      // file names, initialized as void set

   Z1 nFiles_;
      // total number of files which was used to create str_

   string str_;
      // the content of all contributing files one string
      // each data line is (somewhere) preceeded by its
      // section head-line. Sections are ordered (according
      // to the default order of class string) by their
      // section head-lines.
      // Each line which contains a single (non-whitespace-containing)
```

```
      // string is (by definition) a section head-line.
      // The detailed structure of the data entries in ini-files
      // does not matter for the present class.

   bool ini();
      // main work to implement the main constructor

   B valid;
      // initialized as false !

public:

   Word nameOf()const{ return Word("SecCol");}

   explicit SecCol(const Word& fileName, bool binaryI=true);
      // raison d'être

   SecCol(){}
      // useful ???

   const string& str()const{ return str_;}
      // raison d'être

   Z nFiles()const{ return nFiles_;}
      // for information, may help to see whether all intended files
      // were found

   bool isVal()const{ return valid;}
};

//////////////////////// class Key ////////////////////////////////////

class Key: public X2<Word,Word>{
      // now IO and ORDER from X2
      void normalize(){ c1().skipSurSpc_(); c2().skipSurSpc_();}
   public:
      typedef X2<Word,Word> Base;
      typedef Key Type;
      Key():Base(){}
      Key(const Word& w1, const Word& w2):Base(w1,w2){normalize();}
         // assures that space and whitespace (such as carriage return
         // and line feed do not shine up in the keys even if
         // written to and read from file without too much care.
         // This is important since identity of keys is required for
         // functional identity.
   };

//////////////////////// class TextH ////////////////////////////////

   class TextH{
```

```
      // Text with special header of each line; too special
      // to justify the general name Text. Should grow into a general
      // class Text after some experience
      V<string> vs;
      typedef TextH Type;
   public:
      CPM_IO
      Word nameOf()const{ return Word("TextH");}
      friend class Record;
      TextH():vs(0){}
         // notice that the default constructor of V creates
         // vectors of dimension 1. Here it is important to
         // get a truely void text
      explicit TextH(const string& s):vs(1,s){}
      explicit TextH(Word const& w):vs(1,w.str()){}
      void operator &=(const string& s){ vs&=s;}
      void operator &=(const TextH& t){ vs&=t.vs;}
      void operator &=(Word const& line){ vs&=line.str();}
   };

} // aux

class Text{// texts as lists of Words
      V<Word> vw_;
      explicit Text(V<Word> const& vw):vw_(vw){}
      typedef Text Type;
   public:
      CPM_IO
      Word nameOf()const{ return Word("Text");}
      Text():vw_(0){}
      explicit Text(Word const& w):vw_(1,w){}
      explicit Text(string const& s):vw_(1,Word(s)){}
      static Text getLine(istream& in);
      V<Word> const& rep()const{ return vw_;}
   };

//////////////////////// class Record ////////////////////////////////

   class Record{ // data structure for communication with ini-files
      // and other input mechanisms to complex programs and functions. The
      // idea is, that all input data for defining the conditions for
      // a program run should be collectable in a Record.
      // We do not follow the self-suggesting idea to hold in Record
      // only a string representation of all data, together with a type
      // indicator. Instead we like to have all data in their
      // 'native' binary format.
      // A sufficienly general set of types for this usage turned out to
      // be:
      // bool, Z, R, Word, V<bool>, V<Z>, V<R>, V<Word>
      // Reading and writing data of these types is supported by
```

```
      // functions set() and get()
      // Usage and motivation for the V<>-types is explained in the
      // comment to
      //    explicit Record(const Word& fn);

   typedef M<aux::Key,B>            m0;
   typedef M<aux::Key,V<B> >        v0;

   typedef M<aux::Key,Z1>           m1;
   typedef M<aux::Key,V<Z1> >       v1;

   typedef M<aux::Key,R1>           m2;
   typedef M<aux::Key,V<R1> >       v2;

   typedef M<aux::Key,Word>         m3;
   typedef M<aux::Key,V<Word> >     v3;

   typedef M<aux::Key,aux::TextH>   m4;
   typedef M<Word,aux::TextH>       m5;
   typedef aux::TextH               m6;

   m0 rep0;
      // 'list' of names, sections, and values of B-valued variables
   v0 vec0;
      // 'list' of names, sections, and values of B-valued lists

   m1 rep1;
      // 'list' of names, sections, and values of Z1-valued variables
   v1 vec1;
      // 'list' of names, sections, and values of Z1-valued lists

   m2 rep2;
      // 'list' of names, sections, and values of R1-valued variables
   v2 vec2;
      // 'list' of names, sections, and values of R1-valued lists

   m3 rep3;
      // 'list' of names, sections, and values of Word-valued variables
   v3 vec3;
      // 'list' of names, sections, and values of Word-valued lists

// Handling comments (do work well after a change in implementation
// that eliminats all CR symbols from comments (2004-09)

   m4 rep4;
      // holds the comments to section-item pairs
   m5 rep5;
      // holds the comments to the sections
   m6 rep6;
      // comment for the whole Record *this
```

```
 B valid;
    // initialized as false !

 public:
 typedef Record Type;

 static bool startsEntry(Word const& w)
    // returns true if w is one of the type indicating words,
    // which, by definition, preceed a value entry.
    // Since 2004-10-12 all non-comment lines for which
    // the first Word w1 says startsEntry(w)==false is
    // interpreted as a section title. Thus such a title may
    // be a chain of whitespace separated contiguous Words.
    // This allows using unmodified plain language
    // in complicated section titles e.g.
    // 'kinetic energy' instead of 'kinetic_energy' or 'eKin'
 {
    return
    w=="F"  ||
    w=="c"  ||
    w=="B"  ||
    w=="Bs" ||
    w=="Z"  ||
    w=="Zs" ||
    w=="R"  ||
    w=="Rs" ||
    w=="W"  ||
    w=="Ws"  ;
 }

 CPM_IO
// no CPM_ORDER since this cannot be implemented in an efficient
// manner with the given data members. An efficient implementation
// will be given in class Recordable.

Word nameOf()const{ return Word("Record");}

 bool writeFormatted(ostream&, bool writeComments=false)const;
    // writes the content of *this (which, of cource, is being
    // held in the attributes rep0, ..., rep6) in such a
    // way to a stream, that the following read function can
    // recover it from there. Whether 'c'-comments (that tend to
    // become lengthy in my applications) will be written
    // or not is controlled by the second argument.
 bool readFormatted(istream&);
    // reading from a file that follows the syntax for my ini-files
    // Also writes to cpmcerr what it found
 bool readFromFile(const Word& fileName, bool binary);
```

```
   Record(){};
      // this creates a void record. Data are inserted via set. e.g.
      // Record rc;
      // R v=1000, J=1.8, el=30000, hVal=200000;
      // Z verb=2;
      // bool nog=false;
      // Word conf("config1");
      // V<R> waitTimes("",1.1,2.2,5,5);
      // rc.set("Electricity","V",v);
      // rc.set("Magnetism","J",J);
      // rc.set("Electricity","E",el);
      // rc.set("Magnetism","H",hVal);
      // rc.set("General","data",conf);
      // rc.set("General","verbose",verb);
      // rc.set("General","noGraphics",nog);
      // rc.set("General","waitTimes",waitTimes);

      // These set() statement create the following entries in the
      // file by rc.writeFormatted(out), where the stream out is given by
      // cpmofstream(out,filename)
/*
Electricity
R E=30000.
R V=1000.
General
W data=config1
B noGraphics=false
Z verbose=2
Rs waitTimes=1.1 2.2 5. 5.
Magnetism
R J=1.8
R H=20000
*/
   // A even more convenient way of defining a Record is by reading it
   // from a textfile:
   //
   // Record rc("input.ini");
   //
   // The previously displayed
   // data (between /********... and ...*****/) have the matching format
   // for this reading function. In addition such a file may contain
   // ignored lines starting with '/' or ';' or '*'.
   //
   // Lines starting with 'c' (optionally preceeded by some white space)
   // are constituting elements of the Record and are, depending on their
   // position in the file comments to the whole file, or to a section,
   // or to an item within a section. The function writeFormatted

   explicit Record(const Word& fn);
```

```
// bool, Z, R, Word are chosen as the basic data types for controlling
// programs. Since we are dealing with the interface of a program with
// the outside world, program specific user defined types and
// mathematical abstractions such as complex numbers, should not be
// considered here. However, it turned out that lists of the types
// mentioned above occur often, also under circumstances where
// the length of the list has to be changed often so that an independent
// entry for the number of list members is not convenient. This new
// extension now allows entries
//
/*
example
   Rs xl=137 3.14 1.4142
   R  x=1.7321
      // single R
   Rs y=3.14
      // V<R> with only one component
   Bs bl=true false false true
   Ws a=eins zwei drei
   Zs b=1 2 3 137
*/
// The correct reading of these quantities is
/*
   Word sec="example";
   V<R> xl,y;
   R x;
   V<Z> b;
   V<Word> a;
   V<bool> bl;
   rc.get(sec,"xl",xl);
   rc.get(sec,"y",y);
   rc.get(sec,"x",x);
   rc.get(sec,"bl",bl);
   rc.get(sec,"a",a);
   rc.get(sec,"b",b);
*/
// Reading the fundamental basic types from a position determined by
// section and key. A failure in doing so never stops the
// program. For pedantic!=0 diagnostic messages will be left on file
// cpmcerr.txt.
// It is only for return value==true that argument value may get changed.

   bool get(const Word& section, const Word& item, bool& value,
      Z pedantic=1)const;
   bool get(const Word& section, const Word& item, V<bool>& value,
      Z pedantic=1)const;

   bool get(const Word& section, const Word& item, Z& value,
      Z pedantic=1)const;
   bool get(const Word& section, const Word& item, V<Z>& value,
```

```
      Z pedantic=1)const;

   bool get(const Word& section, const Word& item, R& value,
      Z pedantic=1)const;
   bool get(const Word& section, const Word& item, V<R>& value,
      Z pedantic=1)const;

   bool get(const Word& section, const Word& item, Word& value,
      Z pedantic=1)const;
   bool get(const Word& section, const Word& item, V<Word>& value,
      Z pedantic=1)const;

// new fast lean access to data, return the return classes default value
// if there is no matching entry. For data that need a conversion, the
// implementation is not inline, see cpmrecord.cpp

// constant get functions
   bool getB(Word const& section, Word const& item)const
   { return rep0[aux::Key(section,item)];}
   V<bool> getBs(Word const& section, Word const& item)const;
   Z getZ(Word const& section, Word const& item)const
   { return rep1[aux::Key(section,item)];}
   V<Z> getZs(Word const& section, Word const& item)const;
   R getR(Word const& section, Word const& item)const
   { return rep2[aux::Key(section,item)];}
   V<R> getRs(Word const& section, Word const& item)const;
   Word getW(Word const& section, Word const& item)const
   { return rep3[aux::Key(section,item)];}
   V<Word> getWs(Word const& section, Word const& item)const
   { return vec3[aux::Key(section,item)];}
// mutating set functions
   void setB(Word const& section, Word const& item, bool x)
   { rep0[aux::Key(section,item)]=B(x);}

   void setBs(Word const& section, Word const& item, V<bool> const& x);

   void setZ(Word const& section, Word const& item, Z x)
   { rep1[aux::Key(section,item)]=Z1(x);}

   void setZs(Word const& section, Word const& item, V<Z> const& x);

   void setR(Word const& section, Word const& item, R x)
   { rep2[aux::Key(section,item)]=R1(x);}

   void setRs(Word const& section, Word const& item, V<R> const& x);

   void setW(Word const& section, Word const& item, Word const& x)
   { rep3[aux::Key(section,item)]=x;}

   void setWs(Word const& section, Word const& item, V<Word> const& x)
```

```
   { vec3[aux::Key(section,item)]=x;}
// end of new lean access

// Handling comments
   bool get(aux::TextH& comment)const;
      // getting a text ment as a comment to the whole Record
   bool get(const Word& section, aux::TextH& comment)const;
      // getting a text ment as a comment to a section
   bool get(const Word& section, const Word& item,
      aux::TextH& comment)const;
      // getting a text ment as a comment to a section-item pair

// setting the value at a position determined by section and item.
// If section or item was not yet created earlier they become installed
// now. The return value is true for success and false for failure.
// In the present implementation success seems to be guaranteed
// and we return true in any case.

   bool set(const Word& section, const Word& item, const bool& value);
   bool set(const Word& section, const Word& item, const V<bool>& value);
   bool set(const Word& section, const Word& item, const V<B>& value);

   bool set(const Word& section, const Word& item, const Z& value);
   bool set(const Word& section, const Word& item, const V<Z>& value);
   bool set(const Word& section, const Word& item, const V<Z1>& value);

   bool set(const Word& section, const Word& item, const R& value);
   bool set(const Word& section, const Word& item, const V<R>& value);
   bool set(const Word& section, const Word& item, const V<R1>& value);

   bool set(const Word& section, const Word& item, const Word& value);
   bool set(const Word& section, const Word& item, const V<Word>& value);

// Handling comments
   bool set(const aux::TextH& comment);
      // introducing a text as a comment to the whole Record
   bool set(const Word& section, const aux::TextH& comment);
      // introducing a text as a comment to a section
   bool set(const Word& section, const Word& item,
      const aux::TextH& comment);
      // introducing a text as a comment to a section-item pair

   bool isVoid()const
      { return (rep0.isVoid() && rep1.isVoid()
         && rep2.isVoid() && rep3.isVoid()&&vec3.isVoid());}

   bool isVal()const{ return valid;}

   Record operator()(Word const& sec)const;
      // returns all entries of the section sec of *this
```

```
      // as a single-section record which is endowed with a
      // suitable overall comment.

   Record operator()(Word const& sec1, Word const& sec2)const;
      // returns all entries of the section sec1 of *this
      // as a single-section record the single section of which
      // is named sec2.

   M<aux::Key,V<B> > getBs()const{ return vec0;}
   M<aux::Key,V<Z1> > getZs()const{ return vec1;}
   M<aux::Key,V<R1> > getRs()const{ return vec2;}
   M<aux::Key,V<Word> > getWs()const{ return vec3;}
};

//////////////////// class Recordable ////////////////////////////
/*
This class arrose as an abstraction from my first implementation
of class R_FuncRec, which now is implemented as being derived
from Recordable. One could even carry the abstraction one step
further and make class Record a template argument.

This class uses Record to hold the complete state information
and implements the CPM_IO and CPM_ORDER interfaces based on
this.

One could consider to implement Recordable as a class derived from
Record. I prefer not to do this in order to be free in the
implementation of CPM_IO and CPM_ORDER which should here be based
on Record::readFormatted and Record::writeFormatted
instead of Record::scanFrom and Record::prnOn.
*/

class Recordable{// deriving from this class provides IO for complex data

   mutable bool valid_;
   mutable Word rep_;
      // this is a text representation of the content on which
      // the order operations are based. This is a dependent
      // quantity that will be set with function adjust().
      // Notice that stream related operations work with rc_.
protected:

   Record rc_;
      // this is -by definition- the only independent
      // information that makes up the state of an instance
      // of the class.

   virtual bool vadjust()const{ return true;}
      // setting all additional (dependent)
      // data members if rc_ is given
```

```
      // returns true if it was successful. For enabling
      // the definition of those functions as const which
      // are expected to be constant, all dependent data
      // members to be added in derived classes have to
      // be declared mutable (just as the attribute 'valid' above).
      // See R_FuncRec::vadjust() for a typical redefinition
      // of this function.

   bool adjust()const;
      // sets valid_, rep_, and (by calling vadjust) all
      // additional dependent data members
      // so that they are in accordance with the data in
      // rc_. If valid_==true at calling the function,
      // the call has no effect and is not expensive.
      // The function is const in the sense that it does not change
      // the real content. It does, however change the representation
      // of this content, which is possible due to mutable data
      // members

   void renew_(){ valid_=false;}
      // setting valid_ to false makes the next call of
      // adjust busy. Any function in a derived class which
      // changes rc_ (which is possible since we chose to define
      // rc_ as protected) has to call renew_ in order to
      // let adjust do the necessary adjustments.

   typedef Recordable Type;

public:

   CPM_IO
   CPM_ORDER
   virtual Word nameOf()const{ return Word("Recordable");}

   explicit Recordable(Record const& rc=Record()):rc_(rc),valid_(false){}
      // if the constructor would be allowed to call a virtual function
      // one could form a valid instance already here. Things being as
      // they are enforce that adjust can be called only in calls
      // of member functions. See class R_FuncRec (cpmrfuncrec.h)
      // as an example for this mechanism

   Recordable(Record const& rc, Word const& sec):rc_(rc(sec)){}
      // generating a Recordable from one section of a Record

   Record rc()const{ return rc_;}

// access to Boolean content

   void setB(Word const& section, Word const& name, bool x)
      { rc_.setB(section,name,x); renew_();}
```

```
   bool getB(Word const& section, Word const& name)const
      { return rc_.getB(section,name);}

   void setBs(Word const& section, Word const& name, V<bool> const& x)
      { rc_.setBs(section,name,x); renew_();}

   V<bool> getBs(Word const& section, Word const& name)const
      { return rc_.getBs(section,name);}

// access to integer content

   void setZ(Word const& section, Word const& name, Z x)
      { rc_.setZ(section,name,x); renew_();}

   Z getZ(Word const& section, Word const& name)const
      { return rc_.getZ(section,name);}

   void setZs(Word const& section, Word const& name, V<Z> const& x)
      { rc_.setZs(section,name,x); renew_();}

   V<Z> getZs(Word const& section, Word const& name)const
      { return rc_.getZs(section,name);}

// access to real content

   void setR(Word const& section, Word const& name, R x)
      { rc_.setR(section,name,x); renew_();}

   R getR(Word const& section, Word const& name)const
      { return rc_.getR(section,name);}

   void setRs(Word const& section, Word const& name, V<R> const& x)
      { rc_.setRs(section,name,x); renew_();}

   V<R> getRs(Word const& section, Word const& name)const
      { return rc_.getRs(section,name);}

// access to Word content

   void setW(Word const& section, Word const& name, Word const& x)
      { rc_.setW(section,name,x); renew_();}

   Word getW(Word const& section, Word const& name)const
      { return rc_.getW(section,name);}

   void setWs(Word const& section, Word const& name, V<Word> const& x)
      { rc_.setWs(section,name,x); renew_();}

   V<Word> getWs(Word const& section, Word const& name)const
```

```
      { return rc_.getWs(section,name);}

// access to rep_
   Word rep()const // needs to update rep_
   {
      if( adjust())return rep_;
      else {
         cpmerror("Recordable::rep() failed");
         return Word();
      }
   }
};

} // namespace

#endif
```

154 cpmrecord.cpp

```
//? cpmrecord.cpp
//? Status of work 2008-10-25.
//?
#include <vector>

#include <cpmrecord.h>
#include <cpmsystem.h>
#include <cpmgreg.h>

using std::vector;
using std::istringstream;
using std::ostringstream;

using CpmRoot::Word;
using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::toStr;
using CpmRootX::B;
using CpmRootX::Z1;
using CpmRootX::R1;
using CpmRootX::Record;
using CpmRootX::Recordable;
using CpmRootX::Text;
using CpmRootX::aux::Key;
using CpmRootX::aux::TextH;
using CpmRootX::aux::SecCol;
using CpmArrays::V;
using CpmArrays::S;
using CpmArrays::M;
using CpmArrays::X2;
using CpmFunctions::F;
using CpmFunctions::F1;
using CpmSystem::Message;
using CpmSystem::IFileStream;
using CpmSystem::OFileStream;
using namespace CpmStd;

///////////////// class SecCol ////////////////////////

CpmRootX::aux::SecCol::SecCol(const Word& fileName, bool binaryI):
binary(binaryI),nFiles_(1)
{
   el=binary ? "\r\n" : "\n";
      // works, but with binary replaced by !binary
      // it works too at least in some contexts
   fns.add(fileName);
   valid=B(ini());
```

```
}

namespace{

   bool skipHead(string& str)
      // returns true if a leading "c " (named head) was found.
      // after call, str does no longer contain the head.
      // This will be applied only for a str which already is
      // freed from leading white space
   {
      Z n=(Z)str.size();
      if (n<3){
         return false;
      }
      char c1=str[0];
      char c2=str[1];
      if (c1=='c' && c2==' '){
         str.erase(0,2);
         return true;
      }
      else{
         return false;
      }
   }

   bool replaceFillCharacters_(string& str)
      // defining replacements that are OK for non-comment lines
   {
      Z nl=(Z)str.size();
      bool res=false;
      char rep=' '; // rep=replace
      for (Z i=0;i<nl;++i){
         char c=str[i];
         if (c=='='|| c==',' || c==';'){
            str[i]=rep;
            if (c==',' || c==';') res=true;
         }
      }
      return res;
   }

   bool isInclude(const string& line, Word& fnFound)
      // returns true if line imports a file name
   {
      Word w1("include");
      Word w2("F");
      Word l(line);
      Word fw=l.firstWord_();
      if (fw==w1 || fw==w2){
         fnFound=l.firstWord_();
```

```
         fnFound=Message::ext(fnFound); // 2008-04-21
         return true;
      }
      else{
         return false;
      }
   }

   bool ignore(const string& line)
      // returns true if line is to be ignored
   {
      Word l(line);
      Word w1=l.firstWord_();
      return w1.length()==0 || CpmRoot::startsComment(w1[1]);
   }

   bool isSectionHeader(const string& line, Word& headerFound )
      // returns true if line imports a section heading
      // notice that lines startinf with '/' or ';' or ....
      // will never shine up as 'line'
   {
      Word l(line);
      Word w1=l.firstWord();
      bool res=!Record::startsEntry(w1);
      if (res==true){
         l.skipSurSpc_();
         headerFound=l;
      }
      return res;
   }
}

bool CpmRootX::aux::SecCol::ini()
{
   Z mL=2;
   static Word loc("CpmRootX::SecCol::ini()");
   CPM_MA
   M<string,string> sections;
   bool safe=false;
   while (!fns.isVoid()){ // work through all files to which the
      // primary file refers and that may get referred to in such
      // files
      Word fn=fns.first();
      IFileStream fs(fn,safe,binary);
      if (!fs.isVal()){
         cpmwarning(loc&" file stream unvalid");
         CPM_MZ
         return false;
      }
      Word activeHeader;
```

```
      Word nextFile;
      string line;

      while ( std::getline(fs(),line) ){
         if (ignore(line)){
            ;
         }
         else{
            replaceFillCharacters_(line);
               // this means also eliminating ';' which
               // could turn comment-lines into non-comment lines
               // if applied to comment lines; in the present
               // place we are sure not to make this mistake
            if (isInclude(line,nextFile)){
               fns.add(nextFile);
               nFiles_+=1;
               // a new action item for the outer while loop
            }
            else{
               string lineToAdd=line+el;
               bool b=isSectionHeader(line,activeHeader);
               // find the correct section to which line should
               // be added
               if (!b){
                  string key=activeHeader.str();
                  sections[key]+=lineToAdd;
               }
            }
         }
      }
      fns.remove(fn);
         // fn is worked through
   }
   Z ns=sections.card();
   for (Z i=1;i<=ns;i++){
      X2<string,string> xyi=sections.xy(i);
      string xi=xyi.c1();
      string yi=xyi.c2();
      str_+=el;
      str_+=xi;
         // section header
      str_+=el;
      str_+=el;
      str_+=yi;
         // section content
   }
   CPM_MZ
   return true;
}
```

```
//////////////////////// class TextH ////////////////////////////////////////

namespace{
   string skipCR(const string& w)
   {
      string::size_type nCR=w.find(0x0D);
      string res=w;
      return nCR==string::npos ? res : res.replace(nCR,1,"");
   }
}

bool TextH::prnOn(ostream& str)const
{
   const string head("  c ");
   if (!str) return false;
   Z n=vs.dim();
   if (n>=1){
      for (Z i=1;i<=n;i++){
         str<<endl<<head<<skipCR(vs[i]);
         if (!str) return false;
      }
      str<<endl;
   }
   return true;
}

bool TextH::scanFrom(istream& str)
{
   if (!str) return false;
   using CpmRoot::readLine;
   string line;
   while (readLine(line,str)){
      *this&=line;
   }
   return true;
}

//////////////////////// class Record ////////////////////////////////////////

Record::Record(const Word& fn)
{
   Record rc;
   IFileStream in(fn);
   bool bv=rc.readFormatted(in());
   *this=rc;
   valid=bv;
}

bool Record::set(const Word& section, const Word& item,
   const bool& value)
```

```
{
   B val(value);
   Key k(section,item);
   rep0.set(k,val);
   return true;
      // no non-mystical error condition can happen
}

namespace{
B f_bool(bool x){ return B(x);}
Z1 f_Z(Z x){ return Z1(x);}
R1 f_R(R x){ return R1(x);}

bool g_bool(B x){ return x.x1;}
Z g_Z(Z1 x){ return x.x1;}
R g_R(R1 x){ return x.x1;}
}

bool Record::set(const Word& section, const Word& item,
   const V<bool>& value)
{
   Key k(section,item);
   vec0.set(k,value(F<bool,B>(f_bool)));
      // see V<>::operator()(const F<T,X>& g)const
   return true;
}

bool Record::set(const Word& section, const Word& item,
   const V<B>& value)
{
   Key k(section,item);
   vec0.set(k,value);
   return true;
}

bool Record::set(const Word& section, const Word& item, const Z& value)
{
   Z1 val(value);
   Key k(section,item);
   rep1.set(k,val);
   return true;
}

bool Record::set(const Word& section, const Word& item,
   const V<Z>& value)
{
   Key k(section,item);
   vec1.set(k,value(F<Z,Z1>(f_Z)));
   return true;
}
```

```
bool Record::set(const Word& section, const Word& item,
   const V<Z1>& value)
{
   Key k(section,item);
   vec1.set(k,value);
   return true;
}

bool Record::set(const Word& section, const Word& item, const R& value)
{
   R1 val(value);
   Key k(section,item);
   rep2.set(k,val);
   return true;
}

bool Record::set(const Word& section, const Word& item,
   const V<R>& value)
{
   Key k(section,item);
   vec2.set(k,value(F<R,R1>(f_R)));
   return true;
}

bool Record::set(const Word& section, const Word& item,
   const V<R1>& value)
{
   Key k(section,item);
   vec2.set(k,value);
   return true;
}

bool Record::set(const Word& section, const Word& item,
   const Word& value)
{
   Key k(section,item);
   rep3.set(k,value);
   return true;
}

bool Record::set(const Word& section, const Word& item,
   const V<Word>& value)
{
   Key k(section,item);
   vec3.set(k,value);
   return true;
}

bool Record::set(const Word& section, const Word& item,
```

```
   const TextH& value)
{
   Key k(section,item);
   rep4.set(k,value);
   return true;
}

bool Record::set(const Word& section, const TextH& value)
{
   Word sectionAct=section;
   sectionAct.skipSurSpc_();
      // notice that using this function with 'section' read from file
      // after another function has it written to file may result in
      // having the argument cluttered with non-printable characters.
      // So a Word may loose its power as a key
   rep5.set(sectionAct,value);
   return true;
}

bool Record::set(const TextH& value)
{
   rep6=value;
   return true;
}

void Record::setBs(Word const& section, Word const& item,
   V<bool> const& x)
 { vec0[Key(section,item)]=x(F<bool,B>(f_bool));}

void Record::setZs(Word const& section, Word const& item, V<Z> const& x)
{ vec1[Key(section,item)]=x(F<Z,Z1>(f_Z));}

void Record::setRs(Word const& section, Word const& item, V<R> const& x)
{ vec2[Key(section,item)]=x(F<R,R1>(f_R));}

bool Record::get(const Word& section, const Word& item, bool& value,
   Z verb)const
{
   Key k(section,item);
   B val(value);
   bool b=rep0.get(k,val);
   if (b) value=val;
   else if (verb>0){
      cpmwarning(
         "section="&section&", item="&item&", not able to read bool");
   }
   return b;
}

bool Record::get(const Word& section, const Word& item, Z& value,
```

```
   Z verb)const
{
   Key k(section,item);
   Z1 val(value);
   bool b=rep1.get(k,val);
   if (b) value=val;
   else if (verb>0){
      cpmwarning(
         "section="&section&", item="&item&", not able to read Z");
   }
   return b;
}

bool Record::get(const Word& section, const Word& item, R& value,
   Z verb )const
{
   Key k(section,item);
   R1 val(value);
   bool b=rep2.get(k,val);
   if (b) value=val;
   else if (verb>0){
      cpmwarning(
         "section="&section&", item="&item&", not able to read R");
   }
   return b;
}

bool Record::get(const Word& section, const Word& item,
   Word& value, Z verb)const
{
   Key k(section,item);
   Word val(value);
   bool b=rep3.get(k,val);
   if (b) value=val;
   else if (verb>0){
      cpmwarning(
         "section="&section&", item="&item&", not able to read Word");
   }
   return b;
}

bool Record::get(const Word& section, const Word& item,
   V<bool>& value, Z verb)const
{
   Key k(section,item);
   V<B> val;
   bool b=vec0.get(k,val);
   if (b) value=val(F<B,bool>(g_bool));
   else if (verb>0){
      cpmwarning(
```

```
         "section="&section&", item="&item&", not able to read Bs");
   }
   return b;
}

   V<bool> Record::getBs(Word const& section, Word const& item)const
   { return vec0[Key(section,item)](F<B,bool>(g_bool));}


   V<Z> Record::getZs(Word const& section, Word const& item)const
   { return vec1[Key(section,item)](F<Z1,Z>(g_Z));}


   V<R> Record::getRs(Word const& section, Word const& item)const
   { return vec2[Key(section,item)](F<R1,R>(g_R));}


bool Record::get(const Word& section, const Word& item,
   V<Z>& value, Z verb)const
{
   Key k(section,item);
   V<Z1> val;
   bool b=vec1.get(k,val);
   if (b) value=val(F<Z1,Z>(g_Z));
   else if (verb>0){
      cpmwarning(
         "section="&section&", item="&item&", not able to read Zs");
   }
   return b;
}

bool Record::get(const Word& section, const Word& item,
   V<R>& value, Z verb)const
{
   Key k(section,item);
   V<R1> val;
   bool b=vec2.get(k,val);
   if (b) value=val(F<R1,R>(g_R));
   else if (verb>0){
      cpmwarning(
         "section="&section&", item="&item&", not able to read Rs");
   }
   return b;
}

bool Record::get(const Word& section, const Word& item,
   V<Word>& value, Z verb)const
{
   Key k(section,item);
   V<Word> val;
```

```
   bool b=vec3.get(k,val);
   if (b) value=val;
   else if (verb>0){
      cpmwarning(
         "section="&section&", item="&item&", not able to read Ws");
   }
   return b;
}

bool Record::get(TextH& comment)const{
   comment=rep6;
   return true;
}

bool Record::get(const Word& section, TextH& comment)const
{
   Word sectionAct=section;
   sectionAct.skipSurSpc_();
      // see comment to
      // bool Record::set(const Word& section, const TextH& value)
   bool b=rep5.get(sectionAct,comment);
   // no warning for !b since comments need'nt to be present
   return b;
}

bool Record::get(const Word& section, const Word& item, TextH& com)const
{
   Key k(section,item);
   bool b=rep4.get(k,com);
   // no warning for !b since comments need'nt to be present
   return b;
}

// I/0 interface

bool Record::prnOn(ostream& str)const
{
   cpmwt("Record");
   cpmp(valid); // added 2006-02-05
   cpmp(rep0);
   cpmp(vec0);
   cpmp(rep1);
   cpmp(vec1);
   cpmp(rep2);
   cpmp(vec2);
   cpmp(rep3);
   cpmp(vec3);
   cpmp(rep4);
   cpmp(rep5);
   cpmp(rep6);
```

```
   return true;
}

bool Record::scanFrom(istream& str)
{
   cpms(valid); // added 2006-02-05
   cpms(rep0);
   cpms(vec0);
   cpms(rep1);
   cpms(vec1);
   cpms(rep2);
   cpms(vec2);
   cpms(rep3);
   cpms(vec3);
   cpms(rep4);
   cpms(rep5);
   cpms(rep6);
   return true;
}

namespace{ // local tools

   string skipWhitespace(const string& w)
   {
      Z i,n=(Z)w.size();
      if (n==0) return w;
      V<char> v(n);
      V<bool> val(n);
      for (i=1;i<=n;++i){
         char c=w[i-1];
         v[i]=c;
         val[i]=(isspace(c) ? false : true);
            // isspace(ws)==true for whitespace ws
      }
      V<char> prel=v.select(val);
      n=prel.dim();
      string res(n,'x');
      for (i=1;i<=n;++i) res[i-1]=prel[i];
      return res;
   }

   vector<string> parseLine(const string& str)
      // parsing a line into whitespace-separated words
   {
      vector<string> res;
      istringstream ist(str);
      string st;
      while (ist>>st){
         res.push_back(st);
      }
```

```
   return res;
}

vector<string> parseStream(istream& in)
   // parsing a text stream into non-blank, non-comment lines
   // The lines do not contain leading whitespace.
   // Since input from file can get seriously disturbed by violation
   // of syntax, this basic input function writes to cpmcerr what it
   // is dealing with
{
   Z mL=3;
   static Word loc("parseStream(istream&)");
   CPM_MA
   if (cpmverbose>mL){ // intentionally '>'
      cpmcerr<<"these are all data found in the istream"<<endl<<
      "sections are in alphabetic order,"<<endl<<
      "entries within sections in the order of reading"<<endl;
   }
   vector<string> res;
   while(in){
      string st;
      //std::getline(in,st);
      CpmRoot::readLine(st,in);
         // this eliminates carriage returns in addition to newlines
         // added 2004-09-15
      if (cpmverbose>mL) cpmcerr<<st<<endl;
      st=CpmRoot::skipLeadingWhitespace(st);
      Z ds=(Z)st.size();
      if (ds>0){
         char c1=st[0];
         if (c1!='*'&&c1!='/'&&c1!=';'){
               // lines starting (maybe after whitespace) with
               // '*' or '/' or ';' are ignored
            replaceFillCharacters_(st);
            res.push_back(st);
         }
      }
   }
   if (cpmverbose>mL)
      cpmcerr<<"end of data found in the istream"<<endl;
   CPM_MZ
   return res;
}

bool getText(vector<string> vs, Z& k, TextH& text)
{
   Z n=(Z)vs.size();
   if (k>=n){
      return false;
   }
```

```
   bool res=false;
   while(skipHead(vs[k])){
      text&=vs[k];
      k++;
      res=true;
      if (k>=n) break;
      res=true;
   }
   return res;
}

bool getSection(vector<string> vs, Z& k, Word& section)
{
   Z mL=4;
   Word loc("bool anon::getSection(...)");
   CPM_MA
   if (k>=(Z)vs.size()) return false;
   string line=vs[k];

   bool res=isSectionHeader(line,section);
   if (res){
      k++;
      if (cpmverbose>=mL) cpmcerr<<"section="<<section<<endl;
   }
   CPM_MZ
   return res;
}

bool getItem(vector<string> vs, Z& k, const Word& section,
   Word& type, Word& item, V<Word>& val)
{
   Z mL=3;
   Word loc("anon::getItem(...)");
   CPM_MA
   if (k>=(Z)vs.size()) return false;
   string line=vs[k];
   vector<string> wordsOfLine=parseLine(line);
   Z i,wols=(Z)wordsOfLine.size();
   Word sc;
   if (isSectionHeader(line,sc)){ // then we don't get a
      CPM_MZ
      return false;
   }
   k++;
   type=Word(wordsOfLine[0]);
   item=Word(wordsOfLine[1]);
   Z nList=wols-2; // nList>=0 since wols>=2
   bool insertDefault=false;
   if (nList==0){
      if (type=="F" || type=="include"){
```

```
            // for F and include it is natural to have wols==2
            // but no value has to be read
            CPM_MZ
            return false;
         }
         else{
            ostringstream ost;
            ost<<loc<<" value missing at section "<<section<<
               " type"<<type
            <<" item "<<item<<" default value chosen";
            cpmwarning(ost);
            insertDefault=true;
            nList++;
         }
      }
      val=V<Word>(nList); // now nList>=1
      if (insertDefault){
         if (type=="B" || type=="Bs") val[1]="false";
         else if (type=="Z" || type=="Zs" || type=="R" || type=="Rs")
            val[1]=0;
         else if (type=="W" || type=="Ws") val[1]="";
         else cpmerror(loc&": unvalid type");
         CPM_MZ
         return true;
      }
      for (i=1;i<=nList;++i) val[i]=Word(wordsOfLine[i+1]);
         // wordsOfLine[2], wordsOfLine[3], ....
         // transformed into a list of Words
      if (cpmverbose>=mL){
         cpmcerr<<"section="<<section<<", type="<<type<<", item="<<item;
         for (i=1;i<=nList;++i) cpmcerr<<", val["<<i<<"]="<<val[i];
         cpmcerr<<endl;
      }
      CPM_MZ
      return true;
   }

} // namespace


bool Record::writeFormatted(ostream& str, bool writeComments)const
{
   const Z mL=1;
   Word loc("Record::writeFormatted(ostream&,bool)");
   CPM_MA
   if (!str){
      cpmwarning(loc&": called for unvalid ostream");
      CPM_MZ
      return false;
   }
```

```
if (!isVal()){
   cpmwarning(loc&": denied to write unvalid Record");
   CPM_MZ
   return false;
}
S<Word> sections, krep0, kvec0, krep1, kvec1, krep2, kvec2,
   krep3, kvec3;
Z i,n;

n=rep0.dim();
for (i=1;i<=n;i++){
   Key k=rep0.x(i);
   sections.add(k.c1());
   krep0.add(k.c2());
}

n=vec0.dim();
for (i=1;i<=n;i++){
   Key k=vec0.x(i);
   sections.add(k.c1());
   kvec0.add(k.c2());
}

n=rep1.dim();
for (i=1;i<=n;i++){
   Key k=rep1.x(i);
   sections.add(k.c1());
   krep1.add(k.c2());
}

n=vec1.dim();
for (i=1;i<=n;i++){
   Key k=vec1.x(i);
   sections.add(k.c1());
   kvec1.add(k.c2());
}

n=rep2.dim();
for (i=1;i<=n;i++){
   Key k=rep2.x(i);
   sections.add(k.c1());
   krep2.add(k.c2());
}

n=vec2.dim();
for (i=1;i<=n;i++){
   Key k=vec2.x(i);
   sections.add(k.c1());
   kvec2.add(k.c2());
}
```

```
   n=rep3.dim();
   for (i=1;i<=n;i++){
      Key k=rep3.x(i);
      sections.add(k.c1());
      krep3.add(k.c2());
   }

   n=vec3.dim();
   for (i=1;i<=n;i++){
      Key k=vec3.x(i);
      sections.add(k.c1());
      kvec3.add(k.c2());
   }

   n=rep4.dim();
   for (i=1;i<=n;i++){
      Key k=rep4.x(i);
      sections.add(k.c1());
   }

   bool wc=writeComments;

   ostringstream res;
   rep6.prnOn(res);

   Z ns=sections.card();
   cpmmessage(loc&": "&cpm(ns)&" sessions found");
   Z nk,j,i_;
   Word secSep(
      "//--------------------------------\
----------------------------------------");

   for (i=1;i<=ns;++i){
      Word seci=sections[i];
      res<<endl<<seci.str();
      res<<endl<<secSep;
      TextH valsc;
      bool bsc=get(seci,valsc);
         // This is the place where rep5 is used
         // careful version of rep5[seci] that accepts
         // whitespace-cluttered versions of the key
      if (bsc&&wc) valsc.prnOn(res);
      nk=krep0.card();
      for (j=1;j<=nk;j++){
         Word itemj=krep0[j];
         Key k(seci,itemj);
         B value;
         bool b=rep0.get(k,value);
         if (b){
```

```
        res<<endl<<"B "<<itemj.str()<<"="<<toStr(value);
        TextH valc;
        bool bc=rep4.get(k,valc);
        if (bc&&wc) valc.prnOn(res);
    }
}
nk=kvec0.card();
for (j=1;j<=nk;j++){
   Word itemj=kvec0[j];
   Key k(seci,itemj);
   V<B> value;
   bool b=vec0.get(k,value);
   if (b){
      n=value.dim();
      res<<endl<<"Bs "<<itemj.str()<<"=";
      for (i_=1;i_<=n;++i_) res<<toStr(value[i_])<<" ";
      TextH valc;
      bool bc=rep4.get(k,valc);
      if (bc&&wc) valc.prnOn(res);
   }
}
nk=krep1.card();
for (j=1;j<=nk;j++){
   Word itemj=krep1[j];
   Key k(seci,itemj);
   Z1 value;
   bool b=rep1.get(k,value);
   if (b){
       res<<endl<<"Z "<<itemj.str()<<"="<<value.x1;
       TextH valc;
       bool bc=rep4.get(k,valc);
       if (bc&&wc) valc.prnOn(res);
   }
}
nk=kvec1.card();
for (j=1;j<=nk;j++){
   Word itemj=kvec1[j];
   Key k(seci,itemj);
   V<Z1> value;
   bool b=vec1.get(k,value);
   if (b){
      n=value.dim();
      res<<endl<<"Zs "<<itemj.str()<<"=";
      for (i_=1;i_<=n;++i_) res<<toStr(value[i_].x1)<<" ";
      TextH valc;
      bool bc=rep4.get(k,valc);
      if (bc&&wc) valc.prnOn(res);
   }
}
nk=krep2.card();
```

```
for (j=1;j<=nk;j++){
   Word itemj=krep2[j];
   Key k(seci,itemj);
   R1 value;
   bool b=rep2.get(k,value);
   if (b){
       res<<endl<<"R "<<itemj.str()<<"="<<value.x1;
       TextH valc;
       bool bc=rep4.get(k,valc);
       if (bc&&wc) valc.prnOn(res);
   }
}
nk=kvec2.card();
for (j=1;j<=nk;j++){
   Word itemj=kvec2[j];
   Key k(seci,itemj);
   V<R1> value;
   bool b=vec2.get(k,value);
   if (b){
      n=value.dim();
      res<<endl<<"Rs "<<itemj.str()<<"=";
      for (i_=1;i_<=n;++i_) res<<toStr(value[i_].x1)<<" ";
      TextH valc;
      bool bc=rep4.get(k,valc);
      if (bc&&wc) valc.prnOn(res);
   }
}
nk=krep3.card();
for (j=1;j<=nk;j++){
   Word itemj=krep3[j];
   Key k(seci,itemj);
   Word value;
   bool b=rep3.get(k,value);
   if (b){
       res<<endl<<"W "<<itemj.str()<<"="<<value.str();
       TextH valc;
       bool bc=rep4.get(k,valc);
       if (bc&&wc) valc.prnOn(res);
   }
}
nk=kvec3.card();
for (j=1;j<=nk;j++){
   Word itemj=kvec3[j];
   Key k(seci,itemj);
   V<Word> value;
   bool b=vec3.get(k,value);
   if (b){
      n=value.dim();
      res<<endl<<"Ws "<<itemj.str()<<"=";
      for (i_=1;i_<=n;++i_) res<<-value[i_]<<" ";
```

```
            TextH valc;
            bool bc=rep4.get(k,valc);
            if (bc&&wc) valc.prnOn(res);
         }
      }
      res<<endl;
   }
   str<<res.str();
   CPM_MZ
   return (str!=0);
}

bool Record::readFormatted(istream& str)
// When this function is called from Record::readFromFile(), the argument
// is a stringstream that passed a lot of processing already. So it is
// not necessarily of the same line order as the ini-file under
// consideration
// (reason is the new file include mechanism).
{
   const Z mL=2;
   Word loc("Record::readFormatted(istream&)");
   CPM_MA

   if (!str){
      cpmwarning("Record::readFormatted() called for unvalid stream");
      return false;
   }

   rep0=m0();
   vec0=v0();

   rep1=m1();
   vec1=v1();

   rep2=m2();
   vec2=v2();

   rep3=m3();
   vec3=v3();

   rep4=m4();
  // rep5=m5();
   rep6=m6();

   vector<string> pf=parseStream(str);
      // list of all non-void, non-comment lines, not starting with
      // white-space. All intermediate ; , = are eliminated
      // At the end str no longer valid, so return (str!=0) not good!
      // This is the function that writes to cpmcerr
   Z pfs=(Z)pf.size();
```

```
   if (pfs==0){
      cpmwarning(loc&": false returned due to pfs==0");
      CPM_MZ
      return false;
   }
   Z n,i,k=0;
// reading the explanation for the whole Record
   getText(pf,k,rep6);
   Word section;
   while (getSection(pf,k,section)){
      TextH sectionText;
      while(getText(pf,k,sectionText)){
         set(section,sectionText);
      }
      Word item,type;
      V<Word> val;
      while(getItem(pf,k,section,type,item,val)){
         Key key(section,item);
         if (type=="B"){
            B b(val[1].toBool());
            rep0.set(key,b);
         }
         else if (type=="Bs"){
            n=val.dim();
            V<B> vb(n);
            for (i=1;i<=n;++i) vb[i]=B(val[i].toBool());
            vec0.set(key,vb);
         }
         else if (type=="Z"){
            Z1 z1(val[1].toZ());
            rep1.set(key,z1);
         }
         else if (type=="Zs"){
            n=val.dim();
            V<Z1> vz1(n);
            for (i=1;i<=n;++i) vz1[i]=Z1(val[i].toZ());
            vec1.set(key,vz1);
         }
         else if (type=="R"){
            R1 r1(val[1].toR());
            rep2.set(key,r1);
         }
         else if (type=="Rs"){
            n=val.dim();
            V<R1> vr1(n);
            for (i=1;i<=n;++i) vr1[i]=R1(val[i].toR());
            vec2.set(key,vr1);
         }
         else if (type=="W"){
            rep3.set(key,val[1]);
```

```
         }
         else if (type=="Ws"){
            vec3.set(key,val);
         }
         else {
            cpmerror("Record::readFormatted: undefined data type "&type);
         }
         TextH itemText;
         while(getText(pf,k,itemText)){
            rep4.set(key,itemText);
         }
      }
   }
   CPM_MZ
   return true;
}

bool Record::readFromFile(const Word& fn, bool binary)
{
   Z mL=2;
   Word loc("Record::readFromFile(const Word& fn, bool) fn="&fn);
   CPM_MA
   SecCol indat(fn,binary);
   if (!indat.isVal()){
      cpmwarning(loc&": indat not valid");
      CPM_MZ
      return false;
   }
   Z n=indat.nFiles();
   string data=indat.str();
   istringstream ist(data);
   if (cpmverbose>mL) cpmdata<<"Data collected from "<<n<<
      " ini-files:"<<data<<endl;
   // note that the constructor of IniFileBasedApplication creates a nice
   // record of the read data on file cpmdata.txt, so that the present
   // output may be useful for critical diagnostics only
   bool res=readFormatted(ist);
   valid=B(res); // added 2006-02-07
   if (!valid) cpmwarning(loc&": return value is false");
   CPM_MZ
   return res;
}

namespace{
   bool selFunc(Key const& k, Word const& sec){ return k.c1()==sec;}
}

Record Record::operator()(Word const& section)const
{
   F<Key,bool> sel=F1<Key,Word,bool>(section)(selFunc);
```

```
   Record res;
   res.rep0=rep0.select(sel);
   res.vec0=vec0.select(sel);
   res.rep1=rep1.select(sel);
   res.vec1=vec1.select(sel);
   res.rep2=rep2.select(sel);
   res.vec2=vec2.select(sel);
   res.rep3=rep3.select(sel);
   res.vec3=vec3.select(sel);
   Word comment1(
      "This single-section record was made by extracting section");
   comment1&=section;
   Word comment2("from a record with the following overall comment:");
   TextH overall;
   get(overall);
   TextH comm(comment1);
   comm&=comment2;
   comm&=overall;
   res.set(comm);
   res.valid=B(true);
   return res;
}

Record Record::operator()(Word const& sec1, Word const& sec2)const
{
   F<Key,bool> sel=F1<Key,Word,bool>(sec1)(selFunc);
   Z i;
   Record res;
   m0 r0s=rep0.select(sel);
   for (i=1;i<=r0s.dim();++i){
      Word item=r0s.x(i).c2();
      B value=r0s.y(i);
      res.set(sec2,item,value);
   }

   v0 v0s=vec0.select(sel);
   for (i=1;i<=v0s.dim();++i){
      Word item=v0s.x(i).c2();
      V<B> value=v0s.y(i);
      res.set(sec2,item,value);
   }

   m1 r1s=rep1.select(sel);
   for (i=1;i<=r1s.dim();++i){
      Word item=r1s.x(i).c2();
      Z1 value=r1s.y(i);
      res.set(sec2,item,value);
   }

   v1 v1s=vec1.select(sel);
```

```
   for (i=1;i<=v1s.dim();++i){
      Word item=v1s.x(i).c2();
      V<Z1> value=v1s.y(i);
      res.set(sec2,item,value);
   }

   m2 r2s=rep2.select(sel);
   for (i=1;i<=r2s.dim();++i){
      Word item=r2s.x(i).c2();
      R1 value=r2s.y(i);
      res.set(sec2,item,value);
   }

   v2 v2s=vec2.select(sel);
   for (i=1;i<=v2s.dim();++i){
      Word item=v2s.x(i).c2();
      V<R1> value=v2s.y(i);
      res.set(sec2,item,value);
   }

   m3 r3s=rep3.select(sel);
   for (i=1;i<=r3s.dim();++i){
      Word item=r3s.x(i).c2();
      Word value=r3s.y(i);
      res.set(sec2,item,value);
   }

   v3 v3s=vec3.select(sel);
   for (i=1;i<=v3s.dim();++i){
      Word item=v3s.x(i).c2();
      V<Word> value=v3s.y(i);
      res.set(sec2,item,value);
   }

   Word comment1(
      "This single-section record was made by extracting section");
   comment1&=sec1;
   Word comment2("from a record with the following overall comment:");
   TextH overall;
   get(overall);
   TextH comm(comment1);
   comm&=comment2;
   comm&=overall;
   res.set(comm);
   res.valid=B(true);
   return res;
}

///////////////// class Recordable ////////////////////////////////////
```

```
bool Recordable::adjust()const
{
   if (valid_) return true;
   else{
      valid_=true;
      ostringstream ost;
      rc_.writeFormatted(ost,false);
      rep_=Word(ost.str());
      return vadjust();
   }
}

bool Recordable::prnOn(ostream& str)const
{
   cpmwt(-nameOf());
   cpmp(rc_);
   return true;
}

bool Recordable::scanFrom(istream& str)
{
   cpms(rc_);
   renew_();
   return adjust();
}

Z Recordable::com(Recordable const& r)const
{
   return rep().com(r.rep());
}

////////////////////// class Text ////////////////////////////////////
Text Text::getLine(istream& str)
{
   string line;
   CpmRoot::readLine(line ,str);
      // reads over comments an eliminates CRs (not newlines)
   F<string,Word> trans;
   vector<string> resPrel=parseLine(line);
   Z n=(Z)resPrel.size();
   V<Word> res(n);
   for (Z i=1;i<=n;++i) res[i]=Word(resPrel[i-1]);
   return Text(res);
}
```

155 cpmrecordhandler.h

```
//? cpmrecordhandler.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_RECORDHANDLER_H_
#define CPM_RECORDHANDLER_H_

/*
A tool for creating and initializing instances of class
   CpmRootX::Record from 'ini-files'.
   Suitable for data distribution in parallel computing based on MPI.
   See class RecordHandler.
*/

#include <cpmrecord.h>

namespace CpmRootX{

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::Rh;
   using CpmRoot::Word;
   using CpmRootX::Z1;
   using CpmRootX::R1;
   using CpmRootX::Record;

   V<Word> combine(V<Word> const& argsConsole, V<Word> const& argsFile);
      // combining commandline info and command line substitute info
      // into a single command line substitute, where console input
      // is given priority

   V<Word> extComLine(int argc, char *argv[]);
      //: extended command line

   class RecordHandler{ // tool for reading and writing a Record
   // RecordHandler is a tool reading a Record from file putting it
   // into memory and manipulating it there (by the write functions).
   // The read functions access the data in memory. In a SPMD
   // (single program, multiple data ) context, the functionality
   // is such that file opening and reading (by the constructor) is done
   // for rank==1 and the results are broadcasted to the memory of
   // all other ranks.
   // The assumed syntax of the file is described in class Record.

   void ini_(V<Word> const& files,
      Word const& path, Word const& callMessage);
   // construction tool
```

```
public:
   explicit RecordHandler(V<Word> const& files=V<Word>(0),
      Word const& path="", Word const& callMessage="")
   { ini_(files,path,callMessage);}
   // Constructor from a list of file names.
   // If rank==1, the files in the list will be opened in their order
   // until function readFromFile was able to read a valid
   // Record. If this is impossible for all files in the list,
   // the constructed instance res has the property
   // res.isVal()==false. This constructor distributes
   // the data found for rank==1 to all other processes such that
   // following read and write functions work in each process just as in
   // a single process program.
   // The last argument causes
   //       cpmmessage(callMessage);
   // near to the beginning of the defining function block. This helps
   // to stay informed if many ini-files are involved in a program.
   // The argument 'path' has a special meaning, if it equals "fromCon"
   // (meaning 'from configuration'): Then we use Message::getDirBase()
   // for path. This quantity can be set on the last line of the file
   // cpmconfig.txt and thus allows to switch between data sources without
   // recompilation.

   explicit RecordHandler(Word const& file, Word const& path="fromCon",
      Word const& callMessage="")
   { ini_(V<Word>("",file),path,callMessage);}
   // As previous function, but for convenience, a single
   // file name has not to be made a vector before it can be used as an
   // argument. Also for 'path' the default argument is given that
   // is probably most useful in most cases.

   explicit RecordHandler(Record const& rc):
   valid(true),rank(1),size(1),rec(rc){}
      // added 2005-04-05. Since much older code uses class RecordHandler
      // in situations in which Record would be sufficient, here is a
      // conversion function:

// writing the fundamental basic types at a position determined by
// section and key.  The return value false indicates failure.

   bool write(const Word& section, const Word& key,
      const Z& value);

   bool write(const Word& section, const Word& key,
      const R& value);

   bool write(const Word& section, const Word& key,
      const Word& value);

   bool write(const Word& section, const Word& key,
```

```
      const Z1& value);

   bool write(const Word& section, const Word& key,
      const R1& value);

   bool write(const Word& section, const Word& key,
      const bool& value);

   bool write(const Word& section, const Word& key,
      const B& value);

// vector types (only those with CpmRoot-typed components)

   bool write(const Word& section, const Word& key,
      const V<bool>& value);

   bool write(const Word& section, const Word& key,
      const V<Z>& value);

   bool write(const Word& section, const Word& key,
      const V<R>& value);

   bool write(const Word& section, const Word& key,
      const V<Word>& value);

   bool writeFormatted(ostream& , bool writeComments)const;
      // redable representation of content according to the same grammar
      // as reading from file in the constructor. But different
      // order (alphabetic) than in input

// reading the fundamental basic types from a position determined by
// section and key. A failure in doing so stops the
// program for pedantic==2 and diagnostic messages will be left on file
// cpmcerr.txt. If pedantic==1 a message will be left, but the
// program will not stop. For pedantic==0 there is even no message.
// It is only for return value==true that argument value may get changed.

   bool read(const Word& section, const Word& key,
      Z& value, Z pedantic=2)const;

   bool read(const Word& section, const Word& key,
      R& value, Z pedantic=2)const;

   bool read(const Word& section, const Word& key,
      Rh& value, Z pedantic=2)const;

   bool read(const Word& section, const Word& key,
      Z1& value, Z pedantic=2)const;

   bool read(const Word& section, const Word& key,
```

```
      R1& value, Z pedantic=2)const;

   bool read(const Word& section, const Word& key,
      Word& value, Z pedantic=2)const;

   bool read(const Word& section, const Word& key,
      bool& value, Z pedantic=2)const;

   bool read(const Word& section, const Word& key,
      B& value, Z pedantic=2)const;

// vector types (only those with CpmRoot-typed components)
   bool read(const Word& section, const Word& key,
      V<bool>& value, Z pedantic=2)const;

   bool read(const Word& section, const Word& key,
      V<Z>& value, Z pedantic=2)const;

   bool read(const Word& section, const Word& key,
      V<R>& value, Z pedantic=2)const;

   bool read(const Word& section, const Word& key,
      V<Word>& value, Z pedantic=2)const;

   Word getFileName()const{ return filename;}
      //: get file name

   bool prnOn(ostream& str)const{ return rec.prnOn(str);}

   Record getRecord()const{ return rec;}

   Record toRec()const{ return rec;}
      //: to record

   bool isVal()const{ return valid;}
      //: is valid

   V<Word> comLine()const;
      // read a command line substitute and return it.
      // If nothing was found, we return a V of length 0.

private:
   B valid;
   Z rank;
   Z size;
   Word filename;
   // Filename of the inifile
   CpmRootX::Record rec;
      // replacement for the functionality of Windows ini-files
};
```

```
} // namespace
#endif
```

156 cpmrecordhandler.cpp

```
//? cpmrecordhandler.cpp
//? Status of work 2008-10-25.
//?
#include <cpmrecordhandler.h>
#include <cpmsystem.h>
#include <cpmfile.h>

using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::Rh;
using CpmRoot::Word;
using CpmRootX::Z1;
using CpmRootX::R1;
using CpmRootX::B;
using CpmRootX::RecordHandler;
using CpmRootX::Record;
using CpmRootX::sup;
using CpmSystem::IFileStream;
using CpmSystem::glue;
using CpmSystem::Message;
using CpmArrays::V;
using namespace std;

V<Word> CpmRootX::combine(V<Word> const& argsC, V<Word> const& argsF)
{
   if (!argsC.valInd(1)) return V<Word>();
   if (!argsF.valInd(1)) return argsC;
   V<Word> argsIni=argsF.prepend(argsC[1]);
   Z nMax=sup<Z>(argsIni.dim(),argsC.dim());
   V<Word> res(nMax);
   for (Z i=res.b();i<=res.e();++i){
      res[i]= argsC.valInd(i) ? argsC[i] : argsIni[i];
   }
   return res;
}

V<Word> CpmRootX::extComLine(int argc, char *argv[])
{
   Z mL=1;
   Word loc("CpmRootX::extComLine(int,*char[])");
   CPM_MA
   V<Word> args=CpmArrays::comLine(argc, argv);
   cpmassert(args.valInd(1),loc);
   Word prog=args[1].baseName(); // important, since args_[1] carries the
      // full path
   bool extended=true;
      // prog may read vqm2MS.exe, or vqm2_ms.exe, or vqm2MinGW.exe
```

```
      // for program vqm2. So one should also remove the suffixes which
      // the user may have added to the executables name in order to
      // allow comparison of various compilers
   Word iniFile=prog.remExt(extended)&".ini";
 //  V<Word> inputFrom=serList(iniFile);
 //  RecordHandler rch(inputFrom,"",loc);
   RecordHandler rch(iniFile);
   if (!rch.isVal()){
      cpmwarning(loc&
         ": RecordHandler not valid, commandline not extended");
      CPM_MZ
      return args;
      // Thus no action if file fn does not exist
   }
// Now we have a valid RecordHandler to get data from
   V<Word> argsIni=rch.comLine();
   V<Word> res=combine(args,argsIni);
   CPM_MZ
   return res;
}

void RecordHandler::ini_(V<Word> const& files, Word const& path,
   Word const& callMessage)
{
   Z mL=1;
   Word pathAct = path=="fromCon" ? Message::getInpDir() : path;
   ostringstream ost;
   ost<<"files="<<files<<" path="<<pathAct<<" callMessage="<<callMessage;
   Word mes(ost);
   Word loc="RecordHandler::ini_("&mes&")";
      // probaby long, so not use too often
   CPM_MA
      // it is important to see the arguments
   rank=CpmMPI::Cpm_com.getRank();
   size=CpmMPI::Cpm_com.getSize();
   if (rank==1){
      Z i,fd=files.dim();
      V<Word> ffs(fd);
      for (i=1;i<=fd;++i) ffs[i]=glue(pathAct,files[i]);
      bool binary=true;
      i=1;
      while(!valid && i<=fd){ // B valid is data member,
            // thus initialized as false
         valid=rec.readFromFile(ffs[i],binary);
         filename=ffs[i];
         cpmdebug(filename);
         i++;
      }
      if (!valid){
         cpmwarning(" could not read any of the "&cpm(fd)&" files");
```

```
         if (size>1) cpmwarning("nothing sent to other processes");
      }
      else{
         if (size>1 || mL<=1){
            if (size>1) cpmmessage(mL,"Rank 1 to all other processes:");
            else cpmmessage(mL,"this is what we got:");
            ostringstream ost;
            rec.writeFormatted(ost,false);
            cpmmessage(mL,Word(ost));
         }
      }
   } // rank==1
   else if (size>1){
      cpmmessage("I am rank "&cpm(rank)&", my role is receiving");
      CpmMPI::Cpm_com.bcast(rec);
      if (rank!=1){
         cpmmessage(mL,"Rank "&cpm(rank)&", got this from rank 1:");
         ostringstream ost;
         rec.writeFormatted(ost,false);
         cpmmessage(mL,Word(ost));
         valid=rec.isVal();
      }
   }
   else ;
   cpmmessage(": valid="&cpm(valid));
   CPM_MZ
}

//////////////////////// read ////////////////////////////////////////////

bool RecordHandler::read(const Word& section, const Word& key,
   Z& value, Z pedantic)const
{
   bool b=rec.get(section,key,value,pedantic);
      // creates a warning if does not find
      // Z changed only if get successful
   if (b==false && pedantic>0){
      ostringstream ost;
      ost<<string("RecordHandler::read(): not able to read a Z")<<
      string(" in section=")<<section.toStr()<<string(" at key=")<<
      key.toStr();
      // very similar to the warning from get. But good to have
      // the text here too since it comes to the status line
      if (pedantic==2) cpmerror(Word(ost.str()));
   }
   return b;
}

bool RecordHandler::read(const Word& section, const Word& key,
   Z1& value, Z pedantic)const
```

```
{
   Z val; // using val fot type conversion
   // protection of value against changes in unsuccssful
   // reading is given anyway
   bool b=rec.get(section,key,val,pedantic);
   if (b==false && pedantic>0){
      ostringstream ost;
      ost<<string("RecordHandler::read(): not able to read a Z1")<<
      string(" in section=")<<section.toStr()<<string(" at key=")<<
      key.toStr();
      if (pedantic==2) cpmerror(Word(ost.str()));
   }
   else value=Z1(val);
   return b;
}

bool RecordHandler::read(const Word& section, const Word& key,
   R1& value, Z pedantic)const
{
   R val;
   bool b=rec.get(section,key,val,pedantic);
   if (b==false && pedantic>0){
      ostringstream ost;
      ost<<string("RecordHandler::read(): not able to read a R1")<<
      string(" in section=")<<section.toStr()<<string(" at key=")<<
      key.toStr();
      if (pedantic==2) cpmerror(Word(ost.str()));
   }
   else value=R1(val);
   return b;
}

bool RecordHandler::read(const Word& section, const Word& key,
   Rh& value, Z pedantic)const
{
   R val;
   bool b=rec.get(section,key,val,pedantic);
   if (b==false && pedantic>0){
      ostringstream ost;
      ost<<string("RecordHandler::read(): not able to read a Rh")<<
      string(" in section=")<<section.toStr()<<string(" at key=")<<
      key.toStr();
      if (pedantic==2) cpmerror(Word(ost.str()));
   }
   else value=Rh(val);
   return b;
}

bool RecordHandler::read(const Word& section, const Word& key,
   R& value, Z pedantic)const
```

```
{
   bool b=rec.get(section,key,value,pedantic);
   if (b==false && pedantic>0){
      ostringstream ost;
      ost<<string("RecordHandler::read(): not able to read a R")<<
      string(" in section=")<<section.toStr()<<string(" at key=")<<
      key.toStr();
      if (pedantic==2) cpmerror(Word(ost.str()));
   }
   return b;
}

bool RecordHandler::read(const Word& section, const Word& key,
   Word& value, Z pedantic)const
{
   bool b=rec.get(section,key,value,pedantic);
   if (b==false && pedantic>0){
      ostringstream ost;
      ost<<string("RecordHandler::read(): not able to read a Word")<<
      string(" in section=")<<section.toStr()<<string(" at key=")<<
      key.toStr();
      if (pedantic==2) cpmerror(Word(ost.str()));
   }
   return b;
}

bool RecordHandler::read(const Word& section, const Word& key,
   bool& value, Z pedantic)const
{
   bool b=rec.get(section,key,value,pedantic);
   if (b==false && pedantic>0){
      ostringstream ost;
      ost<<string("RecordHandler::read(): not able to read a bool")<<
      string(" in section=")<<section.toStr()<<string(" at key=")<<
      key.toStr();
      if (pedantic==2) cpmerror(Word(ost.str()));
   }
   return b;
}

bool RecordHandler::read(const Word& section, const Word& key,
   B& value, Z pedantic)const
{
   bool val;
   bool b=rec.get(section,key,val,pedantic);
   if (b==false && pedantic>0){
      ostringstream ost;
      ost<<string("RecordHandler::read(): not able to read a B")<<
      string(" in section=")<<section.toStr()<<string(" at key=")<<
      key.toStr();
```

```
      if (pedantic==2) cpmerror(Word(ost.str()));
   }
   value=B(val);
   return b;
}

bool RecordHandler::read(const Word& section, const Word& key,
   V<bool>& value, Z pedantic)const
{
   bool b=rec.get(section,key,value,pedantic);
   if (b==false && pedantic>0){
      ostringstream ost;
      ost<<string("RecordHandler::read(): not able to read Bs")<<
      string(" in section=")<<section.toStr()<<string(" at key=")<<
      key.toStr();
      if (pedantic==2) cpmerror(Word(ost.str()));
   }
   return b;
}

bool RecordHandler::read(const Word& section, const Word& key,
   V<Z>& value, Z pedantic)const
{
   bool b=rec.get(section,key,value,pedantic);
   if (b==false && pedantic>0){
      ostringstream ost;
      ost<<string("RecordHandler::read(): not able to read Zs")<<
      string(" in section=")<<section.toStr()<<string(" at key=")<<
      key.toStr();
      if (pedantic==2) cpmerror(Word(ost.str()));
   }
   return b;
}

bool RecordHandler::read(const Word& section, const Word& key,
   V<R>& value, Z pedantic)const
{
   bool b=rec.get(section,key,value,pedantic);
   if (b==false && pedantic>0){
      ostringstream ost;
      ost<<string("RecordHandler::read(): not able to read Rs")<<
      string(" in section=")<<section.toStr()<<string(" at key=")<<
      key.toStr();
      if (pedantic==2) cpmerror(Word(ost.str()));
   }
   return b;
}

bool RecordHandler::read(const Word& section, const Word& key,
   V<Word>& value, Z pedantic)const
```

```
{
   bool b=rec.get(section,key,value,pedantic);
   if (b==false && pedantic>0){
      ostringstream ost;
      ost<<string("RecordHandler::read(): not able to read Ws")<<
      string(" in section=")<<section.toStr()<<string(" at key=")<<
      key.toStr();
      if (pedantic==2) cpmerror(Word(ost.str()));
   }
   return b;
}

//////////////////////// write ////////////////////////////////////////

bool RecordHandler::write(const Word& section, const Word& key,
   const Z& value)
{
   bool b=rec.set(section,key,value);
   if (b==false)
      cpmerror("RecordHandler::write(Z): failed at "&section&" "&key);
   return b;
}

bool RecordHandler::write(const Word& section, const Word& key,
   const Z1& value)
{
   bool b=rec.set(section,key,value);
   if (b==false)
      cpmerror("RecordHandler::write(Z1): failed at "&section&" "&key);
   return b;
}

bool RecordHandler::write(const Word& section, const Word& key,
   const R& value)
{
   bool b=rec.set(section,key,value);
   if (b==false)
      cpmerror("RecordHandler::write(R): failed at "&section&" "&key);
   return b;
}

bool RecordHandler::write(const Word& section, const Word& key,
   const R1& value)
{
   bool b=rec.set(section,key,value);
   if (b==false)
      cpmerror("RecordHandler::write(R1): failed at "&section&" "&key);
   return b;
}
```

```
bool RecordHandler::write(const Word& section, const Word& key,
   const Word& value)
{
   bool b=rec.set(section,key,value);
   if (b==false)
      cpmerror("RecordHandler::write(Word): failed at "&section&" "&key);
   return b;
}

bool RecordHandler::write(const Word& section, const Word& key,
   const bool& value)
{
   bool b=rec.set(section,key,value);
   if (b==false)
      cpmerror("RecordHandler::write(bool): failed at "&section&" "&key);
   return b;
}

bool RecordHandler::write(const Word& section, const Word& key,
   const B& value)
{
   bool bVal=value;
   bool b=rec.set(section,key,bVal);
   if (b==false)
      cpmerror("RecordHandler::write(B): failed at "&section&" "&key);
   return b;
}

bool RecordHandler::write(const Word& section, const Word& key,
   const V<bool>& value)
{
   bool b=rec.set(section,key,value);
   if (b==false)
      cpmerror("RecordHandler::write(V<bool>): failed at "&section&
         " "&key);
   return b;
}

bool RecordHandler::write(const Word& section, const Word& key,
   const V<Z>& value)
{
   bool b=rec.set(section,key,value);
   if (b==false)
      cpmerror("RecordHandler::write(V<Z>): failed at "&section&" "&key);
   return b;
}

bool RecordHandler::write(const Word& section, const Word& key,
   const V<R>& value)
{
```

```
   bool b=rec.set(section,key,value);
   if (b==false)
      cpmerror("RecordHandler::write(V<R>): failed at "&section&" "&key);
   return b;
}

bool RecordHandler::write(const Word& section, const Word& key,
   const V<Word>& value)
{
   bool b=rec.set(section,key,value);
   if (b==false)
      cpmerror("RecordHandler::write(V<Word>): failed at "
         &section&" "&key);
   return b;
}

bool RecordHandler::writeFormatted(ostream& str,bool writeComments)const
{
   return rec.writeFormatted(str,writeComments);
}

V<Word> RecordHandler::comLine()const
{
   Word sec1("command line"); // preferred style
   Word sec2("command_line"); // for compatibility
   Word key("args");
   V<Word> res;
   bool valid=false;
   Z ped=1; // only warn
   valid=read(sec1,key,res,ped);
   if (valid) return res;
   valid=read(sec2,key,res,ped);
   if (valid) return res;
   return V<Word>();
}
```

157 cpmrfunc.h

```
//? cpmrfunc.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_R_FUNC_H_
#define CPM_R_FUNC_H_
/*

Description:

   Specializing the 'function template' Fa<X,Y> to X=R and Y=R
   'real functions'. Beyond mere specialization, interpolation is
   treated: There are constructors for class R_Func which define a
   function from discrete lists. In addition, differentiation,
   integration, and some other topics are implemented. Graphical
   representation is implemented by a suitable  member function of class
   Graph.

   Notice that the rather complex template Fa<X,Y> is hidden to the
   user of the present class. He has only to include cpmrfunc.h.
   This, however, includes cpmfa.h, so the program size will be
   influenced by the complexities of this construction.

   Good example for usage of complex templates as a 'production machine'
   for small simple non-template classes.

*/

#include <cpmfa.h>
#include <cpmrmatrix.h>
#include <cpmsmoothlocalinterpol.h>

namespace CpmFunctions{

   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmRoot::Word;
   using CpmArrays::V;
   using CpmArrays::R_Vector;
   using CpmArrays::R_Matrix;
   using CpmArrays::IpolControl;
   using CpmGeo::Iv;
   using CpmLinAlg::R2;
   using CpmLinAlg::R4;

//////////////////////////////////// class RomCon ////////////////////

class RomCon{ // control of Romberg integration
```

```
   //: Romberg control
   // Method parameters for Romberg integration
   R epsRel_;
      //: epsilon relative
      // bound for relative error
    R epsAbs_;
      //: epsilon absolute
      // bound for absolute error
      // If the extimated error remains below these bounds,
      // the computation is finished
   Z met_;
      //: method
      // 0 for no extrapolation, only sucessive refinement
      // 1 for polynomial extrapolation
      // 2 for rational extrapolation
   Z len_;
      //: length
      // number of values on which extrapolation is based
   Z cntMax_;
      //: counter maximum
      // bound on the number of interval trisections
   bool robErrEst_;
      //: robust error estimate
      // if this is true, the error estimate is based
      // on actual value change seen in the last
      // interval trisection
   typedef RomCon Type;
public:
   Word nameOf()const{ return Word("RomCon");}
   CPM_IO
   RomCon(R epsRel=1e-6, R epsAbs=1e-9, Z met=1, Z len=5, Z cntMax=9,
      bool robErrEst=false): epsRel_(epsRel),epsAbs_(epsAbs),met_(met),
      len_(len),cntMax_(cntMax), robErrEst_(false){}

    R epsRel()const{ return epsRel_;}
    void setEpsRel_(R eps){ epsRel_=cpmabs(eps);}

    R epsAbs()const{ return epsAbs_;}
    void setEpsAbs_(R eps){ epsAbs_=cpmabs(eps);}

    Z met()const{ return met_;}
    void setMet_(Z m){ met_=cpmpos(m);}

    Z len()const{ return len_;}
    void setLen_(Z n){ len_=cpmpos(n);}

    Z cntMax()const{ return cntMax_;}
    void setCntMax_(Z n){ cntMax_=cpmpos(n);}

    bool robErrEst()const{ return robErrEst_;}
```

```
    void robErrEst(bool b){ robErrEst_=b;}
    V<Word> show()const;
      // returns an array of text lines which can be used for
      // screen display using the Frame::setText(V<Word> const&)
      // method
};

//////////////////////////// class R_Func ////////////////////////////

class R_Func: public Fa<R,R>{ // functions R to R
   // R-valued functions on R with arithmetics and trivially defined
   // order properties, and construction from discrete data via
   // interpolation.
   // Many functions were added to serve the needs of an industrial
   // project dealing with bending wires by computer controlled
   // motion of bending tools. Here the dependence of tool coordidates
   // on time was represented by instances of R_Func.

public:

// carrying over the constructors from Fa<R,R>

   R_Func(const F<R,R>& g):Fa<R,R>(g){}
      // upcast constructor

   R_Func(const Fa<R,R>& g):Fa<R,R>(g){}
      // upcast constructor

   R_Func(const R_Func& g):Fa<R,R>(g){}
      // copy constructor

   R_Func(void):Fa<R,R>(){}
      // default constructor

// Construction from function pointers

explicit   R_Func( R (*f)(const R& )):Fa<R,R>(f){}
explicit   R_Func( R (*f)(R)):Fa<R,R>(f){}

// Construtor for constant function

explicit R_Func(const R& y1):Fa<R,R>(y1){};

// new constructor from lists of values, which define a real function
// via interpolation

explicit R_Func(const R_Vector& data,
   const IpolControl& c=IpolControl());
// data is a list of y-values
// the corresponding x values are 1,2,...dim(data)=:n
```

```
// This applies to the next three functions too.

   R_Func(const V<R>& x, const V<R>& y,
      const IpolControl& c=IpolControl());
      // x is a list of x-values
      // y is a list of y-values

explicit R_Func(const R_Matrix& data,
      const IpolControl& c=IpolControl());
      // data.dim1() has to be 2
      // data[1] is a list of x-values
      // data[2] is a list of y-values

explicit R_Func(const Word& filename,
   const IpolControl& c=IpolControl());
      // filename has to be a name of a file which contains
      // x[1] y[1] x[2] y[2] ... separated by blancs, tabs, or CRs
      // and which may contain comment lines started by *, /, or ;

   R_Func(R a0, R a1, R a2, R a3, R a4=0);
   // generates the 4th order polynomial y=a0+a1*x+a2*x^2+a3*x^3+a4*x^4
   // also a polynomial of third order can be created this way

   R_Func(Iv iv, R mean, R sigma);
   // generates a Gaussian bell curve where mean and sigma are relative
   // to the interval iv

   R_Func(R4 a);
   // generates a smooth wall function function which  transits
   // from 0 to 1 in the interval (a[1],a[2]) and falls back to 0 in the
   // interval (a[3],a[4]). A function call stops with error if the
   // following is not satisfied:
   // a[1]<a[2]<=a[3]<a[4]. Implementation is based on R_Func::hill()

   R_Func(R2 x, R2 y);
   // linear transformation such that x[1] |--> y[1], x[2] |--> y[2]

   R_Func(Iv x, Iv y);
   // linear transformation such that x[1] |--> y[1], x[2] |--> y[2]

   Word nameOf()const{ return Word("R_Func");}

// modifications
   R_Func stretch(R2 const& ri, R2 const& rf)const
   { return circ(R_Func(rf,ri));}
      // returns a function which on the range rf behaves just as
      // the original *this did on the range ri.
      // Thus for R_Func g=f.stretch(ri,rf) we have
      // g(x)=f(tr(x)) where tr: rf-->ri (so, when x varies
```

```
   // over rf, the original function gets evaluated over ri)

R_Func stretch(Iv const& ivi, Iv const& ivf)const
{ return circ(R_Func(ivf.toR2(),ivi.toR2()));}
   // returns a function which on the interrval ivf behaves just as
   // the original *this did on the interval ivi.
   // Thus for R_Func g=f.stretch(ivi,ivf) we have
   // g(x)=f(tr(x)) where tr: ivf-->ivi (so, when x varies
   // over ivf, the original function gets evaluated over ivi)

R_Vector eval(const R_Vector& rx)const;
   // natural action on R_Vectors. Notice that f(rx) is already
   // defined via the base class. But the result is a V<R> and
   // not directly a R_Vector. Thus f.eval(rx)=R_Vector(f(rx));
   // so the present function isn't needed urgently. Notice also that
   // it is creating desaster to define this operator by overloading
   // operator(), since this would hide the old operator ().
   // I consider this one of the most un-natural aspects of C++.

void stretch_(R k, R xFix=0);
   // changes the function f into x|-->f(x0+(x-x0)/k). Thus for
   // k>1 features of the function (e.g. with of a Gauss curve)
   // get enlarged. If x0 is not in the area of the feature, the
   // feature appears more to get shifted than extended.
   // Anyway, x0 is a fixed point of the operation.

R_Func morph(Iv iv, const R_Func& g)const;
   // Let g be monotone increasing on [0,1] such that g(0)=0
   // and g(1)=1. For x \in iv we have
   // (f.morph(iv,p))(x)=f(x'), x':=x0+g(p)(x1-x0)
   // p=iv.ramp(x), iv=Iv(x0,x1)

void shift_(R s);
   // changes *this i a way that the graph gets shifted by s
   // (i.e. to the right on a screen graphics for s>0)

R_Func mirror(R x0, R s=1.)const;
   // Defines a function which is mirror symmetric with respect
   // to x0 and coincides with *this for x<x0. This is a natural
   // requirement for functions that represent 1-dimensional
   // position versus time. Then the "mirrored" function describes
   // a situation where for t=t0 a motion reversal takes  place.
   // First need occured in a stepper motor control problem.
   // (2002-1-21)
   // Let g:=f.mirror(x0).
   // Then g(x)=( x>x0 ?  f(x0-s*(x-x0)) : f(x) )

R_Func append(const R_Func& f, R x0)const;
   // Let h:=f.append(g,x0).
   // Then h(x)=( x>x0 ?  g(x) : f(x) )
```

```
   R_Func constRightOf(R x0)const{ return append(R_Func(evl(x0)),x0);}
      // Let h:=f.constRightOf(x0).
      // Then h(x)=( x>x0 ?  g(x) : g(x0) )

   R_Func squeezeInConstant(Iv const& ivHost, Iv const& ivSqueezeIn)const;
      // Let h:=f.squeezeInConstant(ivHost,ivSqueezeIn)
      // Then f(x)=h(x) for x in the complement of ivHost
      // (thus it is only on ivHost that something happens to f)
      // h(x)=f(x0) for x in ivSqueezeIn, where x0 is the center of
      // ivSqueezeIn. On ivHost\ivSqueeceIn h is defined from f
      // by piecewise linear re-transformation of the argument
      // such that 'space is created for inserting the constant function
      // piece on ivSqueezeIn. Prototype for the most efficient
      // implementation style for R_Func's that evaluate in a way that
      // needs precomputed quantities for efficiency.

   R_Func squeezeInConstantLeft(const Iv& ivHost, R d)const;
      // Let h:=f.squeezeInConstantLeft(ivHost,d)
      // Then f(x)=h(x) for x in the complement of ivHost
      // (thus it is only on ivHost that something happens to f)
      // h(x)=f(x0) for x in [x0, x0+d] where x0=ivHost[1]
      // On ivHost\[x0,x0+d] h is defined from f
      // by linear re-transformation of the argument
      // such that 'space is created for inserting the constant function
      // piece on [x0,x0+d].

   R_Func squeezeInConstantRight(const Iv& ivHost, R d)const;
      // Let h:=f.squeezeInConstantRight(ivHost,d)
      // Then f(x)=h(x) for x in the complement of ivHost
      // (thus it is only on ivHost that something happens to f)
      // h(x)=f(x0) for x in [x0-d, x0] where x0=ivHost[2]
      // On ivHost\[x0-d,x0] h is defined from f
      // by linear re-transformation of the argument
      // such that 'space is created for inserting the constant function
      // piece on [x0-d,x0].

   R_Func shrink(const Iv& ivHost, const Iv& ivSub)const;
      // Consider
      // g=f.shrink(ivHost,ivSub);
      // f=f if ivSub fails to be contained in ivHost.
      // If ivSub is a subinterval of ivHost,
      // g looks on ivSub just as f looked on ivHost.
      // On ivHost\ivSub the g is obtained as constant continuation
      // of g|ivSub.
      // Further: g==f on R\ivHost.

// derivation
   R_Func der(Z n=1, R h=1e-3)const;
      // returns the n-fold derivation of *this taking h as the
```

```
      // finite difference. Since higher order formulas are used for
      // the derivatives, h has not to be very small in order to give
      // accurate results. If n is large, h has to be rather large to
      // give anything meaningful.

    R_Func derR(Z n, R h=1e-3)const;
      // iterative definition of by multiple application of der(1)
      // much slower than previous function.

// integration
   R_Func antiDer(R x0, R h=1e-3)const;
      // anti derivative
      // (this->integral(x0))(x) = integral from x0 to x of function
      // *this. Here h is a proposal for the space between the various
      // x-values used in Simpson's formula, actually an adjusted value
      // will be used there.

   R romInt(R x1, R x2, RomCon const& rc = RomCon() )const;
      // Romberg integration
      // More sophisticated method for integrating from x1 to x2.
      // Function evaluation at x1 and x2 is avoided, so integrable
      // singularities are allowed to be there.
      // Convergence then will not as fast as for non-singular functions.
      // Modified from functions midpnt, and qromo of Press et. al.

 // minimization
   R serMin(R xL, R& xMini, R xU, R tol=1e-6);
      // search minimum
      // after call xMini will be a value where the function takes a
      // minimum value in the interval xL <= xMini <= xU
      // The return value is the value which the function gives for
      // xMini. Method is iterative inverse parabolic interpolation so it
      // is fast and simple, but not robust in dealing with
      // non-cooperative functions

// static members which are useful

   static R_Func x0;
      // function x |--> 1
   static R_Func x1;
      // function x |--> x
   static R_Func x2;
      // function x |--> x*x
   static R_Func x3;
      // function x |--> x*x*x
   static R_Func x4;
      // function x |--> x*x*x*x

   static R_Func log10;
      // function x |--> log10(|x|)
```

```
   static R_Func abs;
      // function x |--> |x|

// Continuous functions which are constant outside the interval (0,1):

   static R_Func hill(R p=0.810747);
   // smooth down-hill connection from (0,1) to (1,0).
   // The function x |--> hill(x) is a smooth (i.e. derivatives of any
   // order exist) non-increasing function.
   // It has the constant value 1 for x<=0 and the constant value
   // 0 for x>=1.  So, it is a smooth down-hill shape, connecting level
   // 1 with level 0.
   // The whole curve is radially symmetric to the point (x,y)=(0.5,0.5).
   // The idea behind the construction is the following: The negative
   // hyperbolic tangent function is already similar to such a hill.
   // A transformation of the argument brings the x-values 0 and 1 to
   // infinity before they are inserted into the hyperbolic tangent.
   // The normal expression for the latter is transformed in order to
   // call exp only once. Futher, to avoid overflow riscs in exp, this
   // function is called only for negative arguments.
   // Actually there is a whole family of such depending on a parameter p
   // (see Kodak TR 278844A 09/24/92), here the value 0.810747 is
   // taken, which minimizes the L2-norm of the second derivative.

   static R_Func hill3;
   // similar to hill, but a polynomial of third order and function has
   // one continuous derivative only instead of all derivatives.

   static R_Func ramp(R p=0.810747);
      // R_Func(R_Func::x0 - R_Func::hill(p))
      // thus smooth up-hill connection from (0,0) to (1,1).

   static R_Func ramp3;
      // R_Func(R_Func::x0 - R_Func::hill3)
      // thus third order polynomial up-hill connection from
      // (0,0) to (1,1).
      // see my Mathematica file trivialramp.nb for interesting
      // properties of ramps under translation

   static R_Func transit(R2 const& x, R2 const& y, bool linear=false);
      // creates a third-order polynomial f such that
      // f(x[1])=y[1],f(x[2])=y[2], f'(x[1])=0, f'(x[2])=0
      // Outside the x-Range given by argument x, the returned
      // function is defined by constant continuation of f
      // Due to the vanishing derivations of f at x[1] and x[2]
      // the resulting total function is differentiable on R.
      // This description referes to the case that the boolean argument
      // has its default value 'false'. If it has the value 'true'
      // the third-order polynomial is replaced by a first-order
```

```
      // (linear) one so that the transition function is still
      // continuous but not differentiable. This turned out to be
      // useful in graphical tone scale transformations.

   static R_Func dTransit(const R2& x, const R2& y);
      // Derivation of transit, evaluates faster than
      // R_Func::transit(x,y).der()
      // Here function transit is always taken with the boolean
      // argument having its default value false.

   static R_Func terrace(R k=0.25);
      // up-hill connection from (0,0) to (1,1). Like ramp,
      // radially symmetric with respect to (0.5,0.5)
      // 0<=k<=1. The derivative at point (0.5,0.5) is
      // by a factor k smaller than in (0,0) and (1,1).
      // Invented by Thomas Dera in the context of stepper
      // motor control (2001-1-18)

   static R_Func terrace2(R k=0.25);
      // similar to previous function; essentially tangens replaced by
      // arcsin (Thomas Dera Mathematica file ramp.nb (2002-01-31)
      // also 0<=k<=1 and the same relation to the derivatives.

   static R_Func terraceBezierDera(R k, R m, bool inv=false, Z z=0,
      Z n=256, const IpolControl& c=IpolControl("polygon"));
      // terrace function as Bezier curve, designed by
      // Thomas Dera 2002-02-06
      // See ramps(TD).pdf
      // If inv==true, the inverse function is returned
      // For z==0 the values k and m are active to shape the curve
      // in a symmetrical way at both ends.
      // For z=-1 the left end is 'free' (there are only 3 Bezier points,
      // the neighbor of (0,0) is missing)
      // For z=1 the right end is 'free' (there are only 3 Bezier points,
      // the neighbor of (1,1) is missing)
      // Note of 2003-07-16: Since the function is defined by
      // interpolation,
      // a small value of n may cause unexpected behavior in detail.
      // This was the case in the CO-former application where a
      // mysterious
      // slope change of a time-position diagram could be removed by
      // changing n from 32 to 64. Now 100 for security
      // Addition 2003-07-17: For applications it is essential that the
      // function and its derivative are monotonic, this can best be
      // guarantied by "polygon"-interpolation and a large value of n.
      // (since polygon interpolation is simple, a large number of n
      // is OK)

static R_Func polyFunc(const R_Vector& c);
      // returns a suitably coded polynomial function
```

```
   // x0:=c[1]
   // 1/r:=c[2]
   // z:=(x-x0)/r
   // f(x)=c[3]+c[4]*z+c[5]*z*z+...+c[n]*z^(n-3).
   // Data vectors c characterizing polynomial functions obtained
   // fit to discrete value lists can be obtained by function
   // CpmArrays::polyFit(...), see cpmvectormore.h
   // Notice that polynomials in x (i.e. x0=0, r=1) that
   // as functions show natural behavior in a
   // short interval of large x (e.g. 100<x<101) have coefficients
   // spanning many orders of magnitude. Therefore allowing for
   // x0 and r helps to get numerically more convenient
   // parametrization of polynomial functions.

static R_Func Bernstein(const R_Vector& y, R2 a=R2(0,1));
   // Let n:=y.dim(). Then Bernstein(y,a) is the Bernstein
   // approximation for a continuous function f taking
   // values y[i]=f(x[i]), where x[i]=a[1]+((i-1)/(n-1))*(a[2]-a[1])
   // This always gives a smooth naturally looking curve which is
   // a special case of a Bezier spline. It is not exactly
   // interpolating, but the list y is a efficient design tool for
   // 'natural curves' showing a certain desired behavior.
   // See G. Aumann, K. Spitzmueller: Computerorientierte Geometrie,
   // BI
   // 1993, pp. 348-350.
   //
   // The implementation may serve as a pattern for other functions
   // that need extensive pre-calculations for fast evaluation:
   // R_Vector y=....;
   // R2 a(-1,1);
   // R_Func f=R_Func::Bernstein(y,a); // does the pre-calculations
   // R xm=0.5;
   // R min=f.mini(-1,xm,+1); // the evaluations of f needed in
   // computing function mini(...) are fast.

static R_Func Bezier(const R_Vector& x, const R_Vector& y, Z n=256);
   // Function defined by a Bezier curve with knots (x[i],y[i]).
   // Stops with cpmerror if this curve fails to be a function graph.
   // The final function is generated by interpolating (via polygon
   // method since here one should employ only monotonity preserving
   // interpolation methods) from computed n function values. Since
   // the method is very efficient, there is no problem in having
   // n rather large).
   // For a prper working of the algorithm we don't need x.dim()
   // to equal y.dim(). So the implementation does not test. Not
   // clarified what this means in terms of multiple knots.

static R_Func Gauss(R x0, R sigma, bool unitHeight=false);
   // Gauss bell curve with mean x0 and standard deviation sigma;
   // If unitHeight==true, the conventional normalization factor
```

```
   // is left out so that the maximum function value is 1,
   // this is sometimes useful if the function is used as a building
   // block for curve design as now implemented in class
   // R_FuncRec.

static R_Func charFunc(Iv const& iv, R soft=0);
   // For soft=0 (default) this is the
   // 'characteristic function' which takes the value 1 for x in iv
   // and the value 0 else.
   // If soft>0, the function also vanishes outside the interval
   // and takes the value 1 at the center of the interval. Let p
   // be a parameter which varies from -1 to +1 in iv (hence
   // p=iv.linear(x);) then the function value in the interior
   // of the interval is given by
   // y=exp(soft*soft*(1-r)) where r:=1/(q*q), q:=(1-p)*(1+p)
   // soft==1: no indication of a plateau in the middle
   // soft==0.01: good plateau, steep fall near the boundaries of
   // the interval.
   // The smaller soft is, the more rectangular is the function graph.
   // For large values of soft, we approximate the delta-peak (located
   // at the center of iv. However, the hight is 1 and not the
   // integral (as would be needed for a approximation of the Dirac-
   // distribution).

static R_Func potMorse(R U0, R r0, R a);
   // Morse potential as described in L.I.Schiff: Quantum Mechanics,
   // third edition,McGraw-Hill, p. 452, (49.16)

static R_Func dPotMorse(R U0, R r0, R a);
   // Derivation of the Morse potential. Analytical expression,
   // which evaluates faster than MorsePot(U0,r0,a).der().

static R_Func potLJ(R U0, R r0);
   // Lennard-Jones potential as described in Hoffmann, Schreiber
   // (Eds)
   // Computational Physics, Springer 1996, p. 270 (3)

static R_Func dPotLJ(R U0, R r0);
   // Derivation of the LJ potential. Analytical expression,
   // which evaluates faster than potLJ(U0,r0).der()

static R_Func potUM(R U0, R r0, R facU=1e4, R facR=2);
   // potUM(0)=facU*U0,potUM(r0)=-U0,potUM(facR*r0)=0
   // So, unlike the potential functions of Morse and Lennard
   // Jones, the potential is continuous at the origin and
   // the 'repulsive energy' is under control. Also
   // the range is finite and under control too.

static R_Func dPotUM(R U0, R r0, R facU=1e4, R facR=2);
   // Derivation of potUM
```

```
   static R_Func concat(const V<Iv>& v, const V<R_Func>& f);
      // We return a function g such that
      // g(x)=f[i](x) for x \in v[i]

   static R_Func Bessel(Z n);
      // Bessel function J_n of integer order

};

R_Vector gradient(const R_Vector& x, const F<R_Vector,R>& f, R h);
// Gradient of scalar function f at point x using h as a increment for
// x-coordinates. Minimum number of function evaluations for fast
// execution.
// Let gr be the return value. Then gr[i]=df(x)/dx[i].

R_Matrix gradient(const R_Vector& x, const F<R_Vector,R_Vector>& f, R h);
// Gradient of vector-valued function f at point x using h as a increment
// for x-coordinates. Minimum number of function evaluations for fast
// execution.
// Let gr be the return value. Then gr[i][j]=d/dx[i] f(x)[j].

// Bessel function of integer order, modified from Press et al.
// Here as a simple function on which the implementation of
// R_Func::Bessel(Z) is based.
R besselJ(Z n, R x);
// Bessel function J_n(x) for real x and integer n, n=0,1,2,...
// The numerical approximation is the one given in Press et al.
// with float numbers in mind, so precision might not be fully
// adequate for double.

} // namespace

#endif
```

158 cpmrfunc.cpp

```
//? cpmrfunc.cpp
//? Status of work 2008-10-25.
//?
#include <cpmrfunc.h>
#include <cpmvectormore.h>
#include <cpmgraph.h>

   using namespace CpmRoot;
   using namespace CpmArrays;
   using namespace CpmSystem;
   using namespace CpmFunctions;
   using namespace CpmGeo;
   using namespace CpmGraphics;

//////////////////////// class RomCon ////////////////////////////////

   bool RomCon::prnOn(ostream& str)const
   {
      cpmwat;
      cpmp(epsRel_);
      cpmp(epsAbs_);
      cpmp(met_);
      cpmp(len_);
      cpmp(cntMax_);
      cpmp(robErrEst_);
      return true;
   }

   bool RomCon::scanFrom(istream& str)
   {
      cpms(epsRel_);
      cpms(epsAbs_);
      cpms(met_);
      cpms(len_);
      cpms(cntMax_);
      cpms(robErrEst_);
      return true;
   }

//////////////////////// class R_Func ////////////////////////////////////

namespace{ // anonymous

R fBell(const R& x, const Iv& iv, const R& mean, const R& sigma)
{
   R iva=iv.absVal();
   R p=(x-iv.inf())*cpminv(iva);
```

```
   R y=(p-mean)*cpminv(sigma);
   return exp(-0.5*y*y);
}

R fWall(const R& x, const R4& a)
{
   Word loc("fWall(R,R4)");
   R a1=a.x1;
   R a2=a.x2;
   R a3=a.x3;
   R a4=a.x4;
   cpmassert(a1<a2,loc);
   cpmassert(a2<=a3,loc);
   cpmassert(a3<a4,loc);
   if (x<=a1 || x>=a4) return 0;
   else if (x>=a2 && x<=a3) return 1;
   else if (x>a1){
      R p=(x-a1)/(a2-a1);
      return R_Func::hill3(p);
   }
   else{
      R p=(x-a3)/(a4-a3);
      return 1-R_Func::hill3(p);
   }
}

// a class is the most flexible frame for an algorithm
class LinObj:public FncObj<R,R>{
   const R2 x2;
   const R2 y2;

// dependend quantities, to be set in the constructor
   const R1 dx; // x-difference
   const R1 m; // slope

public:
   LinObj(R2 x, R2 y): x2(x), y2(y), dx(x2[2]-x2[1]),
      m(dx==0 ? 0.5*(y2[1]+y2[2]) : (y2[2]-y2[1])/dx){}
   // the constructor does all pre-calculations, so that
   // the following function executes as fast as
   // possible

   R operator()(const R& x)const // virtual function in base
   {
      return dx==0 ? m : y2[1]+m*(x-x2[1]);
   }
};

} // anon. namespace
```

```
R_Func::R_Func(Iv iv, R mean, R sigma)
{
   *this=F3<R,Iv,R,R,R>(iv,mean,sigma)(fBell);
}

R_Func::R_Func(R4 a)
{
   *this=F1<R,R4,R>(a)(fWall);
}

R_Func::R_Func(R2 x, R2 y):Fa<R,R>(new LinObj(x,y)){}
   // most direct implementation of a R_Func constructor
   // in terms of an algorithm (here encapsulated in LinObj)

R_Func::R_Func(Iv x, Iv y):
   Fa<R,R>(new LinObj(R2(x[1],x[2]),R2(y[1],y[2]))){}


/**************** CONSTRUCTORS FROM INTERPOLATION *********************/

namespace{ // anonymus

R fR_FuncIpol
(const R& x0, const V<R>& x, const V<R>& y, const IpolControl& c )
// This is the only place where R_Func is linked to locIpol().
{
   return locIpol(x0,x,y,c);
}

} // anonymus

R_Func::R_Func(const V<R>& xIn, const V<R>& yIn, const IpolControl& cIn)
// this is the constructor on which all other interpolation constructors
// are built
{
   if (cIn.get_monoTest()){
      V<R> x=xIn;
      V<R> y=yIn;
      IpolControl c=cIn;
      prepIpol(x,y,c.get_eps());
      *this=F3<R,V<R>,V<R>,IpolControl,R>(x,y,c)(fR_FuncIpol);
   }
   else{
      *this=F3<R,V<R>,V<R>,IpolControl,R>(xIn,yIn,cIn)(fR_FuncIpol);
   }
}

R_Func::R_Func(const R_Matrix& data, const IpolControl& c)
{ *this=R_Func(data[1],data[2],c);}
```

```
R_Func::R_Func(const R_Vector& data, const IpolControl& c)
{
   Z d=data.dim();
   R_Vector xv(d);
   for (Z i=1;i<=d;i++) xv[i]=i;
   *this=R_Func(xv,data,c);
}

R_Func::R_Func(const Word& filename, const IpolControl& c)
{
   Z mL=2;
   Word loc("R_Func(Word,IpolControl)");
   static Word lastFile="not_a_filename";
   static R_Vector xv;
   static R_Vector yv;
   cpmmessage(mL,loc&" started");
   R x,y;
   if (filename!=lastFile ){
      lastFile=filename;
      IFileStream in(filename);
      const Z iMax=2000;
      R_Vector xx(iMax);
      R_Vector yy(iMax);
      Z i=0;
      while ( CpmRoot::read(x,in())==1 && CpmRoot::read(y,in())==1 ){
         i++;
         xx[i]=x;
         yy[i]=y;
         if (cpmverbose>mL)
            cpmcerr<<endl<<"i="<<i<<" x="<<x<<" y="<<y;
      }
      Z d=i;
      if (cpmverbose>=mL)
            cpmcerr<<endl<<d<<" value pairs read from file "<<-filename;
      xv=R_Vector(d);
      yv=R_Vector(d);
      for (i=1;i<=d;i++){
         xv[i]=xx[i];
         yv[i]=yy[i];
      }
   }
   *this=R_Func(xv,yv,c);
   cpmmessage(mL,loc&" done");
}

namespace{ // anonymus

R poly4Func(const R& x, const R& c0, const R& c1, const R& c2,
   const R& c3,const R& c4)
{
```

```
   return (((c4*x+c3)*x+c2)*x+c1)*x+c0;
}

} // anonymus

R_Func::R_Func(R a0, R a1, R a2, R a3, R a4)
{
   F<R,R> res=F5<R,R,R,R,R,R,R>(a0,a1,a2,a3,a4)(poly4Func);
   *this=res;
}

R_Func R_Func::polyFunc(const R_Vector& c)
{
   return F1<R,R_Vector,R>(c)(evalPoly);
}

namespace{

   R scaleFunc(const R& x, const R& k, const R& x0, const R_Func& f)
   { return f(x0+(x-x0)/k);}

   R shiftFunc(const R& x, const R& s, const R_Func& f)
   { return f(x-s);}

   R mirrorFunc(const R& x, const R& x0, const R& s, const R_Func& f)
   { return (x>x0 ? f(x0-s*(x-x0)): f(x));}

   R appendFunc(const R& x, const R_Func& f, const R& x0,
      const R_Func& g)
   { return x<=x0 ? f(x) : g(x);}

   R morphFunc(const R& x, const R_Func& f, const Iv& iv,
      const R_Func& g)
   {
      R p=iv.ramp(x);
      R x1=iv.inf()+g(p)*iv.absVal();
      return f(x1);
   }

   class SqueezeObj: public FncObj<R,R>{

      const R_Func f;
      const Iv iv1;
      const Iv iv2;
// dependent
      const R x0;
      const R2 sL;
      const R2 sL_;
      const R2 sR;
      const R2 sR_;
```

```
   const R_Func tL;
   const R_Func tR;
public:
   SqueezeObj(const R_Func& fI, const Iv& iv1I, const Iv& iv2I);
   R operator()(const R& x)const;
};

SqueezeObj::SqueezeObj(const R_Func& fI, const Iv& iv1I,
   const Iv& iv2I):
   f(fI),iv1(iv1I),iv2(iv2I),x0(iv2.mean()),sL(iv1[1],iv2[1]),
   sL_(iv1[1],x0),sR(iv2[2],iv1[2]),sR_(x0,iv1[2]),tL(sL,sL_),
   tR(sR,sR_){}

R SqueezeObj::operator()(const R& x)const
{
   if (!iv1.contains(x)) return f(x);
   else if (iv2.contains(x)) return f(x0);
   else if (x<x0) return f(tL(x));
   else return f(tR(x));
}

R squeezeLeftFunc(const R& x, const Iv& ivHost, const R& d,
   const R_Func& f)
{
   Word loc("squeezeLeftFunc(...)");
   R x1=ivHost[1];
   R x2=ivHost[2];
   R x21=x2-x1;
   R x1d=x1+d;
   cpmassert(x21>0,loc);
   cpmassert(d<x21,loc);
   if (x<=x1) return f(x);
   else if (x>=x2) return f(x);
   else if (x<=x1d) return f(x1);
   else{
      R beta=x21/(x21-d);
      R xs=x2-beta*(x2-x);
         // requirement: xs linear function of x
         // x==x1+d ==>  xs==x1
         // x==x2   ==>  xs==x2
         // this is easily seen to hold for xs as defined above
      return f(xs);
   }
}

R squeezeRightFunc(const R& x, const Iv& ivHost, const R& d,
   const R_Func& f)
{
   Word loc("squeezeLeftFunc(...)");
   R x1=ivHost[1];
```

```
   R x2=ivHost[2];
   R x21=x2-x1;
   R x2d=x2-d;
   cpmassert(x21>0,loc);
   if (d>=x21){
      cpmdebug(d);
      cpmdebug(x21);
      cpmdebug(x1);
      cpmdebug(x2);
      cpmassert(d<x21,loc);
   }
   if (x<=x1) return f(x);
   else if (x>=x2) return f(x);
   else if (x>=x2d) return f(x2);
   else{
      R beta=x21/(x21-d);
      R xs=x1+beta*(x-x1);
         // requirement: xs linear function of x
         // x==x1 ==>  xs==x1
         // x==x2-d   ==>  xs==x2
         // this is easily seen to hold for xs as defined above
      return f(xs);
   }
}

R shrinkFunc(const R& x, const Iv& ivHost, const Iv& ivSub,
   const R_Func& f)
{
   R x1=ivHost[1];
   R x2=ivHost[2];
   if (x<=x1) return f(x);
   else if (x>=x2) return f(x);
   else{
      if (x1==x2) return f(x);
      else{
         R s1=ivSub[1];
         R s2=ivSub[2];
         if (x<=s1) return f(x1);
         else if (x>=s2) return f(x2);
         else{
            if (s1==s2) return f(x);
            else{
               R_Func tr(R2(s1,s2),R2(x1,x2));
               return f(tr(x));
            }
         }
      }
   }
}
```

```
} // a.. namespace


R_Func R_Func::squeezeInConstantLeft(const Iv& ivHost, R d)const
{
   return F3<R,Iv,R,R_Func,R>(ivHost,d,*this)(squeezeLeftFunc);
}

R_Func R_Func::squeezeInConstantRight(const Iv& ivHost, R d)const
{
   return F3<R,Iv,R,R_Func,R>(ivHost,d,*this)(squeezeRightFunc);
}

R_Func R_Func::shrink(const Iv& ivHost, const Iv& ivSub)const
{
   return F3<R,Iv,Iv,R_Func,R>(ivHost,ivSub,*this)(shrinkFunc);
}

void R_Func::stretch_(R k, R x0)
{
   if (k==0.)
      cpmerror("void R_Func::scale(R k, R x0): k=0 not allowed");
   *this=F3<R,R,R,R_Func,R>(k,x0,*this)(scaleFunc);
}

void R_Func::shift_(R s)
{
   *this=F2<R,R,R_Func,R>(s,*this)(shiftFunc);
}

R_Func R_Func::mirror(R x0, R s)const
{
   return F3<R,R,R,R_Func,R>(x0,s,*this)(mirrorFunc);
}

R_Func R_Func::append(const R_Func& f, R x0)const
{
   return F3<R,R_Func,R,R_Func,R>(*this,x0,f)(appendFunc);
}

R_Func R_Func::morph(Iv iv, const R_Func& g)const

{
   return F3<R,R_Func,Iv,R_Func,R>(*this,iv,g)(morphFunc);
}


R_Func R_Func::squeezeInConstant(const Iv& ivHost, const Iv& ivIn)const
{
```

```
   return F<R,R>(new SqueezeObj(*this,ivHost,ivIn));
}

R_Vector R_Func::eval(const R_Vector& rx)const
{
   return R_Vector(apply<R,R>(*this,rx));
}

/************* DIFFERENTIATION *************************/
// slightly modified from my old calculus.cpp

namespace{
R der(R x, const R_Func& f, R h)
/* der stands for derivative. It returns f'(x). The parameter h
is the finite increment used in the numerical formula.
See Bronstein et al p. 768, Tabelle 7.13
*/
{
   R x_=x-h-h;
   R y=f(x_); x_+=h;
   y-=8*f(x_); x_+=2*h;
   y+=8*f(x_); x_+=h;
   y-=f(x_);
   y/=(12*h);
   return y;
}
// Although higher derivatives can be obtained by multiple application
// of the derivation operator, (see below) we give direct formulae for
// the second and the third derivative.

R der_2(R x, const R_Func& f, R h)
/* der_2 stands for second derivative. It returns f"(x). The parameter h
is the finite increment used in the numerical formula.
See Bronstein et al p. 768, Tabelle 7.13
*/
{
   R x_=x-h-h;
   R y=-f(x_); x_+=h;
   y+=16*f(x_); x_+=h;
   y-=30*f(x_); x_+=h;
   y+=16*f(x_); x_+=h;
   y-=f(x_);
   y/=(12*h*h);
   return y;
}

R der_3(R x, const R_Func& f, R h)
/* der_3 stands for third derivative. It returns the value of the
third derivative of f at position x. The parameter h
is the finite increment used in the numerical formula.
```

```
See Bronstein et al p. 768, Tabelle 7.13
*/
{
   R x_=x-h-h;
   R y=-f(x_); x_+=h;
   y+=2*f(x_);   x_+=2*h;
   y-=2*f(x_);   x_+=h;
   y+=f(x_);
   y/=(2*h*h*h);
   return y;
}

R der_n(Z n, R x, const R_Func& f, R h)
/*
Recursive calculation of higher derivatives. The logic of recursion
is the following: If function der_n is called for a certain value of
n (and x) then the defining block {} is worked through with this
value of n. The instructions are found to calculate this from (n-3) and
so on, till finally only n=0 or n=1 or n=2 or n=3 is needed and found.
All the memory work is done automatically due to the recursivity of the C
language.
*/
{
   R y;
   if (n<0) n=-n; // security
   if (n==0){
      y=f(x);
   }
   else if (n==1){
      y=der(x,f,h);
   }
   else if (n==2){
      y=der_2(x,f,h);
   }
   else if (n==3){
      y=der_3(x,f,h);
   }
   else{
      y=(-der_n(n-3,x-2*h,f,h)+2*der_n(n-3,x-h,f,h)-2*der_n(n-3,x+h,f,h)
         +der_n(n-3,x+2*h,f,h))/(2*h*h*h);
   }
   return y;
}

R fDer(const R& x, const R_Func& f, const Z& n, const R& h)
{
   cpmassert(h!=0.,"fDer(R,R_Func,Z,R)");
   return der_n(n,x,f,h);
}
```

```
} // anon. namespace


R_Func R_Func::der(Z n, R h)const
{
    return F3<R,R_Func,Z,R,R>(*this,n,h)(fDer);
}

R_Func R_Func::derR(Z n, R h)const

{
   R_Func res=*this;
   for (Z j=1;j<=n;j++) res=res.der(1,h);
   return res;
}

/*************** INTEGRATION **********/
// slightly modified from my old calculus.cpp

V<Word> RomCon::show()const
{
   V<Word> res;
   res<<"The RomCon object has the following data:";
   res<<"epsAbs_="+cpm(epsAbs_);
   res<<"epsRel_="+cpm(epsRel_);
   res<<"met_="+cpm(met_);
   res<<"len_="+cpm(len_);
   res<<"cntMax_="+cpm(cntMax_);
   res<<"robErrEst_="+cpm(robErrEst_);
   return res;
}

namespace{
R simpson(R ai, R bi, const R_Func& f, Z n)
/*   Integration according to Simpsons rule
   Bronsteyn p.763.
   The function f is integrated from a to b, by subdividing
   this intervall in 2*n parts. Returns
   integral from ai to bi of f with the usual sign convention for
   ai > bi
*/
{
   R a,b,sig;
   if (ai>bi){
      a=ai; b=bi; sig=1;
   }
   else if (ai<bi){
      a=bi; b=ai; sig=-1;
   }
   else return 0.;
```

```
   const R norm=1./3.;
   R h,h2,x1,x2,y0,y1=0,y2=0;
   Z j;
   if (n<=0) n=16;  // added 93-3-4 for safety
   h=(b-a)/(2*n); h2=2*h;
   if (h==0) return (b-a)*(f(b)+f(a))*0.5; // b-a!=0 and h==0 possible
   x1=a+h; x2=a+h2;
   y0= f(a)-f(b); // having here "-" allows to have a single
           // for loop
   for (j=1;j<=n;j++){
      y1+=f(x1);
      y2+=f(x2);
      x1+=h2; x2+=h2;
   }
   return h*(y0+4*y1+2*y2)*norm*sig;
}

R fIntegral(R const& x, R_Func const& f, R const& x0, R const& h)
{
   cpmassert(h!=0.,"fIntegral(R,R_Func,R,R)");
   Z n=cpmround(cpmabs((x-x0)/(2*h)));
   return simpson(x0, x, f, n);
}

} // anon. namespace

R_Func R_Func::antiDer(R x0, R h)const
{
   return F3<R,R_Func,R,R,R>(*this,x0,h)(fIntegral);
}

namespace{

class MidPoint{ // transcript of function midpnt of Press et al.
   // The comment there concerning the static variable s makes clear
   // that we actually have a stateful class.
      R s_,a_,b_;
      Z j_;
      R_Func f_;
   public:
      MidPoint(R_Func const& f, R a, R b):s_(0),a_(a),b_(b),j_(0),f_(f){}
      R next(void);
   };

   R MidPoint::next(void)
      // each call gives the result of a refined integral evaluation
   {
      const R i3=1./3;
      j_++;
      if (j_==1){
```

```
      s_=(b_-a_)*f_((a_+b_)*0.5);
   }
   else{
      Z j, it=1;
      for (j=1;j<j_-1;++j) it*=3;
      R fac=1./(3*it);
      R del=(b_-a_)*fac;
      R ddel=del+del;
      R x=a_+0.5*del;
      R sf=0;
      for (j=1;j<=it;++j){
         sf+=f_(x);
         x+=ddel;
         sf+=f_(x);
         x+=del;
      }
      sf*=fac;
      s_=(b_-a_)*sf + s_*i3;
   }
   return s_;
}

R Romberg(R_Func const& f, R a, R b,
   RomCon const& ic=RomCon())
{
   Z mL=2;
   Word loc("Romberg(R_Func,R,R,RomCon)");
   const R i9=1./9.;

   Z jMax=ic.cntMax();
   Z jK=ic.len();
   Z met=ic.met();
   R epsRel=ic.epsRel();
   R epsAbs=ic.epsAbs();
   bool robErrEst=ic.robErrEst();

   R_Vector s(jK);
   R_Vector h(jK);
   Z j;
   MidPoint mp(f,a,b);
   R hNext=1;
   R y=mp.next();
   h[1]=hNext; s[1]=y;
   R yOld=y,yNew=y;
   for (j=s.b();j<s.e();++j){
      hNext*=i9;
      h[j+1]=hNext;
      s[j+1]=mp.next();
   }
   R err=0;
```

```
      R h_ex=0;
      for (j=1;j<=jMax;++j){
         if (met==1){
            y=polyInte_4(h,s,h_ex,err);
         }
         else if (met==2){
            y=ratiInte_4(h,s,h_ex,err);
         }
         if (met==0){
            y=mp.next();
         }
         if (robErrEst || met==0){
            yOld=yNew;
            yNew=y;
            err=yNew-yOld;
         }
         R tol=CpmRootX::sup<R>(epsAbs,epsRel*cpmabs(y));
         if (cpmabs(err)<=tol){
            return y;
         }
         if (met==1 || met==2){
            hNext*=i9;
            h.in_(hNext,true);
            y=mp.next();
            s.in_(y,true);
         }
      }
      cpmwarning(loc&": too many iterations, last y returned");
      return y;
   }
}

R R_Func::romInt(R x1, R x2, RomCon const& rc)const
{
   R a,b,sig;
   if (x1<x2){
      a=x1;
      b=x2;
      sig=1;
   }
   else if (x2<x1){
      a=x2;
      b=x1;
      sig=-1;
   }
   else{
      return 0.;
   }
   return Romberg(*this,a,b,rc)*sig;
}
```

```
//////////////////// special functions ////////////////////////////////

namespace{

R hillFunc(const R& x, const R& p)
{
   if (x<=0) return(1.);
   if (x>=1) return (0.);
   R y=2.*p*(x-0.5)/(x*(x-1.));
   if (y<=0) {
      y=exp(y); y/=(1.+y);
   }
   else {
      y=exp(-y); y=1./(1.+y);
   }
   // so the argument of exp is always negativ
   // hence no overflow errors can result by calling exp.
   // Since the values of exp are never negative, 1+y is never
   // zero.
   return y;
}

R terraceFunc(const R& x, const R& k)
// Function proposed by Thomas Dera (02-1-18), slightly simlified
// via Mathematica (tderaramp.nb)
{
   if (x<=0) return 0;
   else if (x>=1) return 1;
   else{
      if (k==1.) return x;
      else if (k==0.) return 0.5;
      else{
         R k1=acos(sqrt(k));
         R k2=sqrt(1./k -1.);
         return 0.5*(1+tan((2*x-1)*k1)/k2);
      }
   }
}

R terraceFunc2(const R& x, const R& k)
{
   if (x<=0) return 0;
   else if (x>=1) return 1;
   else{
      if (k==1.) return x;
      else if (k==0.)
         return 0.5*(1+2*asin(2*x-1.)/Pi);
      else{
         R k1=sqrt(1-k*k);
```

```
         R k2=asin(k1);
         return 0.5*(1+asin((2*x-1)*k1)/k2);
      }
   }
}

R rampFunc(const R& x, const R& p){ return 1-hillFunc(x,p);}

R hillFunc3(const R& x)
{
   if (x<=0) return 1;
   else if (x>=1) return 0;
   else{
      R x_=1-x;
      return x_*x_*(2*x+1);
   }
}

R rampFunc3(const R& x){ return 1-hillFunc3(x);}

R id0(const R& x){ x; return 1;}
R id(const R& x){ return x;}
R id2(const R& x){ return x*x;}
R id3(const R& x){ return x*x*x;}
R id4(const R& x){ R y=x*x; y*=y; return y;}
R absFunc(const R& x){return cpmabs(x); }
R log10Func(const R& x){ return CpmRootX::R1(x).log10();}

} //anon. namespace

R_Func R_Func::hill(R p)
{
   return F1<R,R,R>(p)(hillFunc);
}

R_Func R_Func::ramp(R p)
{
   return F1<R,R,R>(p)(rampFunc);
}

R_Func R_Func::hill3=R_Func(hillFunc3);
R_Func R_Func::ramp3=R_Func(rampFunc3);

R_Func R_Func::terrace(R k)
{
   return F1<R,R,R>(k)(terraceFunc);
}

R_Func R_Func::terrace2(R k)
{
```

```
   return F1<R,R,R>(k)(terraceFunc2);
}

namespace{

   R2 ftBD(R p, R k, R m, Z z)
      // see Thomas Dera: ramp(TD).nb 2002-2-5.
      // Driving smoothly from one position to another
   {
      R mm=m-1;
      R km=k-1;
      R y0=mm/km;
      R x0=(y0*k)/m;
      if (z==-1){
         // freedom at the left end
         R b21=2*(1-p);
         R b22=p*p;
         R x=1-x0;
         R y=1-y0;
         return R2(x,y)*b21+R2(1,1)*b22;
      }
      else if (z==1){
         // freedom at the right end
         R b21=2*(1-p);
         R b22=p*p;
         R x=x0;
         R y=y0;
         R2 res=R2(x,y)*b21+R2(1,1)*b22;
         return res;
      }
      else{
         R pp=p*p; // for efficiency
         R b31=3*(1-p)*(1-p)*p; // Bernstein polynomials b30
            // not appearing since first point is (0,0)
         R b32=3*(1-p)*pp;
         R b33=pp*p;
         R x=x0;
         R y=y0;
         R x1=1-x0;
         R y1=1-y0;
         return R2(x,y)*b31+R2(x1,y1)*b32+R2(1,1)*b33;
      }
   }
}

R_Func R_Func::terraceBezierDera(R k, R m, bool inv, Z z, Z n,
   const IpolControl& ic)
{
   if (n<2) cpmerror("R_Func::terraceBezierDera: n<2");
   if (m>1) cpmerror("R_Func::terraceBezierDera: m>1");
```

```
   if (k>m) cpmerror("R_Func::terraceBezierDera: k>m");
   if (k<0) cpmerror("R_Func::terraceBezierDera: k<0");
   if (k==1 && m==1) return R_Func::x1;
   V<R> x(n);
   V<R> y(n);
   R dp=1./(n-1.);
   R p=0;
   for (Z i=1;i<=n;i++){
      R2 f=ftBD(p,k,m,z);
      x[i]=f.x1;
      y[i]=f.x2;
      p+=dp;
   }
   if (!inv) return R_Func(x,y,ic); else return R_Func(y,x,ic);
}

namespace{

class BerStn: public FncObj<R,R>{
   R_Vector y;
   Z1 n;
   R_Vector by;
   R2 r2;
   R1 dxInv;
      // a lot of auxiliar quantities that will be initialized
      // via a constructor. These allow to compute the
      // result of operator() very fast
public:
   BerStn(const R_Vector& y, R2 a);
   R operator()(const R&x)const;
};

BerStn::BerStn(const R_Vector& y_, R2 r2_):y(y_),n(y.dim()),
by(binomialCoefficients(n-1)),r2(r2_)
{
   Word loc("BerStn::BerStn(...)");
   if (n>1){
      for (Z i=1;i<=n;i++) by[i]*=y[i];
      R dx=r2.x2-r2.x1;
      cpmassert(dx!=0,loc);
      dxInv=1./dx;
   }
}

R BerStn::operator()(const R&x)const
{
   if (n==0) return 0.;
   else if (n==1) return y[1];
   else{
      R u=(x-r2.x1)*dxInv;
```

```
      R_Vector u1(n);
      R_Vector u2(n);
      R res=0;
      R v=1-u;
      u1[1]=1;
      u2[n]=1;
      Z i=2,j=n-1;
      while (i<=n){
         u1[i]=u1[i-1]*u;
         u2[j]=u2[j+1]*v;
         i++;
         j--;
      }
      for (i=1;i<=n;i++) res+=by[i]*u1[i]*u2[i];
      return res;
   }
}

} // anon. namespace

R_Func R_Func::Bernstein(const R_Vector& y, R2 a)
// most advanced implementation of a generating mechanism for R_Func's
// Clean separation between precomputations and evaluation
{
   return F<R,R>(new BerStn(y,a));
}

R_Func R_Func::Bezier(R_Vector const& x, R_Vector const& y, Z n)
// pattern taken from the proven function R_Func::terraceBezierDera
{
   Z mL=3;
   Word loc("R_Func::Bezier(...)");
   CPM_MA
   R2 iv(0,1);
   R_Func fx=R_Func::Bernstein(x,iv);
   R_Func fy=R_Func::Bernstein(y,iv);
   V<R> x_(n);
   V<R> y_(n);
   R dp=1./(n-1.);
   R p=0;
   for (Z i=1;i<=n;i++){
      x_[i]=fx(p);
      y_[i]=fy(p);
      p+=dp;
   }
   IpolControl ic("polygon");
      // important to have a monotonity preserving interpolation
      // method here
   ic.set_monoTest(true);
      // important here since Bezier curves are not necessarily
```

```
      // function graphs
   CPM_MZ
   return R_Func(x_,y_,ic);
}

namespace{
   R gaussFunc(const R& x, const R& x0, const R& sigmaInv,
      bool const& unitHeight)
   {
      static R fac=1./sqrt(2.*CpmRoot::Pi);
      R xr=(x-x0)*sigmaInv;
      R earg=-xr*xr*0.5;
      return unitHeight ? exp(earg) : fac*sigmaInv*exp(earg);
   }

   R charFuncAux(R const& x, Iv const& iv, R const& soft)
   {
      if (!iv.contains(x)) return 0;
      if (soft==0) return 1;// thus for soft==0
            // exactly the old code
      R p=iv.linear(x); // p in [-1,+1]
      if (p==-1 || p==1) return 0;
      R q=(p+1)*(p-1); // < 0, will be squared
      R r=1/(q*q);
      return exp(soft*soft*(1-r));
   }

   R transitFunc(R const& s, R2 const& x,R2 const& y,bool const& linear)
   {
      R dx=0.5*(x.x2-x.x1);
// first returning values for the cases where s is not
// in a well-defined transition interval
      if (dx>0){ // x[1] < x[2]
         if (s<=x.x1) return y.x1;
         if (s>=x.x2) return y.x2;
      }
      if (dx<0){ // x[2] < x[1]
         if (s<=x.x2) return y.x2;
         if (s>=x.x1) return y.x1;
      }
      R yc=0.5*(y.x1+y.x2);
      if (dx==0) return yc;
         // notice that in this case there is no order among
         // x.x1 and x.x2 and so one does not know on which side of
         // x.x1 to use y[1] and y[2]
// now s is in the transition interval
      R xc=0.5*(x.x1+x.x2), dy=0.5*(y.x2-y.x1);
      R p=(s-xc)/dx; // p varies from -1 ro +1
      if (!linear) p=p*p*p;
      return yc+dy*p;
```

```
   }

   R dTransitFunc(const R& s, const R2& x, const R2& y)
   {
      R dx=0.5*(x.x2-x.x1);
// first returning values for the cases where s is not
// in a well-defined transition interval
      if (dx>0){ // x[1] < x[2]
         if (s<=x.x1) return 0;
         if (s>=x.x2) return 0;
      }
      else if (dx<0){ // x[2] < x[1]
         if (s<=x.x2) return 0;
         if (s>=x.x1) return 0;
      }
      else return 0;
      R xc=0.5*(x.x1+x.x2);
      R dy=0.5*(y.x2-y.x1);
      R p=(s-xc)/dx;
      R p2=p*p;
      return 3*(dy/dx)*p2;
   }

   R potUMfunc(const R& r, const R& U0, const R& r0,
      const R& facU, const R& facR)
   {
      R U1=facU*U0;
      R U2=-U0;
      R r1=0;
      R r2=r0;
      if (r<=0) return U1;
      if (r<=r0) return R_Func::transit(R2(r1,r2),R2(U1,U2))(r);
      R r3=facR*r0;
      if (r<r3) return R_Func::transit(R2(r2,r3),R2(U2,0))(r);
      return 0;
   }

   R dPotUMfunc(const R& r, const R& U0, const R& r0,
      const R& facU, const R& facR)
   {
      R U1=facU*U0;
      R U2=-U0;
      R r1=0;
      R r2=r0;
      if (r<=0) return 0;
      if (r<=r0) return R_Func::dTransit(R2(r1,r2),R2(U1,U2))(r);
      R r3=facR*r0;
      if (r<r3) return R_Func::dTransit(R2(r2,r3),R2(U2,0))(r);
      return 0;
   }
```

```
   R concatFunc(const R& x, const V<Iv>& v, const V<R_Func>& f)
      // naive implementation which tries not to conect the
      // intervals and values in a natural way
   {
      Z n=v.dim();
      for (Z i=1;i<=n;i++){
         if (v[i].contains(x)) return (f[i])(x);
      }
      return 0;
   }

   R potMorseFunc(const R& r, const R& U0, const R& r0, const R& a)
   {
      const R xMin=-100;
      if (a<=0 || r<0) return 0;
      R x=(r0-r)/a;
      if (x<xMin) return 0;
      return U0*(exp(2*x)-2*exp(x));
   }

   R dPotMorseFunc(const R& r, const R& U0, const R& r0, const R& a)
   {
      const R xMin=-100;
      if (a<=0 || r<0) return 0;
      R x=(r0-r)/a;
      if (x<xMin) return 0;
      return (2*U0/a)*(exp(x)-exp(2*x));
   }

   R potLJfunc(const R& r, const R& U0, const R& r0)
   {
      if (r<=0) return 0;
      R x=r0/r;
      R y=pow(x,6);
      return 4*U0*y*(y-1);
   }

   R dPotLJfunc(const R& r, const R& U0, const R& r0)
   {
      if (r<=0) return 0;
      R x=r0/r;
      R y=pow(x,6);
      return (24*U0/r)*y*(1-2*y);
   }
}

R_Func R_Func::Gauss(R x0, R sigma, bool unitHeight)
      // Gauss bell curve with mean x0 and standard deviation sigma;
{
```

```
   R sigmaInv=cpminv(sigma);
   return F3<R,R,R,bool,R>(x0,sigmaInv,unitHeight)(gaussFunc);
}

R_Func R_Func::charFunc(Iv const& iv, R soft)
{
   return F2<R,Iv,R,R>(iv,soft)(charFuncAux);
}

R_Func R_Func::potMorse(R U0, R r0, R a)
{
   return F3<R,R,R,R,R>(U0,r0,a)(potMorseFunc);
}

R_Func R_Func::dPotMorse(R U0, R r0, R a)
{
   return F3<R,R,R,R,R>(U0,r0,a)(dPotMorseFunc);
}

R_Func R_Func::potLJ(R U0, R r0)
{
   return F2<R,R,R,R>(U0,r0)(potLJfunc);
}

R_Func R_Func::dPotLJ(R U0, R r0)
{
   return F2<R,R,R,R>(U0,r0)(dPotLJfunc);
}

R_Func R_Func::potUM(R U0, R r0, R facU, R facR)
{
   return F4<R,R,R,R,R,R>(U0,r0,facU,facR)(potUMfunc);
}

R_Func R_Func::dPotUM(R U0, R r0, R facU, R facR)
{
   return F4<R,R,R,R,R,R>(U0,r0,facU,facR)(dPotUMfunc);
}

R_Func R_Func::transit(const R2& x, const R2& y, bool linear)
{
   return F3<R,R2,R2,bool,R>(x,y,linear)(transitFunc);
}

R_Func R_Func::dTransit(const R2& x, const R2& y)
{
   return F2<R,R2,R2,R>(x,y)(dTransitFunc);
}

R_Func R_Func::concat(const V<Iv>& v, const V<R_Func>& f)
```

```
{
   Z n1=v.dim();
   Z n2=f.dim();
   cpmassert(n1==n2,"R_Func::concat(V<Iv>,V<R_Func>)");
   return F2<R,V<Iv>,V<R_Func>,R>(v,f)(concatFunc);
}

//////////////////////////// powers ////////////////////////////////////////

R_Func R_Func::x0=R_Func(id0);

R_Func R_Func::x1=R_Func(id);

R_Func R_Func::x2=R_Func(id2);

R_Func R_Func::x3=R_Func(id3);

R_Func R_Func::x4=R_Func(id4);
//////////////////////////// other useful static functions ////////////////
R_Func R_Func::abs=R_Func(absFunc);
R_Func R_Func::log10=R_Func(log10Func);

/**************** search for minimum ***********************************/
// static tools

namespace{

R sxL,sxU,xCenter,halfWidth;


R distToBoundary(R x)
{
   return (x<=xCenter ? x-sxL : sxU-x);
}

inline R parabola(R x_1, R x0, R x1,  R y_1, R y0, R y1,  Z& ind)
// we assume x1-x0 = x0-x_1 > 0 (thus everything can be coded without
// using x_1. We leave it as an argument for symmetry,
// returns the x-value for which a parabola f(x) satisfying
// y_1=f(x_1), y0=f(x0), y1=f(x1), attains a minimum. If it has no
// minimum we return 0 and give ind a indicative value (0 if the
// 'parabola' is linear, -1 if it takes a maximum)
{
   x_1; // not used see above
   R c=y1-2.*y0+y_1;
   if (c==0.){
      ind=0; return 0.;
   }
   else if (c<0.){
      ind=-1; return 0.;
```

```
   }
   else{
      ind=1; return x0-0.5*(x1-x0)*(y1-y_1)/c;
   }
}

} // anonymous namespace

R R_Func::serMin(R xL, R& xMini, R xU, R tol)
{
   static Z tryMax=3;
   static Z counterMax=10;
   static R reducStart=0.8;
   static R reducIter=0.5;

   static Z counter=0;
   static Z isLinear=0;
   static Z isConstant=0;
   static Z noMinimum=0;
   static Z miniAbove=0;
   static Z miniBelow=0;

   R x_1,x0,x1,y_1,y0,y1,xMini1,xMini2,xOld,xNew,dist,reduc;
   Z ind;
  // CpmRootX::order(xL,xU); release mode linker does not find the
  // template implementation
   // thus we do it explicitely

   if (xU<xL){ R temp=xU; xU=xL;   xL=temp;}

   Word loc("R_Func::mini(R xL, R& xMini, R xU, R tol)");

   cpmassert(xMini<=xU,loc);
   cpmassert(xL<=xMini,loc);
   sxL=xL;
   sxU=xU;
   xCenter=0.5*(sxL+sxU);
   halfWidth=(sxU-sxL)*0.5;
   x0=xMini;
   xOld=x0;
   reduc=reducStart;
Iteration:
// iteration part dealing with xMini1
   dist=distToBoundary(x0);
   dist*=reduc;
   x_1=x0-dist;
   x1=x0+dist;
   y_1=(*this)(x_1);
   y0=(*this)(x0);
   y1=(*this)(x1);
```

```
xMini1=parabola(x_1,x0,x1,y_1,y0,y1,ind);
if (ind==-1){ // try to find a new start, we would get a maximum here
   noMinimum++;
   x0=xCenter+ranVal(halfWidth);
   if (noMinimum<tryMax) goto Iteration; else goto BadEnd;
}
else if (ind==0){
   R slope=y1-y_1;
   if (slope!=0){
      isLinear++;
      x0=xCenter+ranVal(halfWidth);
      if (isLinear<tryMax){
         goto Iteration;
      }
      else{
         xMini=(slope>0 ? xL : xU);
         return (*this)(xMini);
      }
   }
   else{
      isConstant++;
      x0=xCenter+ranVal(halfWidth);
      if (isConstant<tryMax){
         goto Iteration;
      }
      else{
         xMini=xCenter;
         return (*this)(xMini);
      }
   }

}
if (xMini1>=xU){
   miniAbove++;
   x0=xCenter+ranVal(halfWidth);
   if (miniAbove<tryMax){
         goto Iteration;
   }
   else{
      xMini=xU;
      return (*this)(xMini);
  }
}
if (xMini1<=xL){
   miniBelow++;
   x0=xCenter+ranVal(halfWidth);
   if (miniBelow<tryMax){
         goto Iteration;
   }
   else{
```

```
         xMini=xL;
         return (*this)(xMini);
      }
   }
   x0=xMini1;
// iteration part dealing with xMini2, identical to previous block but
// xMini1 replaced by xMini1 (not particularly elegant !!!)
   dist=distToBoundary(x0);
   dist*=reduc;
   x_1=x0-dist;
   x1=x0+dist;
   y_1=(*this)(x_1);
   y0=(*this)(x0);
   y1=(*this)(x1);
   xMini2=parabola(x_1,x0,x1,y_1,y0,y1,ind);
   if (ind==-1){ // try to find a new start, we would get a maximum here
      noMinimum++;
      x0=xCenter+ranVal(halfWidth);
      if (noMinimum<tryMax) goto Iteration; else goto BadEnd;
   }
   else if (ind==0){
      R slope=y1-y_1;
      if (slope!=0){
         isLinear++;
         x0=xCenter+ranVal(halfWidth);
         if (isLinear<tryMax){
            goto Iteration;
         }
         else{
            xMini=(slope>0 ? xL : xU);
            return (*this)(xMini);
         }
      }
      else{
         isConstant++;
         x0=xCenter+ranVal(halfWidth);
         if (isConstant<tryMax){
            goto Iteration;
         }
         else{
            xMini=xCenter;
            return (*this)(xMini);
         }
      }

   }
   if (xMini2>=xU){
      miniAbove++;
      x0=xCenter+ranVal(halfWidth);
      if (miniAbove<tryMax){
```

```
            goto Iteration;
      }
      else{
         xMini=xU;
         return (*this)(xMini);
     }
   }
   if (xMini2<=xL){
      miniBelow++;
      x0=xCenter+ranVal(halfWidth);
      if (miniBelow<tryMax){
            goto Iteration;
      }
      else{
         xMini=xL;
         return (*this)(xMini);
     }
   }
   xNew=0.5*(xMini1+xMini2);
   if (  absVal(xNew-xOld)>tol )
   {
      counter++;
      if (counter<counterMax){
         xOld=xNew;
         x0=xNew;
         reduc*=reducIter;
         goto Iteration;
      }
      else{
         xMini=-1000;
         cpmerror("R_Func::mini(): no convergence after "&
            cpmwrite(counterMax)&" iterations");
         return 0.;
      }
   }
   else{
      xMini=xNew;
      return (*this)(xMini);
   }

BadEnd:
   xMini=-1000;
   cpmerror("R_Func::mini(): no minimum found");
   return 0;
}

R_Vector CpmFunctions::gradient(const R_Vector& x,
   const CpmFunctions::F<R_Vector,R>& f, R h)
{
   cpmassert(h!=0,"CpmFunctions::gradient(...)");
```

```
   R hInv=1./h;
   Z n=x.dim();
   R_Vector res(n);
   R_Vector xval;
   R y=f(x);

   for (Z i=1;i<=n;i++){
      xval=x;
      xval[i]+=h; // shifted on i-axis
      res[i]=(f(xval)-y)*hInv;
   }
   return res;
}

R_Matrix CpmFunctions::gradient(const R_Vector& x,
   const CpmFunctions::F<R_Vector,R_Vector>& f, R h)
{
   cpmassert(h!=0,"R_Matrix CpmFunctions::gradient(...)");
   R hInv=1./h;
   Z n=x.dim();
   R_Matrix res(n,1);
   R_Vector xval;
   R_Vector y=f(x);

   for (Z i=1;i<=n;i++){
      xval=x;
      xval[i]+=h; // shifted on i-axis
      res[i]=(f(xval)-y)*hInv;
         // this says that the first index of the result gives the
         // direction of derivation and the second index gives the
         // component of f for which the derivation is formed. This is as
         // usual d/dx[i] f(x)[j] = grad[i][j]
   }
   return res;
}

namespace{ // auxiliaries
R besselJ0(R x)
{
   const R xLim=8;
   R ax=cpmabs(x);
   if (ax < xLim) {
      R y=x*x;
      R ans1=57568490574.0+y*(-13362590354.0+y*(651619640.7
         +y*(-11214424.18+y*(77392.33017+y*(-184.9052456)))));
      R ans2=57568490411.0+y*(1029532985.0+y*(9494680.718
         +y*(59272.64853+y*(267.8532712+y*1.0))));
      return ans1/ans2;
   }
   else {
```

```
      R z=xLim/ax;
      R y=z*z;
      R xx=ax-0.785398164;
      R ans1=1.0+y*(-0.1098628627e-2+y*(0.2734510407e-4
         +y*(-0.2073370639e-5+y*0.2093887211e-6)));
      R ans2 = -0.1562499995e-1+y*(0.1430488765e-3
         +y*(-0.6911147651e-5+y*(0.7621095161e-6
         -y*0.934935152e-7)));
      return sqrt(0.636619772/ax)*(cos(xx)*ans1-z*sin(xx)*ans2);
   }
}

R besselJ1(R x)
{
   const R xLim=8;
   R ax=cpmabs(x);
   if (ax < xLim) {
      R y=x*x;
      R ans1=x*(72362614232.0+y*(-7895059235.0+y*(242396853.1
         +y*(-2972611.439+y*(15704.48260+y*(-30.16036606))))));
      R ans2=144725228442.0+y*(2300535178.0+y*(18583304.74
         +y*(99447.43394+y*(376.9991397+y*1.0))));
      return ans1/ans2;
   }
   else {
      R z=xLim/ax;
      R y=z*z;
      R xx=ax-2.356194491;
      R ans1=1.0+y*(0.183105e-2+y*(-0.3516396496e-4
         +y*(0.2457520174e-5+y*(-0.240337019e-6))));
      R ans2=0.04687499995+y*(-0.2002690873e-3
         +y*(0.8449199096e-5+y*(-0.88228987e-6
         +y*0.105787412e-6)));
      R ans=sqrt(0.636619772/ax)*(cos(xx)*ans1-z*sin(xx)*ans2);
      return x < 0.0 ? -ans : ans;
   }
}
} // end auxiliaries

R CpmFunctions::besselJ(Z n, R x)
{
   const R ACC=40.0;
   const R BIGNO=1.0e10;
   const R BIGNI=1.0e-10;
   if (n<0) cpmerror("besselJ(Z n, R x) called for negative n");
   if (n==0) return besselJ0(x);
   if (n==1) return besselJ1(x);
   Z j;
   R bj,bjm,bjp,sum,ans;
   R ax=cpmabs(x);
```

```
   if (ax == 0) return 0;
   R tox=2.0/ax;
   if (ax > n) {
      bjm=besselJ0(ax);
      bj=besselJ1(ax);
      for (j=1;j<n;j++) {
         bjp=j*tox*bj-bjm;
         bjm=bj;
         bj=bjp;
      }
      ans=bj;
   }
   else {
      Z m=2*((n+(Z)sqrt(ACC*n))/2);
      bool jsum=false;
      bjp=ans=sum=0.0;
      bj=1.0;
      for (j=m;j>0;j--) {
         bjm=j*tox*bj-bjp;
         bjp=bj;
         bj=bjm;
         if (cpmabs(bj) > BIGNO) {
            bj *= BIGNI;
            bjp *= BIGNI;
            ans *= BIGNI;
            sum *= BIGNI;
         }
         if (jsum) sum += bj;
         jsum=!jsum;
         if (j == n) ans=bjp;
      }
      sum=2.0*sum-bj;
      ans /= sum;
   }
   return x < 0 && n%2 ? -ans : ans;
}

namespace{
   R besselFunc(R const& x, Z const& n)
   { return CpmFunctions::besselJ(n,x);}
}

R_Func R_Func::Bessel(Z n)
{
   return F1<R,Z,R>(n)(besselFunc);
}
```

159 cpmrfuncrec.h

```
//? cpmrfuncrec.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_R_FUNCREC_H_
#define CPM_R_FUNCREC_H_
/*

Description:
   Defining class R_FuncRec = real function recordable.
   A descendant of class R_Func which contains (unlike R_Func
   itself) a complete set of design data (which are streamable)
   so that the I/O interface CPM_IO, and also CPM_ORDER,
   can be implemented. The tool for this is class Recordable
   from which R_FuncRec is derived.

*/

#include <cpmrfunc.h>
#include <cpmrecord.h>

namespace CpmFunctions{

   using CpmRoot::R;
   using CpmRootX::Record;
   using CpmRootX::Recordable;

//////////////////// class R_FuncRec ////////////////////////////////////

class R_FuncRec: public Recordable{ // functions as recordable objects
      // see Description above

   mutable R_Func f_;
      // dependent quantities

   virtual bool vadjust()const;
      // setting f_ if rc_ is given
      // returns true if it was successful

   typedef R_FuncRec Type;
   typedef Recordable Base;

public:
   explicit R_FuncRec(Record const& rc=Record()):Base(rc){}

   R_Func operator()(void)const{ adjust(); return f_;}
   R operator()(R const& x){ adjust(); return f_(x);}
};
```

```
/*
// The following is an example of a file content which by reading
// via Record::readFormatted would define a Record that would work as
// an argument of R_FuncRec::R_FuncRec(Record const&)
//////////////////////////////////////////////////////////////////////
   R_FuncRec
//////////////////////////////////////////////////////////////////////

R polyFac=1
   c polynomial contribution: enables a polynomial contribution to the
   c potential defined by the polynomial coefficients poly.

R bernsteinFac=0
   c Bernstein contribution: enables a contribution to the potential
   c which is a Bernstein polynomial defined by tentative y-values
   c bernstein valid at equidistant x-values

R bernsteinX=-0.5
R bernsteinY=0.5

R bezierFac=0
   c Bezier contribution: enables a contribution of a function the graph
   c of which is represented as a Bezier curve given by a list of knots
   c defined by data bezierX and bezierY

R ipolFac=0
   c interpolated contribution: enables a contribution to the potential
   c which is given by interpolating the (x,y)-values from lists ipolX
   c and ipolY under control of ipolOrd and ipolMet

R gaussFac=0.1
   c Gauss contribution: enables a contribution to the potential as a
   c list
   c Gaussian bell curves characterized by lists gaussX, gaussSigma,
   c and gaussY

R indFac=0

// Defining the data for the contributions mentioned above
Rs poly=0 0 1
Rs bernstein=0 -1 0 0 0 -1 0
Rs bezierX=-0.5 -0.25 0.20 0.25 0.30  0.5
Rs bezierY=0  -1 0.5 0.5 0.5 0
Rs ipolX=-0.5   -0.25 -0.125  0     0.125  0.25    0.5
Rs ipolY=0 0.25 0.25 -0.5 0.25 0.25 0
Rs gaussX=0
Rs gaussSigma=0.05
Rs gaussY=0.05
Rs indX=0
```

```
Rs indWidth=0
Rs indY=0

Z ipolOrd=2
   c order of interpolation: e.g. 3 for cubic. The order may be much
   c smaller than the number of points; nevertheless the interpolated
   c curve will really pass through all points precisely.

Z ipolMet=2
   c interpolation method: 0,2 smooth polynomial,
   c 1 smooth rational, 3 normal (non-smooth) linear.
   c smooth interpolation is a propriatory method that behaves like
   c splines but guarantees that derivatives of all orders exist
   c (and thus are continuous)

*/
} // namespace

#endif
```

160 cpmrfuncrec.cpp

```
//? cpmrfuncrec.cpp
//? Status of work 2008-10-25.
//?
#include <cpmrfuncrec.h>

   using namespace CpmRoot;
   using namespace CpmRootX;
   using namespace CpmArrays;
   using namespace CpmSystem;
   using namespace CpmFunctions;
   using namespace CpmGeo;

bool R_FuncRec::vadjust(void)const
{
   Z mL=3;
   Word loc="R_FuncRec::vadjust()";
   CPM_MA
   Word sec="R_FuncRec";

   Z ipolMet, ipolOrd;
   R polyFac,bernsteinFac,bernsteinX, bernsteinY,bezierFac,ipolFac,
      gaussFac,indFac;
   V<R> poly,bernstein,bezierX,bezierY,ipolX,ipolY,
      gaussX,gaussY,gaussSigma,indX,indY,indWidth;

// efficiency asks for ordering according to increasing type complexity
// and alphabetic order within a type (as is obeyed in the following)

// type Z
   cpmrc(ipolMet);
   cpmrc(ipolOrd);

// type R
   cpmrc(bernsteinFac);
   cpmrc(bernsteinX);
   cpmrc(bernsteinY);
   cpmrc(bezierFac);
   cpmrc(gaussFac);
   cpmrc(indFac);
   cpmrc(ipolFac);
   cpmrc(polyFac);

// type V<R>
   cpmrc(bernstein);
   cpmrc(bezierX);
   cpmrc(bezierY);
   cpmrc(gaussSigma);
```

```
cpmrc(gaussX);
cpmrc(gaussY);
cpmrc(indWidth);
cpmrc(indX);
cpmrc(indY);
cpmrc(ipolX);
cpmrc(ipolY);
cpmrc(poly);

Z i;
if (polyFac !=0.){ // polynomial contribution
   V<R> pcHead("",0.,1.);
      // no rescaling of the polynomial here since this
      // should be done for the whole function
   poly=pcHead&poly;
   R_Vector rpoly(poly);
   R_Func fPoly=R_Func::polyFunc(rpoly)*polyFac;
   f_+=fPoly;
}
if (bernsteinFac !=0){
      R_Vector rbernstein(bernstein);
      R2 r2(bernsteinX,bernsteinY);
      f_+=R_Func::Bernstein(rbernstein,r2)*bernsteinFac;
}
if (bezierFac!=0){
   Z bX=bezierX.dim();
   Z bY=bezierY.dim();
   cpmassert(bX==bY,loc);
      // not needed for R_Func::Bezier(rbX,rbY) to work
      // but probably violated only unintentionally
   R_Vector rbX(bezierX);
   R_Vector rbY(bezierY);
   f_+=R_Func::Bezier(rbX,rbY)*bezierFac;
}
if (ipolFac!=0){
   IpolControl ic(ipolOrd+1,ipolMet);
      // this class does not expect the order but the number of
      // adjustable parameters, which always is higher by one
   f_+=R_Func(ipolX,ipolY,ic)*ipolFac;
}
if (gaussFac!=0){
   Z nX=gaussX.dim();
   Z nSigma=gaussSigma.dim();
   Z nY=gaussY.dim();
   cpmassert(nX==nY,loc);
   cpmassert(nX==nSigma,loc);
   Z nMin=CpmRootX::inf<Z>(nX,nY,nSigma);
   R_Func fGauss;
   bool unitHeight=true;
   for (i=1;i<=nMin;++i){
```

```
         fGauss+=R_Func::Gauss(gaussX[i],gaussSigma[i],unitHeight)
            *gaussY[i];
      }
      fGauss*=gaussFac;
      f_+=fGauss;
   }
   if (indFac!=0){
      Z nX=indX.dim();
      Z nWidth=indWidth.dim();
      Z nY=indY.dim();
      cpmassert(nX==nY,loc);
      cpmassert(nX==nWidth,loc);
      Z nMin=CpmRootX::inf<Z>(nX,nY,nWidth);
      R_Func fInd;
      for (i=1;i<=nMin;++i){
         R xi=indX[i];
         R hwi=0.5*indWidth[i];
         Iv ii(xi-hwi,xi+hwi);
         fInd+=R_Func::charFunc(ii)*indY[i];
      }
      fInd*=indFac;
      f_+=fInd;
   }
   CPM_MZ
   return true;
}
```

161 cpmrigbodydyn.h

```
//? cpmrigbodydyn.h
//? Status of work 2008-10-25.
//?
/*
  Describing: Rigid Body Dynamics: a simple system of a few particles
  for simple demonstrations without the ballast of the full-blown system.
  Only three dimensions to get simple file structure.
*/
#include <cpminifilebasapp.h>
#include <cpmcompar.h>
#include <cpmp.h>

namespace CpmPala1_3{
   // Topics namespace, now integrated into PaLa as CpmPala1_3
   // last number is dimension index, presently 2D not implemented

   using namespace CpmStd;

   using CpmDim3::Vec;
   using CpmDim3::AxVec;
   using CpmDim3::Spc;
   using CpmDim3::Group;
   using CpmDim3::RefSys;
   using CpmDim3::SubsetOfSpace;
   using CpmDim3::BndSph;
   using CpmDim3::Plane;
   using CpmDim3::InTens;
   using CpmDim3::Rdim;
   using CpmDim3::Texture;

   using CpmCamera3::Camera;

   using CpmRigidBody3::RigidObject;

   using CpmRigidBody3::PointPar;
   using CpmRigidBody3::PointParC;
   using CpmRigidBody3::PointParV;

   using CpmRigidBody3::ComPar;
   using CpmRigidBody3::ComParC;
   using CpmRigidBody3::ComParV;

   using CpmPhysics3::Force;
   using CpmPhysics3::ForTor;
   using CpmPhysics3::FieldSpace;

   using CpmProperties::ParticleProperties;
```

```
   using CpmProperties::NuggetData;
   using CpmProperties::SysControl;

   using CpmGraphics::Graph;
   using CpmGraphics::Frame;

   using CpmImaging::Color;
   using CpmImaging::Palette;
   using CpmImaging::INVISIBLE;

   using CpmAlgorithms::CyclicAlarm;
   using CpmApplication::IniFileBasedApplication;

   using CpmRoot::Word;
   using CpmRoot::R;
   using CpmRoot::Z;

   using CpmRootX::Z1;
   using CpmRootX::R1;
   using CpmRootX::B;
   using CpmRootX::RecordHandler;
   using CpmRootX::Record;

   using CpmArrays::R_Vector;
   using CpmArrays::V;
   using CpmArrays::Vp;
   using CpmArrays::Vo;

   using CpmFunctions::F;

   using CpmSystem::OFileStream;
   using CpmSystem::IFileStream;

//////////////////////////////// class RigBodyDyn ///////////////////////

#if defined(CPM_POINT_PARTICLES)

typedef PointPar ParT;
typedef Force ForT;
typedef PointParC ParCT;
typedef PointParV ParVT;

#else

typedef ComPar ParT;
typedef ForTor ForT;
typedef ComParC ParCT;
typedef ComParV ParVT;

#endif
```

```
struct DynCon{ // dynamics control

   Z stepsForTypTime;
   R relTotTime;
   R fitFactor;
   R tWait;
      // wait time for showing frames in function run(Graph)
      // So better rather short such as 0.02
   Z viwPer;
      // viewing period. Only each viwPer_'th computed step will
      // be shown on screen
   Z filPer;
      // filing period. Only each filPer_'th computed step will
      // be written to file
   Z imgPer;
      // filing period. Only each imgPer_'th computed step will
      // be written to image (bitmap)

   R trapPow;
      // power of trap potential

   R lambda;
      // coupling constant of trap potential

   FieldSpace fs;
      // combined homogeneous electric magnetic and
      // gravitational field

   R aCritRel;
      // SysControl::aCrit will be set to aCritRel times
      // the unit acceleration in the natural units of the system

   R vCritRel;
      // SysControl::vCrit will be set to vCritRel times
      // the unit velocity in the natural units of the system

   Z nRunsForOrd;
      // If this is >1 it asks for this number of runs, where always
      // the next run doubles the number of steps but covers the
      // same period of time. The maximum deviation values that get
      // normally reported for any run get listed for these more
      // runs here and a polynomial growth law extracted

   typedef DynCon Type;
   CPM_IO
   DynCon(){}
   DynCon(RecordHandler const& rc, Word const& sec);
      // constructor from objects that can conveniently be initialized
      // from text
```

```
};

struct TypQun{ // 'typical quantities'
   R1 enr,len,mass,time,vel;
      // energy, length, mass, time, velocity
   Spc cenOfMass;
      // center of mass
   SubsetOfSpace bonSph;
      // bounding sphere
};

class RigBodyDyn{ // rigid body dynamics

   DynCon dc_;
      // dynamics control

   V<ParT> vp_;

   V<ForT> accPar_;
      // only used if SysControl::accMem==true

   Z1 stepCon_;
      // step count, initialized as 0
   R1 t_,dt_;
      // time and time step
   R1 enr_;
      // total energy
   Vec mom_;
      // total momentum
   AxVec angMom_;
      // total angular momentum
   CyclicAlarm viws_;
   CyclicAlarm fils_;
   CyclicAlarm imgs_;

   F<Spc,Vec> trapAcc_;
      // trapping acceleration. This is a straightforward mechanism
      // for enforcing confined motion (i.e. motion that lets all
      // particles remain within a certain volume of space. The force
      // on particle vp_[i] is trapAcc_(xi)*mi, where
      // xi=vp_[i].get_x(); mi=vp_[i].get_m(). We will only consider
      // radially symmetric fields which don't invalidate the constancy
      // of the angular momentum which we are interested in as a good
      // test for the accuracy of the dynamics

   F<Spc,R> trapPot_;
      // contribution to the potential energy of the trapping field
      // introduced above. For particle vp_[i] this contribution is
      // trapPot_(xi)*mi with xi and mi defined as above.
// Dependent quantities
```

```
   Z n_;
      // number of particles

   void adjust_();

public:
   typedef RigBodyDyn Type;
   virtual RigBodyDyn* clone()const{ return new RigBodyDyn(*this);}
   CPM_IO_V
   RigBodyDyn(){}
   RigBodyDyn(DynCon const& dc, V<ParT> const& vp):dc_(dc),vp_(vp)
   {adjust_();}
   virtual void step_(R h);
   void run(Camera& cam, ostream& str, Word const& name);
   void play(Camera& cam, ostream& str);
//   bool autoStep()const{ return dc_.autoStep_;}
   R t()const{ return t_;}
   R dt()const{ return dt_;}
   R enr()const{ return enr_;}
   Vec mom()const{ return mom_;}
   AxVec angMom()const{ return angMom_;}
   void adjustStep_(){}
      // should set dt_ to a value for which a step_ is OK
      // not yet implemented

   DynCon dynCon()const{ return dc_;}

   bool viwAlarm_()
   {  return viws_();}
      // the logic is: if dc_.dt_ has a useful value it will
      // be used (together with nSteps_) to determine a time span,
      // that controls function run,  otherwise all control is
      // based on nSteps_ alone.

   bool filAlarm_()
   {  return fils_();}

   bool imgAlarm_()
   {  return imgs_();}

   Z stepCon()const{ return stepCon_;}

   void updConQun_();
      // update conserved quantities

   TypQun typQun()const;
      // typical energy,mass,length,time, velocity determined by the
      // system
```

```
   BndSph bndSph()const;
      //
};

class App1RigBodyDyn:public CpmApplication::IniFileBasedApplication{//appl

public:
   App1RigBodyDyn(const Word& w/*, const Word& path*/):
      IniFileBasedApplication(w /*,path*/){}
   void doTheWork();
};

} // namespace
```

162 cpmrigbodydyn.cpp

```
//? cpmrigbodydyn.cpp
//? Status of work 2008-10-25.
//?
#include <cpmrigbodydyn.h>
#include <cpmparens.h>
#include <cpmframes.h>
#include <cpmgreg.h>
#include <cpmvectormore.h>

using namespace CpmPala1_3;
using namespace CpmDim3;

using CpmProperties::ParEnsPrp;
using CpmFunctions::F4;
using CpmGraphics::Frames;
using CpmTime::Greg;
using CpmPhysics3::VecE;
using CpmPhysics3::VecBH;
using CpmPhysics3::VecG;

////////////////////// struct DynCon ////////////////////////////////
DynCon::DynCon(RecordHandler const& rch, Word const& sec)
// written such that for each item not readable from file
// a useful default value is provided
{
   V<R> fieldE(3), fieldH(3), fieldG(3);
   // initialized
   Z sof=1;
   stepsForTypTime=200;
   rch.read(sec,"stepsForTypTime",stepsForTypTime,sof);
   relTotTime=1;
   rch.read(sec,"relTotTime",relTotTime,sof);
   fitFactor=1.05;
   rch.read(sec,"fitFactor",fitFactor,sof);
   tWait=0.01;
   rch.read(sec,"tWait",tWait,sof);
   viwPer=1;
   rch.read(sec,"viewingPeriod",viwPer,sof);
   filPer=0;
   rch.read(sec,"filingPeriod",filPer,sof);
   imgPer=0;
   rch.read(sec,"imgagingPeriod",imgPer,sof);
   trapPow=4;
   rch.read(sec,"trapPow",trapPow,sof);
   lambda=0;
   rch.read(sec,"lambda",lambda,sof);
   rch.read(sec,"fieldE",fieldE,sof);
```

```
   rch.read(sec,"fieldH",fieldH,sof);
   rch.read(sec,"fieldG",fieldG,sof);
   VecE fE(fieldE);
   VecBH fH(fieldH,"H");
   VecG fG(fieldG);
   fs=FieldSpace(fE,fH,fG);
   aCritRel=-1;
   vCritRel=-1;
      // these values are OK since they disable the
      // v,a-limitation mechanism
   rch.read(sec,"aCritRel",aCritRel,sof);
   rch.read(sec,"vCritRel",vCritRel,sof);
   nRunsForOrd=1;
   rch.read(sec,"nRunsForOrd",nRunsForOrd,sof);
   if (nRunsForOrd>1){
      // in this case it is essential to compute as fast as
      // possible
      filPer=0;
      imgPer=0; // reasonable
   }
}

////////////////////// class RigBodyDyn ////////////////////////////////

namespace{

const Color cRigBodyDyn(Z(180),Z(225),Z(230));

Vec trap1Func(Spc const& sp, Spc const& s0, R const& r0, R const& p,
   R const& lambda)
{
   cpmassert(r0>0,"trap1Func(...)");
   R r0Inv=1./r0;
   Vec x=(sp-s0);
   R r=x.normalize_();
   r*=r0Inv;
   R fac=-p*r0Inv*::pow(r,p-1)*lambda;
   return x*fac;
}

R trap2Func(Spc const& sp, Spc const& s0, R const& r0, R const& p,
   R const& lambda)
{
   cpmassert(r0>0,"trap2Func(...)");
   R r0Inv=1./r0;
   Vec x=(sp-s0);
   R r=x.absVal()*r0Inv;
   return ::pow(r,p)*lambda;
}
```

```
} // namespace

void RigBodyDyn::updConQun_()
{
   mom_=Vec();
   angMom_=AxVec();
   enr_=0;
   Spc ori;
   for (Z i=1;i<=n_;++i){
      ParT pi=vp_[i];
      R mi=pi.get_m();
      Spc xi=pi.get_x();
      mom_+=pi.mom();
      angMom_+=pi.angMom(ori);
      enr_+=pi.kinEnr();
      enr_+=pi.pot(dc_.fs);
         // contribution from a homogeneous field
      R trp=trapPot_(xi)*mi;
      enr_+=trp;
         // contribution from the trapping potential
      for (Z j=i+1;j<=n_;++j) enr_+=pi.pot(vp_[j]);
   }
}

BndSph RigBodyDyn::bndSph()const
{
   using CpmRoot::absVal;
   R mass=0;
   R energy=0;
   V<Spc> vs(n_);
   V<R> masses(n_);
   R rMax=0;
   Z i;
   for (i=1;i<=n_;++i){
      ParT pi=vp_[i];
      R ri=pi.get_r();
      if (ri>rMax) rMax=ri;
      R mi=pi.get_m();
      masses[i]=mi;
      mass+=mi;
      Spc xi=pi.get_x();
      vs[i]=xi;
   }
   bool careful=true;
   Spc xcm=Spc::convex(vs,masses,careful);

   R_Vector ri(n_); // certainly an even number
   for (i=1;i<=n_;++i) ri[i]=(vs[i]-xcm).absVal();
   R length=ri.mean()+rMax;
      // here rMax is the maximum radius of a particle
```

```
      // without the rMax contribution
      // this would be 0 for a single particle
   rMax+=ri.sup();
      // now rMax is the one defining the bounding sphere
   return BndSph(xcm,rMax);
}

TypQun RigBodyDyn::typQun()const
   // typical energy,mass,length,time, velocity determined by the system
{
   using CpmRoot::absVal;
   TypQun res;
   R mass=0;
   R energy=0;
   V<Spc> vs(n_);
   V<R> masses(n_);
   R rMax=0;
   Z i;
   for (i=1;i<=n_;++i){
      ParT pi=vp_[i];
      R ri=pi.get_r();
      if (ri>rMax) rMax=ri;
      R mi=pi.get_m();
      masses[i]=mi;
      mass+=mi;
      Spc xi=pi.get_x();
      vs[i]=xi;
      energy+=pi.kinEnr();
      R trp=trapPot_(xi)*mi;
      energy+=cpmabs(trp);
      R fld=pi.pot(dc_.fs);
      energy+=cpmabs(fld);
         // contribution from the trapping potential
      for (Z j=i+1;j<=n_;++j) energy+=absVal(pi.pot(vp_[j]));
   }
   bool careful=true;
   Spc xcm=Spc::convex(vs,masses,careful);

   R_Vector ri(n_); // certainly an even number
   for (i=1;i<=n_;++i) ri[i]=(vs[i]-xcm).absVal();
   R length=ri.mean()+rMax;
      // here rMax is the maximum radius of a particle
      // without the rMax contribution
      // this would be 0 for a single particle
   rMax+=ri.sup();
      // now rMax is the one defining the bounding sphere

   cpmassert(mass>0,"RigBodyDyn::typQun()");
   R velocity=::sqrt(2*energy/mass);
   cpmassert(velocity>0,"RigBodyDyn::typQun()");
```

```
   R time=length/velocity;
   res.enr=energy;
   res.len=length;
   res.mass=mass;
   res.time=time;
   res.vel=velocity;
   res.cenOfMass=xcm;
   res.bonSph=SubsetOfSpace::sphere(xcm,rMax);
   return res;
}

void RigBodyDyn::adjust_()
{
   R p=dc_.trapPow;
   R lambda=dc_.lambda;
   n_=vp_.dim();
   for (Z i=1;i<=n_;++i) vp_[i].set_id(i);
   accPar_=V<ForT>(n_);
   BndSph bs=bndSph();
      // from the typical quantities we presently only
      // need the diameter
   R r0=bs.get_r();
   Spc x0=bs.get_x();
   trapAcc_=F4<Spc,Spc,R,R,R,Vec>(x0,r0,p,lambda)(trap1Func);
   trapPot_=F4<Spc,Spc,R,R,R,R>(x0,r0,p,lambda)(trap2Func);
   viws_=CyclicAlarm(dc_.viwPer);
   fils_=CyclicAlarm(dc_.filPer);
   imgs_=CyclicAlarm(dc_.imgPer);
   updConQun_();
}

bool RigBodyDyn::prnOn(ostream& str)const
{
   str; return true;
}

bool RigBodyDyn::scanFrom(istream& str)
{
   str; return true;
}

void RigBodyDyn::step_(R h)
// Same logic as function void CPM_FCN::steplet(R t0, R dt, const Camera&
// cam) in  cpmpardynsys.h
{
   Z mL=2;
   static Word loc("RigBodyDyn::step_(R)");
   static R hMem=0;
   static Z nMem=0;
   bool b=false;
```

```
   if (h!=hMem){
      b=true;
      hMem=h;
   }
   if (n_!=nMem){
      b=true;
      nMem=n_;
   }
   if (b) cpmvalues2("step_: dt, np",h,n_,2);
   dt_=h;
   SysControl::tStep=dt_;
   if (dt_==0.) return;
// Free evolution to the midpoint in time: direct midpoint method
   Z i,j;
   R tau1=0.5*dt_;
   R tau2=dt_-tau1;
   Z n=n_;
   for (i=1;i<=n;i++){
      vp_[i].freeStep(tau1);
      // notice that freeMot, which is used in cpmpardynsys.h
      // only differs in the handling of the argument
      vp_[i].phase1(tau1);
   }
// collecting the interaction terms at the mid-point in time
// don't move particles yet, otherwise dynamics would depend on
// numbering of particles
   V<ForT> forces(n);
      // forces[i]=force acting on particle par[i]
   for (i=1;i<=n;i++){ // first particle loop
      ParT pi=vp_[i];
      forces[i]=pi.forces(dc_.fs);
      Spc xi=pi.get_x();
      R mi=pi.get_m();
      Vec trapi=trapAcc_(xi)*mi;
         // force from a trapping potential
      forces[i].addForce(trapi);
      for (j=1;j<=n;++j){
         if (j==i) continue;
         forces[i]+=pi.forces(vp_[j]);
      }
   } // end of first particle loop
// moving the particles according to interaction
   t_+=tau1;
   for (i=1;i<=n;i++){ // second particle loop
      vp_[i].intAct(dt_,forces[i]);
         // Changes velocities according to forces.
      vp_[i].freeStep(tau2);
         // executes the shift in position and angular orientation
         // according to the full acceleration and angular acceleration
         // that the particle feels (no regularization) at this place.
```

```
      vp_[i].reg();
         // regularization of the motion by reducing the velocity
         // if this is much larger than SysControl::vCrit
   } // end of second particle loop
   t_+=tau2;
   stepCon_+=1;
   updConQun_();
   CPM_MZ
}

namespace{

   R logS(R x){ return x<1e-137 ? -137 : log10(x);}
}

void RigBodyDyn::run(Camera& cam, ostream& str,Word const& name)
{
   using namespace CpmImaging;
   using namespace CpmArrays;
   using namespace CpmSystem;

   Z mL=1;
   Word loc="RigBodyDyn::run(Camera, ostream)";
   CPM_MA

   Z n=n_;
   R tWait=dc_.tWait;

   Z stepsForTypTime=dc_.stepsForTypTime;
   R relTotTime=dc_.relTotTime;
   Z nRunsForOrd=dc_.nRunsForOrd;
   Z i,nSteps=(Z)(stepsForTypTime*relTotTime+0.5);

// showing the initial condition
   Graph g;
   g.paint(cRigBodyDyn);
   g.display();
// size of the particle system

   TypQun tq=typQun();
   R typE=tq.enr;
   R typL=tq.len;
   R typM=tq.mass;
   R typT=tq.time;
   R typV=tq.vel;
   Greg date;
   date.now();
   str<<endl<<date.getDate(false);
   str<<endl<<"This is output from run "<<name<<endl;
   str<<"Number of particles: "<<n<<endl;
```

```
   str<<
   "These are the typical quantities of the particle system (in SI units)"
   <<endl;
   str<<"energy="<<typE<<endl;
   str<<"length="<<typL<<endl;
   str<<"mass="<<typM<<endl;
   str<<"time="<<typT<<endl;
   str<<"velocity="<<typV<<endl;

// adjusting the camera to the particle system
   RefSys rs=cam.getRefSys();
   R fitFactor=dc_.fitFactor;
   Angle tilt(0);
   Angle pan(0);

   SubsetOfSpace bs=tq.bonSph;
   cam.dirTo_(bs,fitFactor,tilt,pan,rs);
      // by putting in the orientation of cam as the last argument
      // we need no additional tilt and pan
   for (i=1;i<=n;++i) vp_[i].mark(cam);
   cam.display();
   cpmwait(10*tWait);

   SysControl::vCrit=dc_.vCritRel*typV;
   R typAcc=typV/typT;
   SysControl::aCrit=dc_.aCritRel*typAcc;

   if (nSteps<1) return; // it is often useful to show the initial
   // condition without invoking dynamics

   RigBodyDyn stateMem(*this);

   R tTot=relTotTime*typT;

   R_Vector dT(nRunsForOrd);
   R_Vector devE(nRunsForOrd);
   R_Vector devM1(nRunsForOrd);
   R_Vector devM2(nRunsForOrd);
   R_Vector devM3(nRunsForOrd);
   R_Vector devA1(nRunsForOrd);
   R_Vector devA2(nRunsForOrd);
   R_Vector devA3(nRunsForOrd);

   R_Vector times, energies, mom1, mom2, mom3,
      amom1, amom2, amom3;
   Z countAct=0;
   R xRel=0.02;

   for (Z iRun=1;iRun<=nRunsForOrd;++iRun){
      *this=stateMem;
```

```
R dt=tTot/nSteps;
dT[iRun]=logS(dt);
Z nStepsMod=nSteps+2;
str<<"total time="<<tTot<<endl;
str<<"dt="<<dt<<endl;
str<<"vCrit="<<SysControl::vCrit<<
" negative values have no effect"<<endl;
str<<"aCrit="<<SysControl::aCrit<<
" negative values have no effect"<<endl;

times=R_Vector(nStepsMod);
energies=R_Vector(nStepsMod);
mom1=R_Vector(nStepsMod);
mom2=R_Vector(nStepsMod);
mom3=R_Vector(nStepsMod);
amom1=R_Vector(nStepsMod);
amom2=R_Vector(nStepsMod);
amom3=R_Vector(nStepsMod);

countAct=1;
Z countImg=0;

R e0=enr();
Vec vec=mom();
AxVec avec=angMom();

R m1=vec[1];
R m2=vec[2];
R m3=vec[3];

R am1=avec[1];
R am2=avec[2];
R am3=avec[3];

R typEinv= 1./typE;
R typMomInv= 1./(typM*typV);
R typAngMomInv= typMomInv/typL;

times[countAct]=t();
energies[countAct]=0;
mom1[countAct]=0;
mom2[countAct]=0;
mom3[countAct]=0;

amom1[countAct]=0;
amom2[countAct]=0;
amom3[countAct]=0;

R nStepsInv=1./nSteps;
```

```
      R tStart=cpmtime();

      while (stepCon_<nSteps){  // one state already registered
         // so we show nSteps more, making up nSteps+1 states and nSteps
         // integration steps
         R prog=stepCon_*nStepsInv;
         cpmprogress("Progress",prog,3,3);
         step_(dt);
         cam.clear();
// graphics part
         if (viwAlarm_()){
            countAct++;
            times[countAct]=t();

            energies[countAct]=(enr()-e0)*typEinv;
            vec=mom();
            avec=angMom();
            mom1[countAct]=(vec[1]-m1)*typMomInv;
            mom2[countAct]=(vec[2]-m2)*typMomInv;
            mom3[countAct]=(vec[3]-m3)*typMomInv;

            amom1[countAct]=(avec[1]-am1)*typAngMomInv;
            amom2[countAct]=(avec[2]-am2)*typAngMomInv;
            amom3[countAct]=(avec[3]-am3)*typAngMomInv;

            cam.clear();
            SysControl::tStep*=viws_.per();
               // for showing the velocity in vp_[i].mark(cam)
            for (i=1;i<=n;++i) vp_[i].mark(cam);
            cam.display();

            if(imgAlarm_()){
               countImg++;
               Word imf="RigBodyDyn_run";
               cam.getGraph().write(imf&cpm(countImg,3));
            }
            cpmwait(tWait);
         } // viwAlarm_
      }

      R tEnd=cpmtime();
      R tCom=tEnd-tStart;
      str<<"Main computation time ="<<tCom<<" seconds"<<endl;

      times=times.resize(countAct);
      energies=energies.resize(countAct);
      mom1=mom1.resize(countAct);
      mom2=mom2.resize(countAct);
      mom3=mom3.resize(countAct);
      amom1=amom1.resize(countAct);
```

```
   amom2=amom2.resize(countAct);
   amom3=amom3.resize(countAct);

   R mom1S=mom1.supAbs();
   R mom2S=mom2.supAbs();
   R mom3S=mom3.supAbs();
   R amom1S=amom1.supAbs();
   R amom2S=amom2.supAbs();
   R amom3S=amom3.supAbs();
   R energiesS=energies.supAbs();
   devE[iRun]=logS(energiesS);
   devM1[iRun]=logS(mom1S);
   devM2[iRun]=logS(mom2S);
   devM3[iRun]=logS(mom3S);
   devA1[iRun]=logS(amom1S);
   devA2[iRun]=logS(amom2S);
   devA3[iRun]=logS(amom3S);

 str<<endl<<"Data from run #"<<iRun<<" dt="<<dt<<endl
<<"Maximum relative deviation of Px from initial value "<<mom1S<<endl
<<"Max. relative deviation of Py from initial value "<<mom2S<<endl
<<"Max. relative deviation of Pz from initial value "<<mom3S<<endl
<<"Max. relative deviation of Lx from initial value "<<amom1S<<endl
<<"Max. relative deviation of Ly from initial value "<<amom2S<<endl
<<"Max. relative deviation of Lz from initial value "<<amom3S<<endl
<<"Max. rel. deviation of energy from init. value "<<energiesS<<endl;

   g.clearText();
   g.paint(cRigBodyDyn);
   g.display();
   g.setAutoScale(1);

   Frames frs(g,Z(2),Z(3));
   Graph g11(frs[1][1]);
   Graph g12(frs[1][2]);
   Graph g13(frs[1][3]);

   Graph g21(frs[2][1]);
   Graph g22(frs[2][2]);
   Graph g23(frs[2][3]);

   g11.setWithOrigin(0);
   g12.setWithOrigin(0);
   g13.setWithOrigin(0);

   g21.setWithOrigin(0);
   g22.setWithOrigin(0);
   g23.setWithOrigin(0);

   Word tit;
```

```
      Color c(RED);
      tit="Relative dev. of Px from initial value";
      g11.show(times,mom1,c,0,tit,xRel);

      tit="Rel. deviation of Py from init. value";
      g12.show(times,mom2,c,0,tit,xRel);

      c=Color(GREEN);
      tit="Rel. dev. of energy from init. value";
      g13.show(times,energies,c,0,tit,xRel);

      c=Color(CYAN);
      tit="Rel. dev. of Lx from init. value";
      g21.show(times,amom1,c,0,tit,xRel);

      tit="Rel. dev. of Ly from init. value";
      g22.show(times,amom2,c,0,tit,xRel);

      tit="Rel. dev. of Lz from init. value";
      g23.show(times,amom3,c,0,tit,xRel);

    //  g.write(name&"_run_"&cpm(iRun));
      nSteps*=2;
      stepCon_=0;
      cpmwait(SysControl::tWait);
   } // loop over iRun
// processing devE,....
   using CpmArrays::polyFit;
   if (nRunsForOrd>1){
      R_Vector fE=polyFit(dT,devE,1,0,1,false);

      R_Vector fM1=polyFit(dT,devM1,1,0,1,false);
      R_Vector fM2=polyFit(dT,devM2,1,0,1,false);
      R_Vector fM3=polyFit(dT,devM3,1,0,1,false);

      R_Vector fA1=polyFit(dT,devA1,1,0,1,false);
      R_Vector fA2=polyFit(dT,devA2,1,0,1,false);
      R_Vector fA3=polyFit(dT,devA3,1,0,1,false);

      R rE=fE[4];

      R rM1=fM1[4];
      R rM2=fM2[4];
      R rM3=fM3[4];

      R rA1=fA1[4];
      R rA2=fA2[4];
      R rA3=fA3[4];

   str<<endl<<
```

```
   "Order of dependence on dt for the following quantities:"
   <<endl
<<"Maximum relative deviation of Px from initial value "<<rM1<<endl
<<"Max. relative deviation of Py from initial value "<<rM2<<endl
<<"Max. relative deviation of Pz from initial value "<<rM3<<endl
<<"Max. relative deviation of Lx from initial value "<<rA1<<endl
<<"Max. relative deviation of Ly from initial value "<<rA2<<endl
<<"Max. relative deviation of Lz from initial value "<<rA3<<endl
<<"Max. rel. deviation of energy from init. value "<<rE<<endl;
   g.clearText();
   g.paint(cRigBodyDyn);
   g.display();
   Frames frs(g,Z(2),Z(3));
   Graph g11(frs[1][1]);
   Graph g12(frs[1][2]);
   Graph g13(frs[1][3]);

   Graph g21(frs[2][1]);
   Graph g22(frs[2][2]);
   Graph g23(frs[2][3]);

   g11.setWithOrigin(0);
   g12.setWithOrigin(0);
   g13.setWithOrigin(0);

   g21.setWithOrigin(0);
   g22.setWithOrigin(0);
   g23.setWithOrigin(0);

   Color c(RED);
   Word tit;
   tit="Relative dev. of Px from initial value";
   g11.show(dT,devM1,c,0,tit,xRel);

   tit="Rel. deviation of Py from init. value";
   g12.show(dT,devM2,c,0,tit,xRel);

   c=Color(GREEN);
   tit="Rel. dev. of energy from init. value";
   g13.show(dT,devE,c,0,tit,xRel);

   c=Color(CYAN);
   tit="Rel. dev. of Lx from init. value";
   g21.show(dT,devA1,c,0,tit,xRel);

   tit="Rel. dev. of Ly from init. value";
   g22.show(dT,devA2,c,0,tit,xRel);

   tit="Rel. dev. of Lz from init. value";
   g23.show(dT,devA3,c,0,tit,xRel);
```

```
      cpmwait(nRunsForOrd*SysControl::tWait);
      // reasonable to have time to study this
   }
   CPM_MZ
}

//////////////////////// class App1RigBodyDyn ////////////////////////////

namespace{
   const R rimPaLa=0.;
   const Color cPaLa(Z(150),Z(150),Z(255));
}

void CpmPala1_3::App1RigBodyDyn::doTheWork()
{
   Z mL=1;
   Word loc("CpmPala1_3::App1RigBodyDyn::doTheWork()");
   CPM_MA
   Graph g;
   g.setWithOrigin(0);
   Word sec="camera control";
   Camera cam(rch,sec,g);

   sec="syscontrol";
   SysControl(rch,sec);

   ParEnsPrp gp(rch.getRecord());
   sec="grain properties";
   R gpdC=gp.getR(sec,"dC");
   R gpdT=gp.getR(sec,"dT");
   R dgp=gpdC+gpdT;
//////////////////////////////////////////////////////////////////////////
   sec="system data";
//////////////////////////////////////////////////////////////////////////
   R facBox=1,facVol=0.1,ranBoo=0,ranSpin=0;
   Z cutTo=-1;
   bool showInertiaAxes=false;
   bool newRigBodyInt=true;
   bool balance=false;

   cpmrh(facBox);
   cpmrh(facVol);

   cpmrhf(ranBoo);
   cpmrhf(ranSpin);
   cpmrhf(cutTo);
   cpmrhf(showInertiaAxes);
   cpmrhf(balance);
   CpmRigidBody3::markComParMec=showInertiaAxes;
```

```
   CpmRigidBody3::balance=balance;

   R3 b0;
   R3 bs(dgp,dgp,dgp);
   bs*=facBox;
   Box box(b0,bs);

   Z nLim=64;
      // in order allow making many particles

   CpmRigidBody3::ParEns<ParT> pe(gp,box,facVol,nLim);
   Iv v0(-ranBoo,ranBoo);
   Iv om(0,ranSpin);
   V<ParT> vcp=pe();

   Z i,np=vcp.dim();
   for (i=1;i<=np;++i){
      Vec vb;
      vb[1]=v0.ran();
      vb[2]=v0.ran();
      vb[3]=v0.ran();
      vcp[i].boost(vb);
      AxVec avs;
      avs[1]=om.ran();
      avs[2]=om.ran();
      avs[3]=om.ran();
      vcp[i].spin(avs);
   }
   if (cutTo>0) vcp=vcp.resize(cutTo);
   DynCon dc(rch,"run control");
   RigBodyDyn rbd(dc,vcp);
   OFileStream out1(outFiles[1]);
   rbd.run(cam,out1(),programName_&"_"&runName_);
   cpmwait(tWaitWorkScreen_,2);
   CPM_MZ
}
```

163 cpmrmateig.h

```
//? cpmrmateig.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_RMATEIG_H_
#define CPM_RMATEIG_H_
/*
   cpmrmateig.h    stands for R_Matrix Eigenvectors

   Description:
      Eigenvalues and Eigenvectors of a symmetric matrix

*/
// credit: main algorithm modified from Press. et al.

#include <cpmrmatrix.h>

namespace CpmArrays{

R_Vector eigen(const R_Matrix& a, R_Matrix& eig, bool test=true,
   Z iterMax=30);
/*
Let a be a symmetric (n,n)-matrix and let eig be a defined but not
necessarily meaningful matrix. After a statement like
   R_Vector d=eigenvalues(a,eig)
eig will be a (n,n)-matrix such that
   a*eig[i]=d[i]*eig[i]
i.e. the return vector contains the eigenvalues and the matrix eig is
formed by the eigenvectors. The enumeration of eigenvalues is in
descending order (this is usefull when applying the method for
orthonormalization of function systems with respect to a given scalar
product) and the open +1/-1 phase factor of eigenvectors (if all
eigenvalues are different, which is the natural case, there are no more
degrees of freedom for the eigenvectors) is chosen such that the largest
component (in absolut value) of every eigenvector is positive. Let D :=
diag(d[1],...d[n])
then
   a=~eig*D*eig    (position of ~ is important !)
i.e. the matrix eig also performs the principal axis transformation.
We have: (~eig)*eig=eig*(~eig)=(n,n)-unit matrix. Thus this
is a special case of a singular value decomposition. However the SVD of
a will not always coincide with this decomposition ??? not yet clarified
UM 99-6-8: This is 'the' singular value decomposition if the diagonal
elements of D are non-negative. If some elements are negative, we change
the sign of the corresponding line in in the right factor eig and obtain
a=~eig*|D|*eigmod. Both, eig and eigmod are orthogonal and this, then,
is a SVD.
UM 99-12-18: if the test agument is set to true, an error-value,
```

```
indicating the accuracy of the eigenvectors and eigen values will be
written on file cpmcerr
UM 00-1-5: the test is modified such that the completeness of the
eigenvectors is tested also. This is done by testing
(~eig)*eig=eig*(~eig)=(n,n)-unit matrix.
*/

R scalar_product(const R_Vector& x, const R_Vector& y,
   const R_Vector& d);
/*
Scalar product overloaded such that a differential operator is applied to
the arguments as in Sobolev spaces
*/

void adjust_scalar_product(
// input
   const R_Vector& x, // vector used for adjusting
   R a, // norm that the vector x shall get
   R_Vector w,     // gives the relative weight of the contributions
      // to <x|x>
// output (defined but meaningless as input)
   R_Vector& d // vector defining the adjusted scalar product
);
/*
With the d obtained from this routine, we will have
   scalar_product(x,x,d) = a*a
and
   contrib[1]:contrib[2]:...:contrib[n]=w[1]:w[2]:...:w[n],
where the contributions are defined by
   scalar_product(x,x,d) = d[1]*scalar_product(x,x)
   +d[2]*scalar_product(x.der(1),x.der(1)
   +d[3]*scalar_product((x.der(1)).der(1),(x.der(1)).der(1))
      ...
   +d[n]*scalar_product(x der(1) applied n-1 times,...)
   =: contrib[1]+...+contrib[n].
*/

R_Vector projection(const R_Vector& x, const R_Matrix& f,
   const R_Vector& d, Z i);
/*
The matrix f describes in the standard way an orthogonal function system
f[1], ...f[m]. The return vector is the orthogonal projection of x on
span(f[1],...f[m]), with respect to the scalar product
<x|y> := scalar_product(x,y,d).
This is the case for all values of i except i=-1. Then the projection on
the orthogonal complement of span(f[1], ...f[m]) is returned.
Although it is easy to write down the result of such a projection (see
code), it is harder to write down an expression for the projection
operator as a matrix. This will not be tried here since it seems not to\
 be
```

```
needed.
*/

void ortho_system(const R_Matrix& fs, const R_Vector& d, R_Matrix& os);
/*
The matrix fs describes in the standard way a function system fs[1],
 ...fs[m]. We adopt the scalar product <x|y> := scalar_product(x,y,d).
By forming suitable linear combinations we obtain the orthonormal
function system os (which is defined but meaningless on input and having
the stated meaning on output) by a principal axis transformation of the
matrix
   g[i][j]=<fs[i]|fs[j]>/sqrt(<fs[i]|fs[i]><fs[j]|fs[j]>)
*/

R testR_MatEig(const R_Matrix& a);
   // performs a diagonalisation of (a+~a)/2 and outputs
   // an error sum which should be <<1

} // namespace

#endif
```

164 cpmrmateig.cpp

```
//? cpmrmateig.cpp
//? Status of work 2008-10-25.
//?
#include <cpmrmateig.h>
#include <cpmrmatrix.h>
#include <cpmvectormore.h>
#include <cpmgraph.h>

using namespace CpmRoot;
using namespace CpmSystem;
using namespace CpmArrays;
using CpmGraphics::Graph;

namespace{ // anonymous namespace

////////////// functions modified from Press et al. ///////////

void tred2(R_Matrix& a, R_Vector& d, R_Vector& e)
// all arguments may be changed
{
   const Z mL=3;
   Z n=a.dim1();
   Z n2=a.dim2();
   cpmassert(n==n2,"tred2(...)");
   R_Vector d_(n);
   d=d_;
   R_Vector e_(n);
   e=e_;

   Z l,k,j,i;
   R scale,hh,h,g,f;
   R nInv=1./n;
   for (i=n;i>=2;i--) {
      if (cpmverbose>=mL)
         cpmprogress("tred2()-loop 1",(n-i)*nInv);
      l=i-1;
      h=scale=0.0;
      if (l > 1) {
         for (k=1;k<=l;k++)
            scale += cpmabs(a[i][k]);
         if (scale == 0.0)
            e[i]=a[i][l];
         else {
            for (k=1;k<=l;k++) {
               a[i][k] /= scale;
               h += a[i][k]*a[i][k];
            }
```

```
            f=a[i][l];
            g = f>0 ? -sqrt(h) : sqrt(h);
            e[i]=scale*g;
            h -= f*g;
            a[i][l]=f-g;
            f=0.0;
            for (j=1;j<=l;j++) {
            // Next statement can be omitted if eigenvectors not wanted
               a[j][i]=a[i][j]/h;
               g=0.0;
               for (k=1;k<=j;k++)
                  g += a[j][k]*a[i][k];
               for (k=j+1;k<=l;k++)
                  g += a[k][j]*a[i][k];
               e[j]=g/h;
               f += e[j]*a[i][j];
            }
            hh=f/(h+h);
            for (j=1;j<=l;j++) {
               f=a[i][j];
               e[j]=g=e[j]-hh*f;
               for (k=1;k<=j;k++)
                  a[j][k] -= (f*e[k]+g*a[i][k]);
            }
         }
      } else
         e[i]=a[i][l];
      d[i]=h;
   }
   // Next statement can be omitted if eigenvectors not wanted
   d[1]=0.0;
   e[1]=0.0;
   // Contents of this loop can be omitted if eigenvectors not
   //      wanted except for statement d[i]=a[i][i];
   for (i=1;i<=n;i++) {
      if (cpmverbose>=mL)
         cpmprogress("tred2()-loop 2",(i-1)*nInv);
      l=i-1;
      if (d[i]) {
         for (j=1;j<=l;j++) {
            g=0.0;
            for (k=1;k<=l;k++)
               g += a[i][k]*a[k][j];
            for (k=1;k<=l;k++)
               a[k][j] -= g*a[k][i];
         }
      }
      d[i]=a[i][i];
      a[i][i]=1.0;
      for (j=1;j<=l;j++) a[j][i]=a[i][j]=0.0;
```

```
   }
}

R sign(R a, R b){ return (b<0. ? -cpmabs(a) : cpmabs(a));}

void tqli(R_Vector& d, R_Vector& e, R_Matrix& z, Z iterMax)
// d,e,z should be on input equal to d,e,a of tred2 on output
{
   const Z mL=3;
   Z n=d.dim();
   Z n2=e.dim();
   cpmassert(n==n2,"tqli(...)");

   Z m,l,iter,i,k;
   Z iterSup=0;
   R iterMaxInv=1./iterMax;
   R s,r,p,g,f,dd,c,b;

   for (i=2;i<=n;i++) e[i-1]=e[i];
   e[n]=0.0;
   R nInv=1./n;
   for (l=1;l<=n;l++) {
      if (cpmverbose>=mL){
         cpmprogress("tqli()-loop",(i-1)*nInv);
      }
      iter=0;
      do {
         for (m=l;m<=n-1;m++) {
            dd=cpmabs(d[m])+cpmabs(d[m+1]);
            if ((R)(cpmabs(e[m])+dd) == dd) break;
         }
         if (m != l) {
            if (iter>iterSup){
               iterSup=iter;
               if (cpmverbose>=mL)
                  cpmprogress("iterSup",iter*iterMaxInv,2);
            }
            if (iter++ == iterMax)
               cpmerror("Too many iterations in tqli()");
            g=(d[l+1]-d[l])/(2.0*e[l]);
            r=sqrt((g*g)+1.0);
            g=d[m]-d[l]+e[l]/(g+sign(r,g));
            s=c=1.0;
            p=0.0;
            for (i=m-1;i>=l;i--) {
               f=s*e[i];
               b=c*e[i];
               if (cpmabs(f) >= cpmabs(g)) {
                  c=g/f;
                  r=sqrt((c*c)+1.0);
```

```
                  e[i+1]=f*r;
                  c *= (s=1.0/r);
               } else {
                  s=f/g;
                  r=sqrt((s*s)+1.0);
                  e[i+1]=g*r;
                  s *= (c=1.0/r);
               }
               g=d[i+1]-p;
               r=(d[i]-g)*s+2.0*c*b;
               p=s*r;
               d[i+1]=g+p;
               g=c*r-b;
               // Next loop can be omitted if eigenvectors not wanted
               for (k=1;k<=n;k++) {
                  f=z[k][i+1];
                  z[k][i+1]=s*z[k][i]+c*f;
                  z[k][i]=c*z[k][i]-s*f;
               }
            }
            d[l]=d[l]-p;
            e[l]=g;
            e[m]=0.0;
         }
      } while (m != l);
   }
}

///////////// end functions modified from Press et al.////////

} // anonymous namespace

R_Vector CpmArrays::eigen(R_Matrix const& a, R_Matrix & eig,
   bool test, Z iterMax)
{
   const Z mL=2;
   static Word loc="CpmArrays::eigen(...)";
   CPM_MA
   Z i,n=a.dim1(),n1=a.dim2();
   if (n!=n1) cpmerror("CpmArrays::eigen(...): dimension mismatch");

   R_Vector d,e;

   eig=a;
   R eSup=eig.supAbs();
   cpmassert(eSup>0,loc);
   R eSupInv=1./eSup;
   eig*=eSupInv; // to avoid all trouble that the
   // tred2 and tqli-algorithms may have with extreme large or small
   // numbers. So far, there is no indication that there are problems
```

```
// with large or small numbers.

tred2(eig,d,e);
tqli(d,e,eig,iterMax);
V<Z> p=(-d).permutationForIncreasingOrder();
d=d.permute(p);
d*=eSup; // corrects for the scaling of the matrix

eig=eig.permuteColumns(p);
eig=~eig;
for (i=1;i<=n;i++){
   Iv ise=eig[i].infSup();
   R vmin=ise.inf(),vmax=ise.sup();
   if (cpmabs(vmin)>cpmabs(vmax)) eig[i]*=(-1.);
}
if (test){
   R tWait=1;
   cpmmessage("CpmArrays::eigen(...) test started");
   R aN=a.absVal();
   cpmmessage("The norm of the matrix under consideration is  "&
      cpm(aN));
   R err, errCum=0;
   for (i=1;i<=n;i++){
      err=(a*eig[i]-eig[i]*d[i]).absVal();
      err*=eSupInv;
         // without this normalization very large values of the
         // components of a would (completely legally!) give large
         // numbers here and would suggest the presence of
         // irregularities which don't exist
      errCum+=err;
   }
   cpmmessage("test 1: err="&cpmwrite(errCum));
   cpmwait(tWait);
   R_Matrix eigt=~eig;
   R_Matrix unit=eig.net(1);
   R_Matrix e1=eig*eigt-unit;
   R_Matrix e2=eigt*eig-unit;
   err=e1.absVal();
   errCum+=err;
   cpmmessage("test 2: err="&cpmwrite(err));
   cpmwait(tWait);
   err=e2.absVal();
   errCum+=err;
   cpmmessage("test 3: err="&cpmwrite(err));
   cpmwait(tWait);
   cpmmessage("CpmArrays::eigen(...)  test done, errCum="&
      cpmwrite(errCum));
   cpmwait(2*tWait);
}
CPM_MZ
```

```
   return d;
}

R CpmArrays::scalar_product(const R_Vector& x, const R_Vector& y,
   const R_Vector& d)
{
   R_Vector xm(x);
   R_Vector ym(y);
   R z=0.;
   R h=1;
   Z n=d.dim();
   for (Z i=1;i<=n;i++){
      z+=d[i]*(xm|ym);
      xm=xm.der(h);
      ym=ym.der(h);
   }
   return z;
}

void CpmArrays::adjust_scalar_product(const R_Vector& x, R a, R_Vector w,
   R_Vector& d)
{
   R h=1.,sum=0.;
   Z n=w.dim();
   R_Vector xm=x;
   for (Z i=1;i<=n;i++){
      sum+=(xm|xm)*w[i];
      xm=xm.der(h);
   }
   if (sum==0.) cpmerror("adjust_scalar_product: sum=0");
   R b=a*a/sum;
   d=b*w;
}

R_Vector CpmArrays::projection(const R_Vector& x,
   const R_Matrix& f, const R_Vector& d, Z i)
{
   Z m=f.dim1();
   Z n=f.dim2();
   Z n1=x.dim();
   if (n!=n1) cpmerror("projection: dimension mismatch");
    R_Vector y(n1);
   for (Z ii=1;ii<=m;ii++){
      y=y+scalar_product(x,f[ii],d)*f[ii];
   }
    if (i==-1){
      return x-y;
   }
   else{
      return y;
```

```
   }
}

void CpmArrays::ortho_system(const R_Matrix& fs, const R_Vector& d,
   R_Matrix& os)
{
   Z i,j;
   R_Matrix f=fs;
   Z m=f.dim1();
    R a;
   for (i=1;i<=m;i++){
      a=scalar_product(f[i],f[i],d);
      if (a<=0.) cpmerror(
         "ortho_system: vanishing norm of a input vector");
      a=1./sqrt(a);
      f[i]=a*f[i];
   }       // now the f[i] are unit vectors.
   R_Matrix g(m);
   for (i=1;i<=m;i++){
      for (j=1;j<=m;j++){
         g[i][j]=scalar_product(f[i],f[j],d);
      }
   }
   R_Matrix eig;
   R_Vector D=eigen(g,eig);
   for (i=1;i<=m;i++){
      if (D[i]<=0.) cpmerror("ortho_system: D[i]<=0");
   }
   f=eig*f;
   for (i=1;i<=m;i++){
      a=sqrt(1./D[i]);
      f[i]=a*f[i];
   }
   for (i=1;i<=m;i++){
      Iv ifr=f[i].infSup();
      R vmin=ifr.inf(),vmax=ifr.sup();
      if (cpmabs(vmin)>cpmabs(vmax)) f[i]=(-1.)*(f[i]);
   }  // makes the dominant part positiv
   os=f;
}

R CpmArrays::testR_MatEig(const R_Matrix& a)
{
   R_Matrix sym=a+~a;
   sym*=0.5;
   R_Matrix eig;
   R_Vector d=eigen(sym,eig);
   R_Matrix diag(d);
   R sum=0.;
   sum+=disVal(sym,~eig*diag*eig);
```

```
   R_Matrix u=sym.net(1); // unit matrix
   sum+=disVal((~eig)*eig,u);
   sum+=disVal(eig*(~eig),u);
   return sum;
}
```

165 cpmrmatrix.h

```
//? cpmrmatrix.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_RMATRIX_H_
#define CPM_RMATRIX_H_
/*

   Purpose: class of real matrices together with singular value
      decomposition

*/
#include <cpmrvector.h>
#include <cpmf.h>

namespace CpmArrays{

   using CpmFunctions::F;

//////////////////////// class R_Matrix ////////////////////////////
//
// Matrices with matrix elements of type R. This class implements the
// major algorithms associated with matrices. The most complex (and
// useful) one is forming the pseudo-inverse !a of an arbitrary
// (m,n)-matrix a. Also to be noticed: ~a is the transpose of a.
// Matrix algebra is expressed in terms of *,+,-, and even
// (a/b := a*!b, of course ).

class R_Matrix{ // real matrices, rich interface

   Vr<R_Vector> rep;

   R_Matrix(const V<R_Vector>& x):rep(x){}

public:
   static R etaInv;
      // fraction of the largest 'singular value' which is considered
      // 'intentionally different from zero'. If this is negative, a
      // smooth version of the normal pseudo-inverse is chosen.
   typedef R_Matrix Type;
   typedef R ScalarType;
   CPM_IO
   bool scanFromFreeFor(istream&);
      // FreeFor stands for free format
      // reads numbers from a stream as long as this can successfully
      // be done and changes *this into the (m,n) R_Matrix having all
      // these numbers, grouped into arrays of m, as columns. Thus n has
      // not to be known in advance, wheras m=dim() is given by *this.
```

```
   CPM_ORDER
   CPM_TEST
   CPM_DESCRIPTORS
   CPM_CONJUGATION
   CPM_SUM_M
   CPM_PRODUCT_C
   CPM_SCALAR_M
   CPM_DIFFERENCE_PLAIN
   CPM_DIVISION
   CPM_DOT_PRODUCT
      // Inversion is pseudo-inversion in our case. This is always
      // defined and very useful. An internal test on correctness of
      // inversion (actually of singular value decomposition) will be
      // done if cpmverbose>=3
/*
An explanation of the concept 'pseudo inverse' follows:
If m=n and det(v)!=0 the pseudo inverse coincides with the normal
inverse of a matrix.
The pseudo inverse of a number x is in C notation given by
   (x==0 ? 0. : 1./x), (for numerical work the condition x==0 might be
      replaced by a similar one)
The pseudo inverse !diag of a diagonal matrix diag is obtained by forming
the pseudo inverse of the diagonal elements. The pseudo invers of a
general matrix is defined via singular value decomposition (see next
function svd()).
Let
   v=iso1 * diag * (iso2~)
be singular value decomposition of matrix v, then
   iso2 * !diag * (iso1~)   (note the interchange of 1 and 2)
is the pseudo inverse of v.
Consider the function
   f(x)=length of vector (y-Ax)
for a m-vector y, a (m,n)-matrix A, and a variable n-vector x.
This function has an absolute minimum at x0=(!A)y. Among all vectors x
satisfying
   f(x)=f(x0)
x0 is the shortest one. Thus the pseudo inverse gives the best and
the most natural resolution of the equation
   y=Ax
even if an exact solution is not existing. The matrix A determines two
projection matrices:
   P_final = projection onto the range of A
   P_initial = projection onto the complement of the null space of A.
The pseudo invers !A of A allows to write these projectors as
   A*!A=P_final,   (!A)*A=P_initial.
*/

   Z dim()const{ return rep.dim();}
   Z b()const{ return rep.b();}
   Z e()const{ return rep.e();}
```

```
R_Matrix():rep(1,R_Vector(1)){}
   // creating a (1,1) zero-matrix

R_Matrix(Z m, Z n):rep(m,R_Vector(n)){}
    // creating a (m,n) zero-matrix

R_Matrix(Z m, Z n, R r):rep(m,R_Vector(n,r)){}
    // creating a (m,n) matrix with all matrix elements
    // equal to r

explicit R_Matrix(const VV<R>&);
   // evident

VV<R> toVV()const;
   // evident

V<R_Vector> toRep()const{ return rep;}

R_Matrix(Z m, const R_Vector& row):rep(m,row){}
    // creating a (m,d.dim()) matrix made out of m row-vectors d

explicit   R_Matrix(Z n):rep(n,R_Vector(n)){}
   // creating a (n,n) zero-matrix

explicit R_Matrix(R_Vector const& diag):
   rep(diag.dim(), R_Vector(diag.dim()))
{ for (Z i=1;i<=diag.dim();i++)  rep[i][i]=diag[i];}
   // creating a diagonal matrix from a list diag of diagonal elements

Z dim1()const{ return rep.dim();}
   // number of rows
Z dim2()const{ return rep[1].dim();}
   // number of columns

R_Vector& operator[](Z i){ return rep[i];}

const R_Vector& operator[](Z i)const{ return rep[i];}

R_Vector getCol(Z j)const;
   // returns the vector (rep[1][j],...rep[m][j])
   // (m=dim1())

void setCol(Z j, const R_Vector& v);
   // replaces (rep[1][j],...rep[m][j]) by
   // v[1],...v[k],rep[k+1][j],..rep[m][j], k=v.dim()
   // (in most cases k==m, the the j-th column of rep
   // gets simply replaced by v

R_Matrix permuteColumns(const V<Z>& per);
```

```
R_Matrix permuteRows(const V<Z>& per)
{ return R_Matrix( rep.permute(per));}

R_Matrix ins(Z i, const R_Vector& v)const
{ return rep.ins(i,v);}
   // Let b=a.ins(i,v);
   // Then we have
   //    b[j]=a[j]   for j<i
   //    b[j]=v      for j==i
   //    b[j]=a[j-1] for j>i
   // see also Vl<T>::ins(Z, const T&)

R_Matrix ins(Z i, const R_Matrix& v)const
{  return rep.ins(i,v.toRep());}
   // Let b=a.ins(i,v);
   // d=v.dim()
   // Then we have
   //    b[j]=a[j]   for j<i
   //    b[j+k]=v[k] for 1<=k<=d
   //    b[j]=a[j-d] for j>j+d
   // see also Vl<T>::ins(Z, const Vl<T>&)

R_Matrix resize(Z newDim)const
{ return R_Matrix(rep.resize(newDim));}

R_Matrix resize2(Z newDim2)const
{ return con().resize(newDim2).con();}
   // resizing the range of the second index
   // Could be made twice as fast by a independent implementation.
   // Here I prefer the principle to have each concept in one place.

R_Matrix eliFirst()const
{ return R_Matrix(rep.eliFirst());}
   // returs a matrix with the first line eliminated from *this

R_Matrix eliFirst2()const
{ return con().eliFirst().con();}
   // returs a matrix with the first column eliminated from *this

R_Matrix operator()(F<R,R> const& g)const;
   // returns a matrix obtained by applying function g
   // to each component of *this

R_Matrix log10()const;
   // previous function with g=log10

R_Vector operator * (const R_Vector&)const;
   // natural action of a matrix on a vector
```

```
R2 centerOfMass()const;
   // interpreting the matrix components as a mass distribution
   // we return the center of mass as a pair (i,j) of
   // (fractional) pixel coordinates of the center om mass.
   // Since the components of the matrix may be mixed
   // positiv and negativ it is more general to think about
   // a center of electric charge.

R_Vector linForm()const;
   // rep[1]&rep[2]&...&rep[n], n=rep.dim(). This allows
   // carrying over from R_Vector many useful descriptors:

R inf(Z i=0)const{ return linForm().inf(i);}
   // returns the 'ith-smallest' component. No ordering
   // for i=0. For i>0 it is the component indexed 1+i
   // of the increasingly ordered version of rep

R sup(Z i=0)const{ return linForm().sup(i);}
   // returns the 'ith-largest' largest component.
   // For i=0 fast operation, no ordering
   // For i>0 it is the component indexed rep.dim()-i
   // of the increasingly ordered version of rep

R supAbs()const;
   // returns sup{ rep[i].supAbs()} (i.e. the largest
   // absolute value of a component, sometimes called max norm)

R sum()const{ return linForm().sum();}
   // returns the sum of all components

R mean()const { return linForm().mean();}
   // arithmetic mean value of all components, fast

R med()const { return linForm().med();}
   // median value of all components

Iv infSup()const{ return linForm().infSup();}
 // returns the smallest and the largest component. Fast operation,
   // no ordering

R3 infMeanSup()const{ return linForm().infMeanSup();}
// returns the smallest component, the arithmetic mean value,
// and the largest component of *this. Fast operation, no ordering

R3 infMedSup()const{ return linForm().infMedSup();}
// returns the smallest component, the median value,
// and the largest component of *this. Achieved by ordering the
// components. If one should also need the mean value, don't call
// the previous function in addition, but simply the cheap function
// mean()
```

```
   X2<R_Matrix,Iv> adjustRange(const Iv& ivRange)const;
      // returnd is a matrix, the components of which have been
      // forced into ivRange by a common linear transformation
      // of the components of *this. The original value range
      // is also returned, since it had to be determined anyway.

// concatenation functions as in R_Vector (with the natural replacement
// R_Vector |--> R_Matrix, R |--> R_Vector)

   R_Matrix& operator &=(const R_Matrix& x1)
   { rep&=x1.rep; return *this;}
   // mutating form of concatenation

   R_Matrix& operator &=(R_Vector r)
   { rep&=r; return *this;}

   R_Matrix append2(const R_Matrix& x1)const;
      // concatenation 'with respect to the second index'

   R_Matrix operator & (const R_Matrix& x1)const
   { return R_Matrix(rep&x1.rep);}
   // concatenation

   R_Matrix operator & (const R_Vector& r)const
   { return R_Matrix(rep&r);}

   R_Matrix prepend(const R_Vector& r)const
   { return R_Matrix(rep.prepend(r));}

// formatted read and write for easy communication with
// textfiles of third parties
   bool wrtFor(ostream& str)const;
      // write formatted
   bool readFor_(istream& str);
      // read formatted
   bool wrtForTrn(ostream& str)const;
      // write formatted transposed
   bool readForTrn_(istream& str);
   // read formatted transposed

   bool toGnuPlot(Word const& fileName)const;
   //: to gnu-plot
   // create an outputfile with name 'fileName' and write into it
   // the matrix elements of *this in a manner that the file is suitable
   // as a data input file for making gnuplot graphics with them.
   // Similar to function
   // CpmArrays::toGnuPlot(Word const& fn, V<V<R2> > const& x ).
};
```

```
inline bool R_Matrix::prnOn(ostream& out)const
{ return rep.prnOn(out);}

inline bool R_Matrix::scanFrom(istream& in)
{ return rep.scanFrom(in);}

inline Z R_Matrix::com(const R_Matrix& a)const
{ return rep.com(a.rep);}

inline R_Matrix& R_Matrix::operator *=(const R& r)
{ for (Z i=1;i<=dim(); i++) rep[i]*=r;  return *this;}

inline R_Matrix R_Matrix::operator -()const
{ return R_Matrix(-rep);}

inline R_Matrix R_Matrix::ranVal(Z i)const
{ return R_Matrix(rep.ranVal(i));}

inline R_Matrix R_Matrix::testVal(Z i)const
{ return R_Matrix(rep.testVal(i));}

inline Word R_Matrix::nameOf()const
{ return Word("R_Matrix");}

inline Word R_Matrix::toWord()const
{ return rep.toWord();}

inline R R_Matrix::operator|(R_Matrix const& a)const
{ return (linForm()|a.linForm());}

////////////////////// class SVD ////////////////////////////////////////
//
// Singular Value Decomposition and functional calculus. Let a be an
// arbitrary matrix and f:R->R an arbitrary function. Then one defines
// f(a) by applying f to the singular values. This is an extension of the
// functional calculus based on a spectral representation of a matrix.
// In our setting this f(a) is obtained in the following way:
// F<R,R> f=...;
// R_Matrix a=...;
// SVD sa(a);  // does the work
// R_Matrix fa=sa(f);  // relatively easy job.
//
// If we have to vary f for fixed a, we have to do the pre-computations
// encapsulated in the SVD constructor only once. We would not get this
// benefit if we would implement the functional calculus by a member
// function of R_Matrix.

/*
Let A be a real (m,n)-matrix. Let d=MIN(m,n). A singular value
decomposition of A is a representation
```

```
   A=iso1 * diag * (iso2~)
where
   iso1 is an isometric (m,d)-matrix
   diag is a diagonal   (d,d)-matrix
   iso2 is an isometric (n,d)-matrix.
(A (i,j)-matrix m is called isometric if m~ * m = (j,j)-unit matrix,
where m~ denotes the transposed version of m. If i=j, it also satisfies
m * m~ = unit matrix (not trivial !), then it is called orthogonal.)
The diagonal components of diag will be held as a vector w.
The singular value decomposition is not unique. If all singular
values are different (most natural case) the only degree of freedom is
in the order of the singular values and in 'phasefactors' +/-1.
Here the singular values are ordered in descending order of their
absolute value.
*/

class SVD{
// singular value decomposition of real (m,n)-matrices

   Z m,n,d;
      // as in previous explanation
   R_Matrix A;
      // as in previous explanation
   R_Matrix U;
      // iso1 of previous explanation
   R_Matrix V;
      // iso2
   R_Matrix Vt;
      // iso2~
   R_Vector w;
      // diag=R_Matrix(w)
public:
   explicit SVD(const R_Matrix& a);
      // creates a instance of SVD which provides all quantities which
      // result from a singular value decomposition of a. If such an
      // instance is created once, several 'services' like finding f(a)
      // for arbitrary f:R->R can be provided without much computational
      // burden.
   R_Matrix operator()(const F<R,R>& f)const;
      // returns f(A)
// simple access functions
   R_Matrix getU()const{ return U;}
      // isometric leading factor of singular value decomosition
      // into three factors
   R_Matrix getW()const{ return R_Matrix(w);}
      // singular values as a diagonal matrix central factor of singular
      // value decomosition into three factors
   R_Vector get_w()const{ return w;}
      // list of singular values
   R_Matrix getV()const{ return V;}
```

```
      // isometric last 'factor' (actually Vt is the factor, Vt has not
      // to be isometric, however)
   R_Matrix getVt()const{ return Vt;}
      // last factor of singular value decomposition
   Z get_m()const{ return m;}
   Z get_n()const{ return n;}
   Z get_d()const{ return d;}
      // number of singular values
   R getError()const;
      // defined such that the result should deviate from 0 only by
      // numerical inaccuracies. Any deviation from the defining
      // properties of a singular value decomposition will be detected.
};

} // namespace

#endif
```

166 cpmrmatrix.cpp

```
//? cpmrmatrix.cpp
//? Status of work 2008-10-25.
//?
#include <cpmrmatrix.h>
#include <cpmc.h>
#include <cpmrfunc.h>
#include <cpmfile.h>

using namespace CpmRoot;
using namespace CpmSystem;

using CpmArrays::R_Matrix;
using CpmArrays::R_Vector;
using CpmArrays::SVD;
using CpmArrays::V;
using CpmArrays::VV;
using CpmArrays::X2;
using CpmFunctions::F;
using CpmFunctions::F2;
using CpmGeo::Iv;
using CpmLinAlg::R2;

R R_Matrix::etaInv=1e-12;

R_Matrix::R_Matrix(const VV<R>& vv):rep(vv.dim())
{
   Z m=rep.dim();
   for (Z i=1;i<=m;i++) rep[i]=R_Vector(vv[i]);
}

R_Matrix R_Matrix::permuteColumns(const V<Z>& per)
{
   Z n=dim();
   R_Matrix res(n);
   for (Z i=1;i<=n;i++) res[i]=(*this)[i].permute(per);
   return res;
}

R_Matrix R_Matrix::con(void)const
{
   Z m=dim1();
   Z n=dim2();
   R_Matrix res(n,m); // sic, (m,n) are the data for *this
   for (Z i=1;i<=m;i++) for (Z j=1;j<=n;j++) res[j][i]=rep[i][j];
   return res;
}
```

```
R_Vector R_Matrix::operator *(const R_Vector& x)const
// always defined
{
   Z m=dim1();
   Z n=dim2();
   Z nx=x.dim();
   Z nMin=(nx<=n ? nx : n);
   R_Vector res(m);
   for (Z i=1;i<=m;i++){
      R sum=0;
      for (Z j=1;j<=nMin;j++) sum+=rep[i][j]*x[j];
      res[i]=sum;
   }
   return res;
}

R_Matrix R_Matrix::operator *(const R_Matrix& a2)const
{
   Z m1=dim1();
   Z n1=dim2();
   Z m2=a2.dim1();
   Z n2=a2.dim2();
   Z nk=(n1<=m2 ? n1 : m2); // min of n1 and m2, notice that
   // (m1,n1)(m2,n2)
      // is the natural order, so the two middle terms have to interact
      // the outer indexes m1,n2 label the results of 'interaction'
   R_Matrix res(m1,n2);
   for (Z i=1;i<=m1;i++){
      for (Z j=1;j<=n2;j++){
         R sum=0;
         for (Z k=1;k<=nk;k++){
            sum+=(*this)[i][k]*a2[k][j];
         }
         res[i][j]=sum;
      }
   }
   return res;
}

R R_Matrix::supAbs()const
{
   R res=0;
   for (Z i=1;i<=dim();++i){
      R xi=rep[i].supAbs();
      if (xi>res) res=xi;
   }
   return res;
}

R_Matrix R_Matrix::append2(const R_Matrix& x1)const
```

```
{
   Z m=CpmRootX::inf<Z>(dim1(),x1.dim1());
   Z n1=dim2();
   Z n2=x1.dim2();
   Z i,j,k,n=n1+n2;
   R_Matrix res(m,n);
   for (i=1;i<=m;++i){
      for (j=1;j<=n1;++j){
         res[i][j]=rep[i][j];
      }
      for (j=1,k=n1+1; j<=n2; ++j,++k){
         res[i][k]=x1[i][j];
      }
   }
   return res;
}

R_Vector R_Matrix::getCol(Z j)const
{
   Word loc("R_Matrix::getCol(Z)");
   Z m=dim1(),n=dim2();
   cpmassert(j>=1 && j<=n,loc);
   R_Vector res(m);
   for (Z i=1;i<=m;i++) res[i]=rep[i][j];
   return res;
}

void R_Matrix::setCol(Z j, const R_Vector& v)
{
   Word loc("R_Matrix::setCol(Z,R_Vector)");
   Z m=dim1(),n=dim2();
   cpmassert(j>=1 && j<=n,loc);
   Z mm=CpmRootX::inf<Z>(m,v.dim());
   for (Z i=1;i<=mm;i++) rep[i][j]=v[i];
}

R_Matrix& R_Matrix::operator +=(const R_Matrix& a)
{
   Z n=dim();
   Z na=a.dim();
   Z nnaMin=(n<=na ? n : na);
   for (Z i=1;i<=nnaMin;i++) rep[i]+=(a.rep)[i];
   return *this;
}

R_Matrix& R_Matrix::operator -=(const R_Matrix& a)
{
   Z n=dim();
   Z na=a.dim();
   Z nnaMin=(n<=na ? n : na);
```

```
   for (Z i=1;i<=nnaMin;i++) rep[i]-=(a.rep)[i];
   return *this;
}

R_Matrix R_Matrix::operator -(const R_Matrix& a)const
 {  R_Matrix res=*this; return res-=a;}


R2 R_Matrix::centerOfMass()const
{
   Z mL=3;
   static Word loc("R_Matrix::centerOfMass()");
   cpmmessage(mL,loc&" started");
   Z m=dim();
   Z n=dim2();
   R si=0;
   R sj=0;
   R s=0;
   Z i,j;
   for (i=1;i<=m;i++){
      for (j=1;j<=n;j++){
         R val=(*this)[i][j];
         si+=i*val;
         sj+=j*val;
         s+=val;
      }
   }
   cpmassert(s!=0,loc);
   R sInv=cpminv(s);
   cpmmessage(mL,loc&" done");
   return R2(si*sInv,sj*sInv);
}

R_Vector R_Matrix::linForm()const
      // rep[1]&rep[2]&...&rep[n], n=rep.dim()
{
   Z m=dim();
   if (m<1) return R_Vector(0);
   R_Vector res=rep[1];
   for (Z i=2;i<=m;i++) res&=rep[i];
   return res;
}

namespace{
   R_Vector ffAux(R_Vector const& x, F<R,R> const& f){ return x(f);}
   //R_Vector logAux(R_Vector const& x){ return x.log10();}
}

R_Matrix R_Matrix::operator()(F<R,R> const& g)const
{
```

```
   return R_Matrix(rep(F1<R_Vector,F<R,R>,R_Vector>(g)(ffAux)));
}

R_Matrix R_Matrix::log10()const
{
   return this->operator()(CpmFunctions::R_Func::log10);
}

VV<R> R_Matrix::toVV()const
{
   Z m=dim1();
   Z n=dim2();
   Z i,j;
   VV<R> res(m,n);
   for (i=1;i<=m;i++){
      for (j=1;j<=n;j++){
         res[i][j]=rep[i][j];
      }
   }
   return res;
}

bool R_Matrix::wrtFor(ostream& str)const
      // write formatted
{
   const Word sep(" ");
   Z m=dim1();
   Z n=dim2();
   Z i,j;
   str<<m<<sep<<n<<endl;
   for (i=1;i<=m;++i){
      for (j=1;j<=n;++j){
         str<<rep[i][j]<<sep;
      }
      str<<endl;
   }
   return (str) ? true : false;
}

bool R_Matrix::readFor_(istream& str)
      // read formatted
{
   Z m=-1,n=-1;
   str>>m;
   if (m<0 || !str) return false;
   str>>n;
   if (n<0 || !str) return false;
   rep=V<R_Vector>(m,R_Vector(n));
   Z i,j;
   for (i=1;i<=m;++i){
```

```
      for (j=1;j<=n;++j){
         str>>rep[i][j];
      }
   }
   return (str) ? true : false;
}

bool R_Matrix::wrtForTrn(ostream& str)const
      // write formatted transposed
{
   const Word sep(" ");
   Z m=dim1();
   Z n=dim2();
   Z i,j;
   str<<m<<sep<<n<<endl;
   for (i=1;i<=n;++i){
      for (j=1;j<=m;++j){
         str<<rep[j][i]<<sep;
      }
      str<<endl;
   }
   return (str) ? true : false;
}

bool R_Matrix::toGnuPlot(Word const& fileName)const
{
   Z m=dim1(), n=dim2();
   CpmSystem::OFile ofi(fileName);
   for (Z i=1;i<=n;++i){
      for (Z j=1;j<=m;++j){
         ofi()<<rep[j][i]<<" ";
      }
      ofi()<<endl;
   }
   bool res=ofi.isVal();
   return res;
}

bool R_Matrix::readForTrn_(istream& str)
   // read formatted transposed
{
   Z m=-1,n=-1;
   str>>m;
   if (m<0 || !str) return false;
   str>>n;
   if (n<0 || !str) return false;
   rep=V<R_Vector>(m,R_Vector(n));
   Z i,j;
   for (i=1;i<=n;++i){
      for (j=1;j<=m;++j){
```

```
         str>>rep[j][i];
      }
   }
   return (str) ? true : false;
}

namespace{

   R fAux(const R& x, const Iv& iv1, const Iv& iv2)
   {
      return iv1.linear(x,iv2.inf(),iv2.sup());
   }

}

X2<R_Matrix,Iv> R_Matrix::adjustRange(const Iv& ivRange)const
{
   Iv ivVal=infSup();
   F<R,R> g=F2<R,Iv,Iv,R>(ivVal,ivRange)(fAux);
   return X2<R_Matrix,Iv>(operator()(g),ivVal);
}

////////////////////// new version of singular value decomp //////////////
// written such that no conversion to NR data structures is needed
// modified from Press et al.
// an more recently introduced error was corrected 2000-10-31.
// (and one more in 2003-07-14)

namespace{

R sign2(R r1, R r2)
{
   using CpmRoot::absVal;
   return ( r2 >= 0. ? absVal(r1) : -absVal(r1) );
}

R pythag(R r1, R r2)
{
   using CpmRoot::C;
   return C(r1,r2).absVal();
}

void svdcmp(R_Matrix& a, R_Vector& w, R_Matrix& v)
{
   const Z iterMax=40;

   using CpmRoot::absVal;
   using CpmRootX::sup;
   using CpmRootX::inf;
   Z mL=3;
```

```
   Word loc("svdcmp(R_Matrix,R_Vector,R_Matrix)");
   CPM_MA

   Z m=a.dim1(), n=a.dim2();

   if (!(m>=n))
      cpmerror("svdcmp(): bad frame of a");
   if (n!=w.dim())
      cpmerror("svdcmp(): bad frame of w");
   if ((n!=v.dim1()) || (n!=v.dim2()))
      cpmerror("svdcmp(): bad frame of v");

   Z flag,i,its,j,jj,k;
   Z l=0,nm=0;
   R anorm,c,f,g,h,s,scale,x,y,z;

// almost verbally Press et al. with some names replaced
   R_Vector rv1(n);
   g=scale=anorm=0.0;
   for (i=1;i<=n;i++) {
      l=i+1;
      rv1[i]=scale*g;
      g=s=scale=0.0;
      if (i <= m) {
         for (k=i;k<=m;k++) scale += absVal(a[k][i]);
         if (scale) {
            for (k=i;k<=m;k++) {
               a[k][i] /= scale;
               s += a[k][i]*a[k][i];
            }
            f=a[i][i];
            g = -sign2(sqrt(s),f);
            h=f*g-s;
            a[i][i]=f-g;
            for (j=l;j<=n;j++) {
               for (s=0.0,k=i;k<=m;k++) s += a[k][i]*a[k][j];
               f=s/h;
               for (k=i;k<=m;k++) a[k][j] += f*a[k][i];
            }
            for (k=i;k<=m;k++) a[k][i] *= scale;
         }
      }
      w[i]=scale *g;
      g=s=scale=0.0;
    //  if (i <= m && i+1 != n) { // i != n before, reported as bug
      // on the official NR bug report, corrected 2007-08-03, and
      // withdrawn 2008-01-20, since only the old form works properly
      if (i <= m && i != n) { //old and good form
         for (k=l;k<=n;k++) scale += absVal(a[i][k]);
         if (scale) {
```

```
            for (k=l;k<=n;k++) {
               a[i][k] /= scale;
               s += a[i][k]*a[i][k];
            }
            f=a[i][l];
            g = -sign2(sqrt(s),f);
            h=f*g-s;
            a[i][l]=f-g;
            for (k=l;k<=n;k++) rv1[k]=a[i][k]/h;
            for (j=l;j<=m;j++) {
               for (s=0.0,k=l;k<=n;k++) s += a[j][k]*a[i][k];
               for (k=l;k<=n;k++) a[j][k] += s*rv1[k];
            }
            for (k=l;k<=n;k++) a[i][k] *= scale;
         }
      }
      anorm=sup(anorm,(absVal(w[i])+absVal(rv1[i])));
   }
   for (i=n;i>=1;i--) {
      if (i < n) {
         if (g) {
            for (j=l;j<=n;j++) v[j][i]=(a[i][j]/a[i][l])/g;
            for (j=l;j<=n;j++) {
               for (s=0.0,k=l;k<=n;k++) s += a[i][k]*v[k][j];
               for (k=l;k<=n;k++) v[k][j] += s*v[k][i];
            }
         }
         for (j=l;j<=n;j++) v[i][j]=v[j][i]=0.0;
      }
      v[i][i]=1.0;
      g=rv1[i];
      l=i;
   }
   for (i=inf(m,n);i>=1;i--) {
      l=i+1;
      g=w[i];
      for (j=l;j<=n;j++) a[i][j]=0.0;
      if (g) {
         g=1.0/g;
         for (j=l;j<=n;j++) {
            for (s=0.0,k=l;k<=m;k++) s += a[k][i]*a[k][j];
            f=(s/a[i][i])*g;
            for (k=i;k<=m;k++) a[k][j] += f*a[k][i];
         }
         for (j=i;j<=m;j++) a[j][i] *= g;
      } else for (j=i;j<=m;j++) a[j][i]=0.0;
      ++a[i][i];
   }
   for (k=n;k>=1;k--) {
      for (its=1;its<=iterMax;its++) {
```

```
flag=1;
for (l=k;l>=1;l--) {
   nm=l-1;
   if ((absVal(rv1[l])+anorm) == anorm) {
         // if one casts to float on both sides of ==, one gets
         // indexes out of range!. Very subtle point that fooled
         // me once. Erronous version detected and corrected
         // 2000-10-30.
         // Not clear when the error was introduced. Long ago,
         // everything was OK since tested several times
      flag=0;
      break;
   }
   if ((absVal(w[nm])+anorm) == anorm) break;
      // see comment above
}
if (flag) {
   c=0.0;
   s=1.0;
   for (i=l;i<=k;i++) {
      f=s*rv1[i];
      rv1[i]=c*rv1[i];
      if ((absVal(f)+anorm) == anorm) break;
          // see comment above
      g=w[i];
      h=pythag(f,g);
      w[i]=h;
      h=1.0/h;
      c=g*h;
      s = -f*h;
      for (j=1;j<=m;j++) {
         y=a[j][nm];
         z=a[j][i];
         a[j][nm]=y*c+z*s;
         a[j][i]=z*c-y*s;
      }
   }
}
z=w[k];
if (l == k) {
   if (z < 0.0) {
      w[k] = -z;
      for (j=1;j<=n;j++) v[j][k] = -v[j][k];
   }
   break;
}
if (its == iterMax)
   cpmerror("svdcmp: no convergence in "&cpmwrite(iterMax)&
   " iterations");
x=w[l];
```

```
        nm=k-1;
        y=w[nm];
        g=rv1[nm];
        h=rv1[k];
        f=((y-z)*(y+z)+(g-h)*(g+h))/(2.0*h*y);
        g=pythag(f,1.0);
        f=((x-z)*(x+z)+h*((y/(f+sign2(g,f)))-h))/x;
        c=s=1.0;
        for (j=l;j<=nm;j++) {
           i=j+1;
           g=rv1[i];
           y=w[i];
           h=s*g;
           g=c*g;
           z=pythag(f,h);
           rv1[j]=z;
           c=f/z;
           s=h/z;
           f=x*c+g*s;
           g = g*c-x*s;
           h=y*s;
           y *= c;
           for (jj=1;jj<=n;jj++) {
              x=v[jj][j];
              z=v[jj][i];
              v[jj][j]=x*c+z*s;
              v[jj][i]=z*c-x*s;
           }
           z=pythag(f,h);
           w[j]=z;
           if (z) {
              z=1.0/z;
              c=f*z;
              s=h*z;
           }
           f=c*g+s*y;
           x=c*y-s*g;
           for (jj=1;jj<=m;jj++) {
              y=a[jj][j];
              z=a[jj][i];
              a[jj][j]=y*c+z*s;
              a[jj][i]=z*c-y*s;
           }
        }
        rv1[l]=0.0;
        rv1[k]=f;
        w[k]=x;
     }
  }
  CPM_MZ
```

```
   return;
}

//////////// end of singular value decomposition algorithm /////////////

} // anon namespace

R_Matrix R_Matrix::inv()const
{
   const Z testLimit=5;
   R sum;
   Z i, j, k, m_=dim1(), n=dim2();
   bool rightframe=(m_>=n);
   R_Matrix a=*this;
   if (!rightframe) a=~a;
   m_=a.dim1(); n=a.dim2();
   R_Matrix v(n,n);
   R_Vector w(n);
   R_Matrix vr(n,m_); //  sic
   svdcmp(a,w,v);
   if (cpmverbose>=testLimit){ // test of svdcmp,
      // svdcmp got corrupted once, so good to test
      R_Matrix u=v.net(1);
      R error1=(~a*a-u).absVal();
      cpmmessage("R_Matrix::inv(): error1= "&cpm(error1));
      R error2=(~v*v-u).absVal();
      cpmmessage("R_Matrix::inv(): error2= "&cpm(error2));
      R error3=(v*~v-u).absVal();
      cpmmessage("R_Matrix::inv(): error3= "&cpm(error3));
      R error4=(*this-a*R_Matrix(w)*~v).absVal();
      cpmmessage("R_Matrix::inv(): error4= "&cpm(error4));
      R err=error1+error2+error3+error4;
      cpmmessage(" total inaccuracy in svdcmp = "&cpmwrite(err));
   }
   w=w.pseudoInverse(etaInv);  // w does not come out orderd
      // now w is the pseudo-inverse of the original w
   for (i=1;i<=n;i++){
      for (j=1;j<=m_;j++){
         sum=0;
         for (k=1;k<=n;k++){
            sum+=v[i][k]*w[k]*a[j][k];
         }
         vr[i][j]=sum;
      }
   }
   if (!rightframe) vr=~vr;
   return vr;
}

namespace{
```

```
R_Vector svd(R_Matrix& u, R_Matrix& v)
{
   R_Matrix u_=u;
   Z m=u_.dim1(), n=u_.dim2(), i;
   if (m>=n){
      R_Matrix vv(n,n);
      v=vv;
      R_Vector w(n);
      svdcmp(u_,w,v);
      R_Vector w_(n);
      for (i=1;i<=n;i++) w_[i]=-cpmabs(w[i]);
      V<Z> p=w_.permutationForIncreasingOrder();
      u_=u_.permuteColumns(p);
      v=v.permuteColumns(p);
      w=w.permute(p);
      u=u_;
      { // making the dominant component of ~v positive
         R_Matrix vt=~v;
         R_Matrix ut=~u;
         Z nv=vt.dim1(),nu=ut.dim1();
         if (nv!=nu) cpmerror("svd: malfunction in +/- part");
         for (Z i=1;i<=nv;i++){
            Iv ivt=vt[i].infSup();
            R xmin=ivt.inf(), xmax=ivt.sup();
            if (cpmabs(xmin)>cpmabs(xmax)){
               vt[i]=(-1.)*(vt[i]);  // makes dominant part positiv
               ut[i]=(-1.)*(ut[i]); // necessary to stay with a svd
            }
         }
         u=~ut; v=~vt;
      }
      return w;
   }
   else{
      n=u_.dim1();   // now m>=n  and u is a (n,m)-matrix
      R_Matrix vv(n,n);
      v=vv;
      R_Vector w(n);
      u_=~u_; //  now u_ is a (m,n)-matrix
      svdcmp(u_,w,v);
      R_Vector w_(n);
      for (i=1;i<=n;i++) w_[i]=-cpmabs(w[i]);
      V<Z> p=w_.permutationForIncreasingOrder();
      u_=u_.permuteColumns(p);
      v=v.permuteColumns(p);
      w=w.permute(p);
      R_Matrix umem=u_; u_=v; v=umem;
      u=u_;
      { // making the dominant component of ~v positive
         R_Matrix vt=~v;
```

```
         R_Matrix ut=~u;
         Z nv=vt.dim1(), nu=ut.dim1();
         if (nv!=nu) cpmerror("svd: malfunction in +/- part");
         for (Z i=1;i<=nv;i++){
            Iv ivt=vt[i].infSup();
            R xmin=ivt.inf(),xmax=ivt.sup();
            if (cpmabs(xmin)>cpmabs(xmax)){
               vt[i]=(-1.)*(vt[i]);  // makes dominant part positiv
               ut[i]=(-1.)*(ut[i]); // necessary to stay with a svd
            }
         }
         u=~ut; v=~vt;
      }
      return w;
   }
}
}

R_Matrix R_Matrix::net(Z i)const
{
   Z m=dim1();
   Z n=dim2();
   R_Matrix res(m,n); // all zero by definition
   if (i!=1) return res;
// now i==1, thus unit matrix required
   if (m!=n) cpmwarning(
      "R_Matrix::net(Z): non-square unit matrix formed");
   Z mnMin=(m<=n ? m : n);
   for (Z j=1; j<=mnMin; j++) res[j][j]=1;
   return res;
}

bool R_Matrix::scanFromFreeFor(istream& ifs)
{
   Word loc("R_Matrix::scanFromFreeFor(istream& ifs)");
   Z mL=3;
   CPM_MA
   CpmRootX::R1 x;
   Z m=dim();
   if (m==0) return false;
   R_Vector y(m);
   std::vector<R_Vector> vec;
   Z i,j;
   bool work=true;
   while (work){
      for (i=1;i<=m;++i){
         bool succ=x.scanFrom(ifs);
         if (succ)y[i]=x;
         else{
            work=false;
```

```
            cpmmessage(loc&": last valid x was "&cpm(x));
            break;
         }
      }
      if (work) vec.push_back(y);
   }
   Z n=(Z)vec.size();
   R_Matrix res(m,n);
   for (i=1;i<=m;++i) for (j=1;j<=n;++j) res[i][j]=vec[j-1][i];
   *this=res;
   CPM_MZ
   return n>0 ? true : false;
}

//////////////////////////// class SVD ////////////////////////////////

SVD::SVD(const R_Matrix& a):m(a.dim1()),n(a.dim2()),
A(a),U(a),V()
{
   d=CpmRootX::inf(m,n);
   w=svd(U,V);
   Vt=~V;
}

R SVD::getError()const
{
   R_Matrix u1=~U*U;
   R error1=(u1-u1.net(1)).absVal();
      // deviation from isometry
   R_Matrix v1=~V*V;
   R error2=(v1-v1.net(1)).absVal();
      // deviation from isometry
   R error3=(A-U*R_Matrix(w)*Vt).absVal();
      // deviation from representing the matrix we started from
   bool b1=(U.dim1()==m);
   bool b2=(U.dim2()==d);
   bool b3=(w.dim()==d);
   bool b4=(Vt.dim1()==d);
   bool b5=(Vt.dim2()==n);
   R error4=(b1 && b2 && b3 && b4 && b5) ? 0 : 1;
      // deviation from the expected frame-type of
      // the matrices
   return error1+error2+error3+error4;
}

R_Matrix SVD::operator()(const F<R,R>& f)const
{
   R_Vector fw(d);
   for (Z i=1;i<=d;i++) fw[i]=f(w[i]);
   return U*R_Matrix(fw)*Vt;
```

```
}
```

167 cpmrnfunc.h

```
//? cpmrnfunc.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_RNFUNC_H_
#define CPM_RNFUNC_H_
/*

Description: Working out in C++ the topic from
   page 6 (03-05-14) of my yellow Konzeptheft

*/

#include <cpmfa.h>
#include <cpmrvector.h>
#include <cpmalgorithms.h>

namespace CpmFunctions{

   using CpmRoot::R;
   using CpmRootX::R1;
   using CpmLinAlg::R2;
   using CpmRoot::Z;
   using CpmFunctions::F;
   using CpmArrays::R_Vector;
   using CpmArrays::V;
   using CpmAlgorithms::bitsFromZ;

/********************** class Rn<n>  ******************************/
   // class of real n-tupels. Unlike in R_Vector, the dimension
   // of the space is fixed (by a template parameter)
   // class will form the domain of the following function class
   // Rn_Func<n>
   // Presently this is exploratory. If it should be seriously used,
   // one had to specialize Rn<n> for n=1,2,3,4,5,6
   // to CpmLinAlg::R1,CpmLinAlg::R2,CpmLinAlg::R3,...CpmLinAlg::R6

template <Z n>
class Rn: public R_Vector{ // R_Vector with specified dimension

   typedef Rn<n> Type;
   typedef R_Vector Base;
   static Z ok(Z i) // constant continuation outside the
         // primary index range
   { if (i<1) return 1; else if (i>n) return n; else return i;}

   static Type fFunc(const Z& i, const Type& a, const Type& b,
      const bool& random);
```

```
      // basic function for implementing f

public:

   Rn():Base(n){}

   Rn(const Base& x):Base(n)
   { Z i=1; while( i<=n && i<=x.dim() ){ (*this)[i]=x[i]; i++;} }

   R vol(const Type& y);
      // a.vol(b) is the product over i of
      // b[i]-a[i].
      // It is, up possibly to a sign equal to the volume of
      // the n-rectangle with corners a and b

   static R1 c(const Type& x, const Z& i){ return R1(x[ok(i)]);}
      // basic function Type x Z --> R1
      // for forming the component functions in the next class

   static R1 volFunc(const Type& x);
      // returns the product of the components of x

// means for discrete exhaustion of Type.
// The basic feature of Type is that a very general useful subset
// of type can ge characterized by only two instances of Type
// by a natural geleralization of the concept of an interval in R.
// The arguments a and b in the following two functions just play
// this role.

   static V< Type > v(Z m, const Type& a, const Type& b,
      bool random=true );
      // Let
      // Z m=...; Rn<4> a=...; Rn<4> b=...;
      // Let V< Rn<4> > x=Rn<4>::v(m,a,b);
      // then x.dim()==m and
      //  inf<R>(a[j],b[j])<=x[i][j]<=sup<R>(a[j],b[j]) for all
      // i \in {1,...,m} and j \in {1,...,n}.
      // The sequence forms an equi-spaced lattice for random=false
      // (not yet implemented) and a random cloud for random=true
      // (implemented).

   static  F<Z, Type > f(const Type& a, const Type& b, bool random=true);
      // same as above with x[i] repalced by f(i), so the length has not
      // to be given. Actually this is the basic algorithm and the
      // previous
      // function is implemented through the present one.
};

template <Z n>
R Rn<n>::vol(const Type& y)
```

```
{
   R ai,bi,res=1;
   for (Z i=1;i<=n;i++){
      ai=(*this)[i];
      bi=y[i];
      res*=(bi-ai);
   }
   return res;
}

template <Z n>
R1 Rn<n>::volFunc(const Type& x)
{
   R res=1;
   for (Z i=1;i<=n;i++) res*=x[i];
   return R1(res);
}

template <Z n>
Rn<n> Rn<n>::fFunc(const Z& i, const Type& a, const Type& b,
 const bool& random)
{
   Type res;
   if (random){
      Z shift = n==137 ? 127:137;
      const Z iMax=137731127;
      Z i0=i;
      for (Z j=1;j<=n;j++){
         res[j]=a[j]+CpmRoot::randomR(i0)*(b[j]-a[j]);
         i0+=shift;
         if (i0>iMax) i0=0;
      }
   }
   else{
      cpmerror("Rn<n>::fFunc(...random=false) not implemented");
   }
   return res;
}

template <Z n>
F<Z, Rn<n> > Rn<n>::f(const Type& a, const Type& b, bool random)
      // same as above with x[i] repalced by f(i), so th elength has not
      // to be given
{
   return F3<Z,Type,Type,bool,Type >(a,b,random)(Type::fFunc);
}

template <Z n>
V< Rn<n> > Rn<n>::v(Z m, const Type& a, const Type& b, bool random)
{
```

```
   F<Z, Type> ff=Type::f(a,b,random);
   V< Type > res(m);
   for (Z i=1;i<=m;i++) res[i]=ff(i);
   return res;
}
//////////////////////////// class Rn_Func<n> ////////////////////////////

template <Z n>
class Rn_Func: public Fa<Rn<n>,R1>{ // R1-valued functions on Rn<n>
   // R1-valued functions on Rn<n>  with arithmetics and trivially
   // defined order properties, and construction from discrete data via
   // interpolation.

   typedef Rn<n> Dom;
      // domain, where the function is defined
   typedef R1 Ran;
      // 'range' (co-domain), type to which the values of the function
      // belongs
   typedef Rn_Func<n> Type;
   typedef Fa<Dom,Ran> Base;
      // 'a' stands for arithmetic, defines addition and multiplication
      // of functions

   static V<Type> iniX();
      // initializes the array x of component functions
   static Ran parDer(const Dom& x, const Type& f, const Z& i,
      const R& h, const bool& lean);

   static Ran fullDer(const Dom& x, const Type& f, const R& h);

   static Ran linFunc(const Dom& x, const Dom& y)
   {
      R res=0;
      for (Z i=1;i<=n;i++) res+=x[i]*y[i];
      return Ran(res);
   }

   static Ran shiftFunc(const Dom& x, const Type& f, const Dom& s)
   {  return f(x-s);}

public:

// carrying over the constructors from Fa<R,R>

   Rn_Func (const F<Dom,Ran>& g):Base(g){}
      // most universal constructor. Cooperates with e.g.
      // F2<Dom,X1,X2,Ran>

   Rn_Func (const Base& g):Base(g){}
      // upcast constructor
```

```
Rn_Func (const Type & g):Base(g){}
   // copy constructor

Rn_Func (void):Base(){}
   // default constructor

explicit   Rn_Func ( Ran (*f)(const Dom &)):Base(f){};
   // constructor from native function pointer

explicit   Rn_Func ( Ran (*f)(Dom)):Base(f){};
   // constructor from native function pointer

explicit   Rn_Func (R r):Base(Ran(r)){};
  // constructor for constant function

explicit   Rn_Func (const Dom& a):Base(F1<Dom,Dom,Ran>(a)(linFunc)){}
  // constructor for linear form determined by a

Type shift(const Dom& s)const
   { return F2<Dom,Type,Dom,Ran>(*this,s)(shiftFunc);}
   // If f has a peak at xP, f.shift(s) has a peak at xP+s !
   // Thus we need a '-' (sic!) in the definition of shiftFunc.

Type der(Z i, R h=1e-3)const;
   // returns the partial derivative of function *this
   // with respect to component i

Type der(R h=1e-3)const;
   // f.der() is the complete mixed derivative
   // f.der(1).der(2).... .der(n)

Type derX(R h=1e-3)const;
   // returnes the complete mixed derivative
   // according to a second implementation based on
   // evl2. This consists of 2^n terms with 'alternating' signs
   // so that for n>4 one may get no reliable results for any h.
   // Before this bad situation occurs, one may find varying h
   // at least a platau-dependence on h over, say, a order of
   // magnitude; the stable plateau value may than be taken
   // as the value of the function. This remark also applies to
   // function der, but there the dependence on h seems to be
   // less critical.

Ran evl2(const Dom& x, const Dom& y )const;
   // the two argument evaluation
   // f.eval2(a,b) generalizes f(b)-f(a)
   // basic construct for my formulation of the 'fundamental
   // theorem of calculus' in n dimensions
```

```
   static R test1(Z i=1000);
      // should give very small values for all i
      // Volume of a rectangle can be expressed by eval2()
      // acting on the volume-generating function

   static R2 test2(R h=1e-3);
      // should give a pair of values which are close together
      // presently based only on one function as which 'waird'
      // is used since the random generator is not yet implemented
      // in a useful manner

   static V<Type> x;
      // list of component functions
      // x[i].der(j)=kronecker(i,j)

   static Type waird(Z i=1000);
      // For each i this is a
      // product of n random generated affine-linear functions.
      // Thus a rather general n-order polynomial in the
      // n component functions Type::x[1],...Type::x[n].
};


template <Z n>
R1 Rn_Func<n>::evl2(const Dom& x, const Dom& y )const
{
   Z i, j, p=Z1(2).pow(n);
   R si,sum=0;
   for (i=0;i<p;i++){
      V<bool> b=bitsFromZ(i,n);
      Z par=1;
      Dom z;
      for (j=1;j<=n;j++){
         bool bj=b[j];
         if (bj){
            par*=-1;
            z[j]=x[j];
         }
         else{
            z[j]=y[j];
         }
      }
      si=par*evl(z);
      sum+=si;
   }
   return R1(sum);
}

template <Z n>
R Rn_Func<n>::test1(Z i)
```

```
{
   R size=2;
   R_Vector x(n,size);
   R_Vector x1=x.ranVal(i);
   R_Vector x2=x.ranVal(i+1);
   Dom y1(x1);
   Dom y2(x2);
   Type vol(Dom::volFunc);
   R1 r1=vol.evl2(y1,y2);
   R1 r2=R1(y1.vol(y2));
   return r1.disVal(r2);
}

template <Z n>
R2 Rn_Func<n>::test2(R h)
{
   R size=2;
   R_Vector x(n,size);
   R_Vector x1=x.ranVal();
   Dom y1(x1);
   Type w=waird();
   Type f1=w.der(h);
   Type f2=w.derX(h);

   R r1=f1(y1);
   R r2=f2(y1);
   return R2(r1,r2);
}

template <Z n>
R1 Rn_Func<n>::parDer(const Dom& x, const Type& f, const Z& i,
   const R& h, const bool& lean)
{
   static Word loc("Rn_Func<n>::parDerparDer(...)");
   cpmassert(i>=1 && i<=n,loc);
   cpmassert(h!=0,loc);
   R h2=2*h;
   Dom x_=x;
   R y;
   if (lean){
      x_[i]+=h;
      y=f(x_);
      x_[i]-=h2;
      y-=f(x_);
      y/=h2;
   }
   else{
      x_[i]-=h2;
      y=f(x_); x_[i]+=h;
      y-=8*f(x_); x_[i]+=h2;
```

```
      y+=8*f(x_); x_[i]+=h;
      y-=f(x_);
      y/=(12*h);
   }
   return R1(y);
}

template <Z n>
R1 Rn_Func<n>::fullDer(const Dom& x, const Type& f, const R& h)
{
   static Word loc("Rn_Func<n>::fullDerparDer(...)");
   cpmassert(h!=0,loc);
   Dom y=x;
   R z=1;
   for (Z i=1;i<=n;i++){
      y[i]+=h;
      z*=h;
   }
   return f.evl2(x,y)/z;
}


template <Z n>
Rn_Func<n> Rn_Func<n>::waird(Z j)
{
   R size=2;
   Z i,ic=j;
   V<R_Vector> tv(n,R_Vector(n,size));
   for (i=1;i<=n;i++){
      tv[i]=tv[i].ranVal(ic);
      ic++;
   }
   V<R_Vector> sv(n,R_Vector(n,size));
   for (i=1;i<=n;i++){
      sv[i]=sv[i].ranVal(ic);
      ic++;
   }
   Type res(1.); // function constant 1
   for (i=1;i<=n;i++){
      Type fi=Type(tv[i]).shift(sv[i]);
         // shifted linear form
      res*=fi; // product from Fa<>, works !
   }
   return res;
}

template <Z n>
V<Rn_Func<n> > Rn_Func<n>::iniX()
{
   V<Type > res(n);
```

```
   for (Z i=1;i<=n;i++){
      res[i]=F1<Dom,Z,R1>(i)(Dom::c);
   }
   return res;
}

template <Z n>
V<Rn_Func<n> > Rn_Func<n>::x=Rn_Func<n>::iniX();

template <Z n>
Rn_Func<n> Rn_Func<n>::der(Z i, R h)const
{
   F<Dom,R1> f=F4<Dom,Type,Z,R,bool,R1>(*this,i,h,true)(Type::parDer);
   return Type(f);
}

template <Z n>
Rn_Func<n> Rn_Func<n>::derX(R h)const
{
   F<Dom,R1> f=F2<Dom,Type,R,R1>(*this,h)(Type::fullDer);
   return Type(f);
}

template <Z n>
Rn_Func<n> Rn_Func<n>::der(R h)const
{
   Type f=*this;
   for (Z i=1;i<=n;i++){
      f=f.der(i,h);
   }
   return f;
}

} // namespace

#endif
```

168 cpmroot.h

```
//? cpmroot.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_ROOT_H_
#define CPM_ROOT_H_

/*

Purpose: Root finding for a real-valued function of a real
   argument in 'modern' form, where the function is
   represented as a instance of F<R,R> which is a data member of
   class FindRoot.
   Notice that this includes functions which depend on an additional
   parameter which considerably widens the scope of the method.

*/

#include <cpmtypes.h>
#include <cpmf.h>

namespace CpmAlgorithms{

   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmFunctions::F;

/************* Typical usage of class FindRoot ***************************

FindRoot r(F<R,R>(cos),0,1);
   // sets up the problem to find a solution of cos(x)=0 for 0<=x<=1.

R myPi=2*r.getRoot();
   // should be PI

*/

//////////////////////////// class FindRoot ////////////////////////////////

class FindRoot{  // finds a root of a function

public:

   FindRoot(const F<R,R>& f, R x1, R x2, R eps=1e-9);
      // setting the function under consideration and the interval
      // in which the root is searched first (will be extended if
      // no root can be found there) . Also the accuracy of the
      // root can be set
```

```
   FindRoot(R x1=0, R x2=1, R eps=1e-9);
      // also default constructor, does not specify a function, the root
      // of which should be found (the previous constructor does). This
      // can be done by function setF()

   void setLimits(R x1, R x2);
      // sets the interval on which root searching is based
      // correct order is assured internally.

   void setAcc(R eps);
      // sets the relative accuracy of the root

   void setF(const F<R,R>& f);
      // setting the 'function' ff for which a root is to to be found.

   R getRoot();
      // finds and returns the root x0. Provisions against calling this
      // function for undefined ff are made.

   R operator()(void){ return getRoot();}

   R getRootCrit(Z& success);
      // computes and returns the root x0
      // sets success to 0 if no root was found .
      // Provisions against calling this function for undefined ff are
      // made.

   R getValue()const;
      // returns ff(x0);
      // Can meaningfully be called only after one of the
      // previous functions was called, so that a root finding process
      // was already performed.

private:

   R x0, y0, xL, xU, tol;
   F<R,R> ff;
      // 'function' the root of which has to be found
   void solve(void);
      // activates root finding
   void solveCrit(Z& success);
      // activates root finding
      // sets success to 0 if no root was found
   Z done;
   Z ffDefined;
   Z brac(R* x1, R* x2);
   R brent(R x1, R x2);
};
```

```
} // namespace

#endif
```

169 cpmroot.cpp

```
//? cpmroot.cpp
//? Status of work 2008-10-25.
//?
#include <cpmroot.h>

using CpmAlgorithms::FindRoot;
using CpmRoot::R;
using CpmRoot::Z;
using CpmFunctions::F;

FindRoot::FindRoot(const F<R,R>& f, R x1, R x2, R eps):ff(f)
{
   setLimits(x1,x2);
   setAcc(eps);
   ffDefined=1;
}

FindRoot::FindRoot(R x1, R x2, R eps)
{
   setLimits(x1,x2);
   setAcc(eps);
   ffDefined=0;
}

void FindRoot::setLimits(R x1, R x2)
{
   if (x1<x2){ xL=x1; xU=x2;}
   else { xL=x2; xU=x1;}
}

void FindRoot::setAcc(R eps){ tol=eps;}

void FindRoot::setF(const F<R,R>& f)
{
   ff=f;
   ffDefined=1;
}

R FindRoot::getRoot()
{
   if (ffDefined!=1) cpmerror("FindRoot::getRoot(): ffDefined!=1");
   solve();
   done=1;
   return x0;
}

R FindRoot::getRootCrit(Z& success)
```

```
{
   if (ffDefined!=1) cpmerror("FindRoot::getRootCrit(): ffDefined!=1");
   solveCrit(success);
   done=1;
   return x0;
}

R FindRoot::getValue()const
{
   if (done!=1) cpmerror("FindRoot::getValue(): done!=1");;
   return y0;
}

namespace{

R SIGN(R a, R b){ return  b >= 0.0 ? cpmabs(a) : -cpmabs(a) ;}

}

Z FindRoot::brac(R* x1, R* x2)
//modified from Press et al. Numerical Recipes in C
{
   const R FACTOR=1.6;
   const Z NTRY=50;
   Z j;
   R f1,f2;
   if (*x1 == *x2) cpmerror("FindRoot::brac(...): *x1 == *x2");
   f1=ff(*x1);
   f2=ff(*x2);
   for (j=1;j<=NTRY;j++) {
      if (f1*f2 < 0.0) return 1;
      if (cpmabs(f1) < cpmabs(f2))
         f1=ff(*x1 += FACTOR*(*x1-*x2));
      else
         f2=ff(*x2 += FACTOR*(*x2-*x1));
   }
   return 0;
}

R FindRoot::brent(R x1, R x2)
//modified from Press et al. Numerical Recipes in C
{
   const Z ITMAX=100;
   const R EPS=1.e-12; // also influences accuracy, 3.e-8 in original
   Z iter;
   R a=x1,b=x2,c=x2,d=0,e,min1,min2;
   R fa=ff(a),fb=ff(b),fc,p,q,r,s,tol1,xm;

   if((fa > 0. && fb > 0.) || (fa < 0. && fb < 0.))
      cpmerror(
```

```
    "R FindRoot::brent(R x1, R x2): x1 and x2 do not bracket a root");
fc=fb;
for (iter=1;iter<=ITMAX;iter++) {
   if ((fb > 0.0 && fc > 0.0) || (fb < 0.0 && fc < 0.0)) {
      c=a;
      fc=fa;
      e=d=b-a;
   }
   if (cpmabs(fc) < cpmabs(fb)) {
      a=b;
      b=c;
      c=a;
      fa=fb;
      fb=fc;
      fc=fa;
   }
   tol1=2.0*EPS*cpmabs(b)+0.5*tol;
   xm=0.5*(c-b);
   if (cpmabs(xm) <= tol1 || fb == 0.0) return b;
   if (cpmabs(e) >= tol1 && cpmabs(fa) > cpmabs(fb)) {
      s=fb/fa;
      if (a == c) {
         p=2.0*xm*s;
         q=1.0-s;
      } else {
         q=fa/fc;
         r=fb/fc;
         p=s*(2.0*xm*q*(q-r)-(b-a)*(r-1.0));
         q=(q-1.0)*(r-1.0)*(s-1.0);
      }
      if (p > 0.0) q = -q;
      p=cpmabs(p);
      min1=3.0*xm*q-cpmabs(tol1*q);
      min2=cpmabs(e*q);
      if (2.0*p < (min1 < min2 ? min1 : min2)) {
         e=d;
         d=p/q;
      } else {
         d=xm;
         e=d;
      }
   } else {
      d=xm;
      e=d;
   }
   a=b;
   fa=fb;
   if (cpmabs(d) > tol1)
      b += d;
   else
```

```
         b += SIGN(tol1,xm);
      fb=ff(b);
   }
   return 0.0;
}

void FindRoot::solve(void)
{
   Z i=brac(&xL,&xU);
   if (i==0) cpmerror("FindRoot::solve(): brac() failed");
   x0=brent(xL,xU);
   y0=ff(x0);
}

void FindRoot::solveCrit(Z& success)
{
   static Z warningCounter=0;
   const Z warningMax=10;
   success=brac(&xL,&xU);
   if (success){
      x0=brent(xL,xU);
      y0=ff(x0);
   }
   else{
      x0=0;
      y0=0;
      if (warningCounter<warningMax){
         cpmwarning("FindRoot: no root found, garbage returned");
         warningCounter++;
      }
      else if (warningCounter==warningMax){
         cpmwarning(
            "FindRoot: ... garbage returned, messages dicontinued");
         warningCounter++;
      }
      else{
         ;
      }
   }
}
```

170 cpmrvector.h

```
//? cpmrvector.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_RVECTOR_H_
#define CPM_RVECTOR_H_

/*
   Purpose: see class R_Vector

*/
#include <cpmvr.h>
#include <cpmperm.h>
#include <cpmlinalg.h>
#include <cpminterval.h>

#include <vector>

namespace CpmArrays{

   using std::vector;
   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::C;
   using CpmArrays::Perm;
   using CpmLinAlg::R2;
   using CpmLinAlg::R3;
   using CpmLinAlg::R4;
   using CpmGeo::Iv;

   typedef Vr<Z> I_Vector;

   class R_Vector{
// vectors with R-valued components, rich interface including arithmetics
      // arbitrary length, dynamic allocation

   Vr<R> rep;
      // we choose composition instead of inheritance, so the
      // interface of Vr<R> needs to be known only to the implementer
      // of this class, the user finds the interface of R_Vector
      // in the present file together with the macro definitions
      // in cpminterfaces.h

   R_Vector(const Vr<R>& x):rep(x){}

   static R addFunc(const R& r1, const R& r2, const R& t)
      { return r1+r2*t;}
```

```
public:
   typedef R_Vector Type;
   typedef R ScalarType;
   CPM_IO
   bool scanFromFreeFor(istream&);
      // FreeFor stands for free format
      // reads numbers from a stream as long as this can successfully
      // be done and changes *this into the R_Vector having all these
      // numbers as components. So, one doesn't need to know the
      // number of these numbers in advance
   CPM_ORDER
   CPM_TEST
   CPM_DESCRIPTORS
   CPM_CONJUGATION
   CPM_DOT_PRODUCT
   CPM_SUM_M
   CPM_PRODUCT_M
   CPM_SCALAR_M
   CPM_DIFFERENCE
   CPM_DIVISION
   Z dim()const{ return rep.dim();}
   Z b()const{ return rep.b();}
   Z e()const{ return rep.e();}
   R_Vector& add(const R_Vector& v, R t)
      // adds v*t to *this and returns *this.
   { rep.c().fg_(addFunc,v.rep.c(),t); return *this;}

   static R etaInv;
// constructors
   explicit R_Vector(Z n=0):rep(n,0.){}
   explicit R_Vector(const Vl<R>& v):rep(V<R>(v)){}
   explicit R_Vector(const V<R>& v):rep(v){}
   explicit R_Vector(const vector<R>& v):rep(V<R>(v)){}
   explicit R_Vector(const R2& v):rep(V<R>("",v[1],v[2])){}
   explicit R_Vector(C z):rep(V<R>("",z[1],z[2])){}
   explicit R_Vector(const R3& v):rep(V<R>("",v[1],v[2],v[3])){}
   explicit R_Vector(const R4& v):rep(V<R>("",v[1],v[2],v[3],v[4])){}

   R_Vector(Word ind, R t1):
      rep(V<R>(ind,t1)){}
   R_Vector(Word ind, R t1, R t2):
      rep(V<R>(ind,t1,t2)){}
   R_Vector(Word ind, R t1, R t2, R t3):
      rep(V<R>(ind,t1,t2,t3)){}
   R_Vector(Word ind, R t1, R t2, R t3, R t4):
      rep(V<R>(ind,t1,t2,t3,t4)){}
   R_Vector(Word ind, R t1, R t2, R t3, R t4, R t5):
      rep(V<R>(ind,t1,t2,t3,t4,t5)){}
   R_Vector(Word ind, R t1, R t2, R t3, R t4, R t5, R t6):
      rep(V<R>(ind,t1,t2,t3,t4,t5,t6)){}
```

```
   R_Vector(Word ind, R t1, R t2, R t3, R t4, R t5, R t6, R t7):
      rep(V<R>(ind,t1,t2,t3,t4,t5,t6,t7)){}
   R_Vector(Word ind, R t1, R t2, R t3, R t4, R t5, R t6, R t7, R t8):
      rep(V<R>(ind,t1,t2,t3,t4,t5,t6,t7,t8)){}
   R_Vector(Word ind, R t1, R t2, R t3, R t4, R t5, R t6, R t7, R t8,
      R t9):rep(V<R>(ind,t1,t2,t3,t4,t5,t6,t7,t8,t9)){}
   R_Vector(Word ind, R t1, R t2, R t3, R t4, R t5, R t6, R t7, R t8,
      R t9, R t10):rep(V<R>(ind,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10)){}

   R_Vector(Z n, R x):rep(n,x){}
      // constructor for a vector with n comonents equal to x

   R_Vector(R xStart, R incrementStep, R xEnd);
// Chain constructor:
// Creates a vector which holds an equidistant chain of real values
// starting with xStart, xStart+incrementStep, xStart+2*incrementStep,
// ...
// The end point is the point x of this chain which satisfies
//   xEnd <= x < xEnd+incrementStep
// For instance,  R_Vector x(0, 1./n, 1) gives a vector of
// dimension n+1 such that x[1]=0 and x[n+1]=1.

   R_Vector(Iv iv, Z n);
      // n equidistant components filling the interval iv (boundaries
      // included, (*this)[1]=iv.inf(), (*this)[n]=iv.sup() )
      // For n=1 (*this)[1]=(iv.inf()+iv.sup())/2
      // n=0 calls cpmassert

   R_Vector(R2 p, Z n);
      // List of n equidistant components starting with p[1] and ending
      // exactly with p[2]
      // For n=1 (*this)[1]=(p[1]+p[2])/2
      // n=0 calls cpmassert

// end of constructors
   operator V<R>()const{ return rep;}
      // the opposite conversion is not automatic !

   vector<R> std()const{ return rep.std();}
      // returns a std::vector with the same components
      // Let res be the return value, then clearly
      // res[0]=rep[1]

   const R& cyc(Z i)const{ return rep.c().cyc(--i);}
   R& cyc(Z i){ return rep.c().cyc(--i);}

   R& operator[](Z i){ return rep.c()[--i];}
   const R& operator[](Z i)const{ return rep.c()[--i];}

   R& fir(){ return rep.fir();}
```

```
   //: first
R const& fir()const{ return rep.fir(); }
   //: first
R& last(){ return rep.last(); }
   //: last
R const& last()const{ return rep.last(); }
   //: last

R const& read(Z i, Outside mode)const{ return rep.c().read(--i,mode);}
   // reading components in an efficient (as &'s), safe,
   // and flexible manner
   // mode 0: values outside assumed to be given as T()
   // mode 1: values outside assumed to be given by cyclic
   // continuation
   // else: values outside assumed to be given by constant
   // continuation

R in_(R const& t, bool reverse=false){ return rep.in_(t,reverse);}
   // This operations treats *this as a shift register:
   // All components get shifted by one position 'to the right' and
   // the total length (dim) of the vector remains the same. So what
   // was the last component prior to the operation has to be removed
   // from the vector, in order to not wasting information, this
   // removed component will shine up as the return value of the
   // function. After the operation, the first component of the vector
   // will be t.
   // If reverse==true, t gets inputted at the end, and all shift
   // operations go 'to the left'.

R_Vector eliFirst()const{ return R_Vector(rep.eliFirst());}

R_Vector eliLast()const{ return R_Vector(rep.eliLast());}

R_Vector eli(Z i, Z nEli=1)const{ return R_Vector(rep.eli(i,nEli));}

R_Vector cut(Z i)const{ return R_Vector(rep.cut(i));}
   //: cut
   // returned is a R_Vector which results from *this by removing i
   // components from the end

V<Z> permutationForIncreasingOrder()const
   {return rep.permutationForIncreasingOrder();}

R_Vector permute(const V<Z>& p)
   { return R_Vector(rep.permute(p));}

R_Vector operator*(const Perm& p)const
   { return R_Vector(rep.permute(p.getRep()));}

R_Vector increasingCopy()const
```

```
   { return R_Vector(rep.increasingCopy());}

R_Vector decreasingCopy()const
   { return R_Vector(rep.decreasingCopy());}

R_Vector select(const V<bool>& sel)const
   { return R_Vector(rep.select(sel));}

void order(){ rep.order();}

void sort_(){ rep.order();}

void set_(const R& t){ rep.set_(t);}
   // changes all components to the value t

bool isVal()const;
   // returns false iff for any of the components
   // x[i] we have CpmRoot::isVal(x[i])==false

R inf(Z i=0)const;
   // returns the 'ith-smallest' component. No ordering
   // for i=0. For i>0 it is the component indexed 1+i
   // of the increasingly ordered version of rep

R sup(Z i=0)const;
   // returns the 'ith-largest' largest component.
   // For i=0 fast operation, no ordering
   // For i>0 it is the component indexed rep.dim()-i
   // of the increasingly ordered version of rep

R supAbs()const;
   // returns sup{ |rep[i]|}

Z indSup()const{ return rep.indSup();}
   // returns the smallest i for which (*this)[i]=(*this).sup()

Z indInf()const{ return rep.indInf();}
   // returns the smallest i for which (*this)[i]=(*this).inf()

R sum()const{ return rep.sum();}
   // returns the sum of all components

R mean()const{Z n=rep.dim(); return n==0 ? 0 : rep.sum()/n;}
   // arithmetic mean value of all components, fast

R med()const // median value of all components
   { return infMedSup()[2];}

Iv infSup()const
{ pair<R,R> prr=rep.infSup(); return Iv(prr.first, prr.second);}
```

```
   // returns the smallest and the largest component. Fast operation,
   // no ordering

   R3 infMeanSup()const;
   // returns the smallest component, the arithmetic mean value,
   // and the largest component of *this. Fast operation, no ordering

   R3 infMedSup()const;
   // returns the smallest component, the median value,
   // and the largest component of *this. Achieved by ordering the
   // components. If one should also need the mean value, don't call
   // the previous function in addition, but simply the cheap function
   // mean()

   R_Vector pseudoInverse(R acc)const;
// Gives the pseudoinverse of the vector *this.
// This means that all components get inverted except those that are
// close to zero in comparison to the component with the largest
// absolute value.
// For small components c ( |c| ~ acc*sup |(*this)[i]| ) a continuous
// transition between the inverse and zero becomes operational.

    R_Vector scaleTo(const R& r)const;
// returns a vector of absolute value |r| and the direction of
// *this for positive r and antidirection for negative r.
// same function is available for complex n

   R_Vector resize(Z newDim)const
   { return R_Vector(rep.resize(newDim));}

   R_Vector subMean()const{ R s=-mean(); return R_Vector(rep+s);}
      //: subtract mean

   R subMean_(){ R m=mean(); rep-=m; return m;}
      //: changes *this by subtractng the mean from all components
      // and returns the mean

   R_Vector posCopy()const;
      // positive copy
      // returns a R_Vector obtained from *this by replacing all
      // components
      // by their absolute Value

   R_Vector log10()const;
      // log10
      // returns a R_Vector obtained from *this by replacing all
      // components
      // by the log10 of their absolute value. The safe version of log10
      // which is a member function of R1 is employed for this.
```

```
   R_Vector operator()(F<R,R> const& f)const{ return R_Vector(rep(f));}

   R_Vector cutCopy(Iv iv)const;//{ return *this;}

   R3 variation()const;
      // returns (a,b,c) such that a=Min{ x[i+1]-x[i] : i=1,...d-1}
      // c=Max{ x[i+1]-x[i] : i=1,...d-1}
      // b=Mean{ |x[i+1]-x[i]| : i=1,...d-1}.
      // Thus a<=c, b<=Max(|a|,|c|). If *this is monotone ascending,
      // a is never negative. If *this consists of empirical data
      // which in the absence of errors and uncertainties would
      // be monotone ascending, a slightly negative a may be OK
      // if |a|<<b. Implemented as a single pass operation
      // without ordering or copying.
      // Notice that the meanvalue of x[i+1]-x[i] (thus without the abs)
      // is simply (x[n]-x[1])/(n-1) and thus has not to be determined by
      // a loop over all components.
      // Notice also that b*(d-1) is the sum of all |x[i+1]-x[i]|
      // which comes close to a polygon length if the spacing in
      // i-direction is neglegiable compared to the swing in x-direction.
      //

   R_Vector der(R h=1, bool periodic=false) const;
// The vector a.der() results from the vector a by numerical
// differentiation of a function, the equi-distantly sampled version
// (sampling step = h) of which the vector a is considered to be.
// Formulas: Bronstein Semendiajew, Taschenbuch der Mathematik S. 768
// Tabelle 7.13 for generic points and Abramowitz Stegun p. 914
// Table 25.2 for the boundary points.
// If the second argument is true the function is periodic so that
// for i outside the allowed range we use this->cyc(i) as the component

   R_Vector& operator &=(const R_Vector& x1)
   { rep&=x1.rep; return *this;}
   // mutating form of concatenation

   R_Vector& operator &=(R r)
   { rep&=r; return *this;}

   R_Vector operator & (const R_Vector& x1)const
   { return R_Vector(rep&x1.rep);}
   // concatenation

   void operator<<(R const& r){ rep<<r;}
      // changes *this by appending a component, Ruby style

   void operator<<(R_Vector const& x1){ rep<<x1.rep;}
      // changes *this by appending a R_Vector, Ruby style

   R_Vector operator & (const R& r)const
```

```
   { return R_Vector(rep&r);}

   R_Vector prepend(const R& r)const
   { return R_Vector(rep.prepend(r));}

   void printLineOn(ostream& out , const Word& separator="  " )const
      // prints the components of *this, separated by the character
      // string separator. In default mode all numbers are in a line
      // which
      // is usefull for creating data that later have to be given to a
      // plot program such as gnuplot.
   { for (Z i=1;i<=rep.dim();i++) out<<separator<<rep[i];}

   R_Vector crsPrd(const R_Vector& x)const;
      // cross product (vector product) defined for
      //    dim()==2 && x.dim()>=2
      // and
      //    dim()==3 && x.dim()>=3
      // else error, since then presumably something went severely wrong

};

// inline reduction of most functions to those of rep in Vr<R>
// in order not to loose speed

inline bool R_Vector::prnOn(ostream& out)const
{ return rep.prnOn(out);}

inline bool R_Vector::scanFrom(istream& in)
{ return rep.scanFrom(in);}

inline Z R_Vector::com(const R_Vector& x)const
{ return rep.com(x.rep);}

inline R_Vector& R_Vector::operator +=(const R_Vector& x)
{ rep+=x.rep; return *this;}

inline R_Vector& R_Vector::operator *=(const R& r)
{ rep*=r; return *this;}

inline R_Vector& R_Vector::operator *=(const R_Vector& x)
{ rep*=x.rep; return *this;}

inline R_Vector R_Vector::neg()const
{ return R_Vector(-rep);}

inline R_Vector R_Vector::ranVal(Z j)const
{ return R_Vector(rep.ranVal(j));}

inline Z R_Vector::hashVal()const
```

```
{ return rep.hashVal();}

inline R_Vector R_Vector::testVal(Z j)const
{ return R_Vector(rep.testVal(j));}

inline Word R_Vector::nameOf()const
{ return Word("R_Vector");}

inline Word R_Vector::toWord()const
{ return rep.toWord();}

inline R_Vector R_Vector::net(Z i)const
{ return R_Vector(rep.net(i));}

inline R_Vector R_Vector::inv()const
{ return pseudoInverse(etaInv);}

inline R_Vector R_Vector::con()const
{ return *this;}

inline R R_Vector::operator |(const R_Vector& x )const
{ return rep|x.rep;}

} // namespace

#endif
```

171 cpmrvector.cpp

```
//? cpmrvector.cpp
//? Status of work 2008-10-25.
//?
#include <cpmrvector.h>
#include <cpmrfunc.h>

using namespace CpmRoot;
using namespace CpmSystem;
using namespace CpmArrays;

R R_Vector::etaInv=1e-12;

bool R_Vector::isVal()const
{
   Z n=dim();
   for (Z i=1;i<=n;i++){
      if (!CpmRoot::isVal(rep[i])) return false;
   }
   return true;
}

R_Vector::R_Vector(Iv iv, Z n):rep(n)
{
   Word loc("R_Vector(Iv iv, Z n)");
   cpmassert(n>0,loc);
   R x1=iv.inf();
   R xn=iv.sup();
   if (n==1){
      rep[1]=0.5*(x1+xn);
   }
   else if (n==2){
      rep[1]=x1;
      rep[2]=xn;
   }
   else if (n>2){
      rep[1]=x1;
      rep[n]=xn;
      R h=(xn-x1)/(n-1.);
      for (Z i=2;i<n;i++) rep[i]=rep[i-1]+h;
   }
}

R_Vector::R_Vector(R2 p, Z n):rep(n)
{
   Word loc("R_Vector(R2 p, Z n)");
   cpmassert(n>0,loc);
   R x1=p[1];
```

```
   R xn=p[2];
   if (n==1){
      rep[1]=0.5*(x1+xn);
   }
   else if (n==2){
      rep[1]=x1;
      rep[2]=xn;
   }
   else if (n>2){
      rep[1]=x1;
      rep[n]=xn;
      R h=(xn-x1)/(n-1.);
      for (Z i=2;i<n;i++) rep[i]=rep[i-1]+h;
   }
}

R3 R_Vector::infMeanSup()const
{
   Z n=rep.dim();
   if (n==0) return R3();
   else if (n==1){
      R r=rep[1];
      return R3(r,r,r);
   }
   else{
      R r=rep[1];
      R vMin=r,vMax=r,sum=r;
      for (Z i=2;i<=n;i++){
         r=rep[i];
         sum+=r;
         if (r>vMax) vMax=r;
         if (r<vMin) vMin=r;
      }
      R vMean=sum/n;
      return R3(vMin,vMean,vMax);
   }
}

R3 R_Vector::variation()const
{
   Z n=rep.dim();
   if (n==1){
      return R3();
   }
   else if (n==2){
      R diff=rep[2]-rep[1];
      return R3(diff,cpmabs(diff),diff);
   }
   else{
      Z i=2;
```

```
      R xim=rep[1];
      R xi=rep[i];
      R di=xi-xim;
      R a=di;
      R b=cpmabs(di);
      R c=di;
      for (i=3;i<=n;i++){
         xim=xi;
         xi=rep[i]; // only one call to [] needed!
         di=xi-xim;
         if (di<a) a=di;
         if (di>c) c=di;
         b+=cpmabs(di);
      }
      b/=(n-1); // there were n-1 terms in the sum
      return R3(a,b,c);
   }
}

R_Vector::R_Vector(R xStart, R incrementStep, R xEnd)
/*
Chain constructor
Creates a vector which holds an equidistant chain of real values
starting with xStart, xStart+incrementStep, xStart+2*incrementStep, ...
The end point is the point x of this chain which satisfies
   xEnd <= x < xEnd+incrementStep
   98-6-4: Now a wrong sign of incrementStep gets corrected and creates a
   warning only
*/
{
   static R tiny=1e-6;
      // 97-6-25 to achieve that x(0, 1./n, 1) gives a vector of
      // dimension n+1 as it should be
   if (incrementStep==0.){
      cpmerror("R_Vector::R_Vector(...): incrementStep=0 forbidden");
   }
   R nR=(xEnd-xStart)/incrementStep;
   if (nR<0.){
      incrementStep=-incrementStep;
      nR=-nR;
      cpmwarning(
         "R_Vector::R_Vector(): had to change the sign of incrementStep")\
         ;
   }
   nR+=tiny;
   Z n=1+(Z)(floor(nR));
   R_Vector res(n);
   R y=xStart;
   for (Z i=1;i<=n;i++){
      res[i]=y;
```

```
      y+=incrementStep;
   }
   *this=res;
}

R_Vector R_Vector::pseudoInverse(R acc) const
{
   using CpmRoot::absVal;
   using CpmRoot::inv;
   R_Vector d=*this;
   Z i, n=dim();
   acc=absVal<R>(acc);
   if (acc==0.){ // most common case, thus first and without ordering
   // overhead
      for (i=1;i<=n;i++) d[i]=inv<R>((*this)[i]);
      return d;
   }
   R_Vector absthis(n);
   for (i=1;i<=n;i++) absthis[i]=absVal<R>((*this)[i]);
   R value, maxValue=-1;
   for (i=1;i<=n;i++){ // finding the maximum of the absolute values of
   // the
         // components as fast as possible, thus without ordering
      value=absthis[i];
      if (value>maxValue) maxValue=value;
   }
   R eta=maxValue*acc;
   eta*=0.5;
   eta*=eta;
   R nom,xx;
   for (i=1;i<=n;i++){
      xx=(*this)[i];
      nom=xx*xx+eta;
      cpmassert(nom>0,"R_Vector::pseudoInverse(R acc)");
      d[i]=xx/nom;
   }
   return d;
}

R_Vector R_Vector::scaleTo(const R& r)const
      // returns a vector of absolute value |r| and the direction of
      // *this for positive r and antidirection for negative r.
      // same function is available for complex numbers
{
   R rOrig=absVal();
   R_Vector res=*this;
   if (rOrig!=0.){
      R fac=(r/rOrig);
      res*=fac;
   }
```

```
   return res;
}

R_Vector R_Vector::posCopy()const
{
   return this->operator()(F<R,R>(CpmFunctions::R_Func::abs));
}
/*
{
   R_Vector res=*this;
   Z i, n=dim();
   for (i=1;i<=n;++i){
      if (res.rep[i]<0) (res.rep[i])*=-1;
   }
   return res;
}
*/

R_Vector R_Vector::log10()const
{
   return this->operator()(F<R,R>(CpmFunctions::R_Func::log10));
}

R_Vector R_Vector::cutCopy(Iv iv)const
{
   R x1=iv[1];
   R x2=iv[2];
   if (x1<0) x1=0;
   if (x2>1) x2=1;
   Z i,n=dim();
   Z span=n-1;
   Z i1=cpmround(1+x1*span);
   Z i2=cpmround(1+x2*span);
   Z nRes=1+i2-i1;
   Z i1_=i1-1;
   R_Vector res(nRes);
   for (i=1;i<=nRes;++i){
      res.rep[i]=rep[i1_+i];
   }
   return res;
}

R3 R_Vector::infMedSup()const
{
   Z n=dim();
   R r=rep[1];
   if (n==1) return R3(r,r,r);
   else{
      V<Z> indNew=rep.permutationForIncreasingOrder();
      R v_min=rep[indNew[1]];
```

```
      R v_max=rep[indNew[n]];
      R v_median;
      Z n_half=n/2;
      if (n_half*2==n){ // then n is even
         v_median=0.5*(rep[indNew[n_half]]+rep[indNew[1+n_half]]);
      }
      else{
         v_median=rep[indNew[1+n_half]];
      }
      return R3(v_min,v_median,v_max);
   }
}

R R_Vector::sup(Z i)const
{
   if (i==0) return rep.sup();
   static Word loc("R_Vector::sup(Z i)");
   cpmassert(i>0,loc);
   Z n=dim();
   cpmassert(n>=1,loc);
   if (n==1) return rep[1];
   else{
      Z iRes=n-i;
      cpmassert(iRes>=1,loc);
      V<Z> indNew=rep.permutationForIncreasingOrder();
      return rep[indNew[iRes]];
   }
}

R R_Vector::inf(Z i)const
{
   if (i==0) return rep.inf();
   static Word loc("R_Vector::inf(Z i)");
   cpmassert(i>0,loc);
   Z n=dim();
   cpmassert(n>=1,loc);
   if (n==1) return rep[1];
   else{
      Z iRes=1+i;
      cpmassert(iRes<=n,loc);
      V<Z> indNew=rep.permutationForIncreasingOrder();
      return rep[indNew[iRes]];
   }
}

R R_Vector::supAbs()const
{
   R res=0;
   for (Z i=1;i<=dim();++i){
      R xi=cpmabs(rep[i]);
```

```
      if (xi>res) res=xi;
   }
   return res;
}

R_Vector R_Vector::der(R h, bool periodic) const
// The vector a.der() results from the vector a by numerical
// differentiation of a function, the equi-distantly sampled image
// (sampling step = h) of which the vector a is considered to be.
// Formulas: Bronstein Semendiajew, Taschenbuch der Mathematik S. 768
// Tabelle 7.13 for generic points and Abramowitz Stegun p. 914 Table
// 25.2 for the boundary points.
// Notice: D2x:=x[i+1]-x[i-1]/2
// D4x:=x[i+2]-x[i-2]/4, then the formula to be used can be written as
// D := (4/3)D2-(1/3)D4 which is OK since for D2==D4 we get D=D2.
{
   Z mL=4;
   static Word loc("R_Vector::der(R,bool)");
   cpmmessage(mL,loc&" started");
   Z i,n=dim();
   R_Vector res(n);
   if (n<=1) return res;
   cpmassert(h!=0,loc);
   R s;
   if (!periodic){
      if (n==2){
         s=1./h;
         res[1]=s*((*this)[2]-(*this)[1]);
         res[2]=res[1];
      }
      else{  // n>=3
         s=1./(12*h);
         R s1=1./(2*h);
         res[1]=s1*(-3*(*this)[1]+4*(*this)[2]-(*this)[3]);
         res[2]=s1*(-(*this)[1]+(*this)[3]);
         res[n-1]=s1*(-(*this)[n-2]+(*this)[n]);
         res[n]=s1*((*this)[n-2]-4*(*this)[n-1]+3*(*this)[n]);
         for (i=3;i<=n-2;i++){
            res[i]=s*((*this)[i-2]-8*(*this)[i-1]+
               8*(*this)[i+1]-(*this)[i+2]);
         }
      }
   }
   else{
      s=1./(12*h);
      for (i=1;i<=n;i++){
         R a1=this->cyc(i-2);
         R a2=this->cyc(i-1);
         R a3=this->cyc(i+1);
         R a4=this->cyc(i+2);
```

```
         res[i]=s*(a1-8*a2+8*a3-a4);
      }
   }
   return res;
}

R_Vector R_Vector::crsPrd(const R_Vector& x)const
{
   Z d1=dim();
   if (d1==2){
      Z d2=x.dim();
      if (d2<d1){
         cpmerror(
            "R_Vector::crsPrd(const R_Vector&): dimension mismatch"
         );
         return R_Vector(); // never reached; to avoid warning
      }
      else{
         R_Vector res(1);
         res[1]=rep[1]*(x.rep)[2]-rep[2]*(x.rep)[1];
         return res;
      }
   }
   if (d1==3){
      Z d2=x.dim();
      if (d2<d1){
         cpmerror(
            "R_Vector::crsPrd(const R_Vector&): dimension mismatch"
         );
         return R_Vector(); // never reached; to avoid warning
      }
      else{
         R_Vector res(3);
         res[1]=rep[2]*(x.rep)[3]-rep[3]*(x.rep)[2];
         res[2]=rep[3]*(x.rep)[1]-rep[1]*(x.rep)[3];
         res[3]=rep[1]*(x.rep)[2]-rep[2]*(x.rep)[1];
         return res;
      }
   }
   else{
      cpmerror(
         "R_Vector::crsPrd(const R_Vector&): dim differs from 2 and 3"
      );
      return R_Vector(); // never happens; to avoid warning
   }
}

bool R_Vector::scanFromFreeFor(istream& ifs)
{
   CpmRootX::R1 x;
```

```
   std::vector<R> res;
   while (x.scanFrom(ifs)){
      res.push_back((R)x);
   }
   *this=R_Vector(res);
   return true;
}
```

172 cpms.h

```
//? cpms.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_S_H_
#define CPM_S_H_
/*

   Description: Defines a template class S<T> of sets
       the elements of which are of type T.
      Intented as a basis for a mathematics style definition of
           associative lists.

*/

#include <cpmvo.h>
#include <cpmtypes.h>

//////////////////////////// class S<> ////////////////////////////////////

namespace CpmArrays{

   using CpmFunctions::F;
   using CpmRoot::Word;

template <class T>
class S{ // sets of homotypic elements
   // Represents sets, all the elements of which are of type T.
   // It is assumed that T<T and T>T is defined.

protected:

   void create(Z n)
   // tool for constructors
   { n_=n; Vo<T> xx(n_); Vo<Z> vv(n_); x_=xx; v_=vv;}

   void create(const S& s)
   // tool for constructors
   { n_=s.n_; x_=s.x_; v_=s.v_;}

explicit   S(Z n){ create(n);}
      // Not yet a set since elements not initialized
      // for internal use only !!!!

   void setValidity(Z b);
      // sets a constant value b into the v-list
   void setValidity(T const&  t, Z b);
      // sets the validity of the element which equals t to b
```

```
// data all well typed so no copy constructor and assignment operator
// will be defined correctly by the compiler

   Z n_;
      // number of elements (i.e. cardinality)

   Vo<T> x_;   // ordered list of elements

   Vo<Z> v_;
      // auxiliary list indicating validity (valid:1 unvalid: 0 )

public:

   typedef S<T> Type;

   CPM_IO

// constructors.

   S(void){ create(0);}
      // constructor for the void set (n=0 added 95-11-24)

   S(const Word& w, T const&  t){w;create(1);x_[1]=t;v_[1]=1;}
      // constructor for the set {t}
      // e.g. S<Z> a("",5); seems to be acceptable
      // note that explicit S(T const&  t){create(1);x[1]=t;v[1]=1;}
      // would be ambiguous for T==Z

   S(Type const&  s){create(s);}
      // copy constructor

   void merge(V<Type> const& setList);
      // changes *this into the union of *this and the sets in the list.

   explicit S(V<T> const& vec):n_(vec.dim()),x_(vec),v_(n_,1)
      { normalize();}
      // constructor from an array of T-values (V<T> sufficient, not
      // needing Vo<T>; T, however, has to support ordering (which is
      // clear if S<T> is under consideration). Clearly, multiple
      // appearing of components in vec does not prevent their
      // single appearence in the result (which, otherwise, would not be
      // a set)

    explicit S(Vl<T> const& vec):n_(vec.dim()),x_(vec),v_(n_,1)
      {normalize();}
      // constructor from an array of T-values
      // T has to support ordering (which is clear if S<T> is under
      // consideration). Clearly, multiple
      // appearing of components in vec does not prevent their
```

```
      // single appearence in the result (which, otherwise, would not be
      // a set)

   S(V<T> const& vec, V<bool> const& vecValid);
      // constructor from an array of T-values, together with a list of
      // validity values. vec[i] is considered valid iff
      // 1<=i<=vec.dim() && 1<=i<=vecValid.dim() && vecValid[i]==true
      // The valid values are put into a set.
      // Clearly, muliply appearing components will appear only
      // once in the result (which, otherwise, would not be a set)

// cardinality access
   Z card()const{ return n_;}
   Z dim()const{ return n_;} // for uniformity with other array types
   Z size()const{ return n_;} // for uniformity with STL code

   Z b()const{ return 1;}
      // begin , as in V<>

   Z e()const{ return n_;}
      // end , as in V<>

// set formation operations
   bool addAct(T const& t);
      // changes *this into the union of (*this) and {t}
      // and returns true if t was not already there
      // so that it had to be added 'actually'

   void add(T const& t){ addAct(t);}
      // changes *this into the union of (*this) and {t}
      // return value of addAct not used

   void add_(T const& t){ Type s(t); add(s);}
      // experimental

   void add(Type const& s);
      // changes *this into the union of (*this) and s

   void insert(T const&  t){ add(t);}
      // for uniformity with STL code

   void push_back(T const&  t){ add(t);}
      // for uniformity with STL code where the container is vector<>

   bool remove(T const&  t);
      // *this will be changed into *this\{t}
      // If this reduces the cardinality of *this, the return
      // value is 'true' and 'false' else

   Type operator |(Type const&  s)const;
```

```
   // forming the union of *this and s (corresponds to the
   // 'or' operation for indicator functions)

Type operator &(Type const&  s)const;
   // forming the section of *this and s (corresponds to the
   // 'and' operation for indicator functions)

Type operator -(Type const&  s)const;
   // forming *this\s i.e. the set of all those elements of
   // *this which don't belong to s (relative complement)

Type operator ^(Type const&  s)const{ return (*this-s)|(s-*this);}
   // forming the symmetric difference of s1 and s2
   // (corresponds to 'exclusive or' for indicator functions)

T const&  operator[](Z i)const{ return x_[i];}
   // Means of iterating over all elements of the set
   // The specific indexing is gouverned by the < order.
   // If i is 'out of range' i.e. not 1<=i<=card(), this is
   // considered an error which is handled by Vo<T> (finally
   // by Vl<T>).

T& ref(Z i){ return x_[i];}
   // dangerous operation since changing x[i] may invalidate the
   // internal representation of *this. Needed for implementing
   // M<X,Y>.
   // Till 02-1-7 I had instead:  T operator[](Z i){ return x[i];}
   // This prevented Y& M<X,Y>::operator[](const X&) from working
   // properly. See also function normalize().

T const&  first()const;
   // first element of *this according to the ordering on
   // which the construction of Type is based.
   // causes error if *this is void

T const&  last()const;
   // last element of *this according to the ordering on
   // which the construction of Type is based.
   // causes error if *this is void

void normalize(void);
   // shortens a list representation with multiple appearence of
   // a component such that all components are mutually different.
   // It also eliminates from the x-list all those elements x[i] for
   // which v[i]==0.
   // It thus achieves the status which for a well formed
   // list representation of a set is mandatory.
   // Not needed to be called normally (although this would not
   // cause problems). However if function ref() was called and
   // used to change elements it might be necessary to call
```

```
      // normalize to get an benign set again.

   T operator()(Z i)const{ return x_(i);}
      // Means of iterating over all elements of the set
      // The specific indexing is gouverned by the < order.
      // For i's 'out of range', T() is returned. This is not
      // considered an error.

// boolean operations

   bool isVoid(void)const{ return n_==0;}

   bool hasElm(T const& t)const;
      //: has element
      // returns true iff t is an element of *this

   bool ni(T const&  t)const{ return hasElm(t);}
      // 'in' reversed, from LATEX symbol \ni for the mirror image of
      // the epsilon-like 'is element'-symbol

   Z locate(T const&  t)const;
      // returns 0 if t is not an element of *this
      // and its index if it is an element of *this

   bool isSubSetOf(Type const&  s)const;
      // returns true iff *this is a subset of s

   bool isSuperSetOf(Type const&  s)const;
      // returns true iff s is a subset of *this

// selection by functions

   void select(Z (*f)(T const& ));
      // turns *this into the subset consisting of those elements
      // x[i] of *this for which f(x[i])!=0

   void select(F<T,bool> const& f);
      // as previous function, only different type of the selection
      // function

   void select(V<bool> const& sel);
      // eliminates from *this all components x[i] for which sel[i]
      // is defined and statisfies sel[i]==false.
      // (see corresonding (non-mutating) function of V<T>)

   void eliminate(Z i);
      // eliminates from *this just component x[i] for 1<=i<=card()
      // Else, nothing is done. Does the same as remove((*this)[i])
      // but much faster , due to known position of the element to
      // remove.
```

```
   Type subSet(F<T,bool> const& f)const;
      // turns *this into the subset consisting of those elements
      // x[i] of *this for which f(x[i])==true

// relations to other classes:

   Vo<T> toV(void)const{ return x_;}
      // getting the representing vector as a new entity. It may be
      // useful to manipulate this vector by, for instance,
      // the V<T>::condense() function and then form a new set via the
      // constructor from vectors

// Infrastructure functions as declaration macros  known from including
// <cpmvo.h>
   CPM_ORDER
      // since T has to allow ordering, we implement the order
      // operations. This allows to form S< Type >.

   virtual S<T>* clone()const{ return new S(*this);}

//   static bool effectivelyEqual(T const&  t1, const T& t2){ return
// t1==t2;}
      // if no additional infrastructure in T i´s available, we can't do
      // better could be made non-static and virtual or template argument
      // to allow equivalence classes to be formed during set formation;
      // not persued here

   Word toWord()const;

   virtual Word nameOf()const{
      Word wi="S< ";
      Word wt=CpmRoot::Name<T>()(T());
      return wi&wt&" >";
   }
};

// Implementation of S<T>

#define CPM_FOR_ALL for (i=1; i<=n_; i++)

template <class T>
void S<T>::setValidity(Z b)
{
   Z i;
   CPM_FOR_ALL v_[i]=b;
}

template <class T>
bool S<T>::addAct(T const&  t)
```

```
{
   if (n_==0){
      x_=Vo<T>(1,t);
      v_=Vo<Z>(1,1);
      n_=1;
      return true;
   }
   else if (n_==1){
      if (t==x_[1]){
         return false;
      }
      else{
         x_=x_.enqueue(t);
            // no call to locate() in this case
         v_=v_&true;
         n_++;
         return true;
      }
   }
   else{
      Z i=x_.locate(t);
      Z i1=i;
      Z i2=i1+1;
      if (i1>0 && i1<n_ && x_[i1]==t){
         return false; // t found in x
      }
      else if (i2<=n_ && x_[i2]==t){
         return false; // t found in x
      }
      else{ // no call to located needed
        x_=x_.ins(i2,t);
        v_=v_&true;
        n_++;
        return true;
      }
   }
}

template <class T>
void S<T>::add(const S<T>& s)
{
   Z m=s.card();
   if (m==0) return; // nothing to do
   if (n_==0){
      *this=s;
      return;
   }
   n_+=m;
   x_=x_&s.x_;
   v_=v_&s.v_;
```

```
   normalize(); // gives n the right value
}

template <class T>
void S<T>::normalize(void)
{
   Z mL=10;
   cpmmessage(mL,"S<T>::normalize(void) started");
   if (n_==0){
      cpmmessage(mL,"S<T>::normalize(void) done, first n_==0 exit");
      return; // nothing to do
   }

   Z i,j,nOld=n_;

// Eliminating the invalid entries

   n_=0;
   for (i=1;i<=nOld;i++){
      if (v_[i]==1) n_++;
   }
      // now n_ is the number of valid terms

   if (n_==0){
      Vo<T> xx;
      Vo<Z> vv;
      x_=xx;
      v_=vv;
      if (cpmverbose>=4)
         cpmmessage("S<T>::normalize(void) done, second n_==0 exit");
      return;
   }

   Vo<T> arr(n_);
   i=1;j=1;
   while(i<=n_ && j<=nOld ){
      while(j<=nOld && v_[j]==0){ j++; }
      if (j>nOld) break;
      arr[i++]=x_[j++];
   }

   Vo<Z> indx=arr.permutationForIncreasingOrder();

 // Eliminating  repetitions of terms

   Vo<Z> ind(n_);
   T t;
   i=0;j=0;
   while (i<n_ && j<n_){
      i++; j++;
```

```
      t=arr[indx[j]];
      // now we have to look whether j has to be increased.
      // while (j<n && effectivelyEqual(t,arr[indx[j+1]]) ){j++;}
      while (j<n_ && t==arr[indx[j+1]] ){j++;}
         // now j is the largest value for which arr[indx[j]]==t
         // since arr[indx[j+1]]!=t by the while condition
      ind[i]=j;
   }

   n_=i;
   Vo<T> xx(n_);
   Vo<Z> vv(n_);

   x_=xx;
   v_=vv;

   CPM_FOR_ALL{
      x_[i]=arr[indx[ind[i]]];
      v_[i]=true;
   }
   cpmmessage(mL,"S<T>::normalize(void) done");
}

template <class T>
S<T> S<T>::operator |(const S<T>& s)const
{
   Z nNew=n_+s.n_, i;
   S res(nNew);
   for (i=1;i<=n_;i++){
      (res.x_)[i]=x_[i];
      (res.v_)[i]=v_[i];
   }
   for (i=1;i<=s.n_;i++){
      (res.x_)[n_+i]=(s.x_)[i];
      (res.v_)[n_+i]=(s.v_)[i];
   }
   res.normalize();
   return res;
}

template <class T>
S<T> S<T>::operator &(const S<T>& s)const
{
   Z i;
   S<T> res;
   if (n_>s.n_){
      res=*this;
      res.setValidity(false);
      for (i=1;i<=s.n_;i++){
         res.setValidity((s.x_)[i],true);
```

```
      }
   }
   else{
      res=s;
      res.setValidity(false);
      for (i=1;i<=n_;i++){
         res.setValidity(x_[i],true);
      }
   }
   res.normalize();
   return res;
}

template <class T>
S<T> S<T>::operator -(const S<T>& s)const
{
   Z i;
   S<T> res=*this;
   for (i=1;i<=s.n_;i++){
      res.setValidity((s.x_)[i],false);
   }
   res.normalize();
   return res;
}


template <class T>
bool S<T>::hasElm(T const&  t)const
{
   Z r=x_.find(t);
   return (r!=0);
}

template <class T>
Z S<T>::locate(T const&  t)const
{
   return x_.find(t);
}

template <class T>
bool S<T>::remove(T const&  t)
{
   Z jf=x_.find(t);
   if (jf==0){
      return false;
   }
   else{
      v_[jf]=0;
      normalize();
      return true;
```

```
   }
}

template <class T>
void S<T>::setValidity(T const&  t, Z b)
{
   Z r=x_.find(t);
   if (r!=0) v_[r]=b;
}

template <class T>
bool S<T>::isSuperSetOf(const S<T>& s)const
{
   if (n_<s.n_) return false;
   S diff=s-(*this);
   return diff.isVoid();
}

template <class T>
bool S<T>::isSubSetOf(const S<T>& s)const
{
   if (n_>s.n_) return false;
   bool b=s.isSuperSetOf(*this);
   return !b;
}

template <class T>
void S<T>::select(Z (*f)(T const&))
{
   Z i;
   CPM_FOR_ALL v_[i]=f(x_[i]);
   normalize();
}

template <class T>
void S<T>::select(F<T,bool> const& f)
{
   Z i;
   CPM_FOR_ALL v_[i]=f(x_[i]);
   normalize();
}

template <class T >
void S<T>::select(const V<bool>& sel)
{
   Z i;
   Z ns=sel.dim();
   CPM_FOR_ALL v_[i]=(i<=ns ? sel[i] : true);
   normalize();
}
```

```
template <class T >
void S<T>::eliminate(Z i)
{
   if (i>1 && i<=n_){
      v_[i]=0;
      normalize();
   }
}

template <class T>
S<T> S<T>::subSet(F<T,bool> const& f)const
{
   S<T> res=*this;
   res.select(f);
   return res;
}

template <class T>
Z S<T>::com(S<T> const& s)const
{
   if (n_<s.n_){
      return 1;
   }
   else if (n_>s.n_){
      return -1;
   }
   else{ // now n==s.n
      for (Z i=1;i<=n_;i++){
         if ( x_[i]<(s.x_)[i]) return 1;
         if (x_[i]>(s.x_)[i] ) return -1;
      }
      return 0; // if all are equal
   }
}

template <class T>
S<T>::S(const V<T>& vec, const V<bool>& vecValid)
{
   n_=vec.dim();
   x_=vec;
   v_=Vo<Z>(n_);
   Z n2=CpmRootX::inf<Z>(n_,vecValid.dim());
   if (n2==0)
      cpmerror("S<T>::S(const V<T>&, const V<bool>&): dimension\
 mismatch");
   for (Z i=1;i<=n2;i++) v_[i]=(vecValid[i] ? 1 : 0);
   normalize();
}
```

```
template <class T>
Word S<T>::toWord()const
{
   Z nd=dim();
   Word res="{ ";
   for (Z i=1;i<nd;i++){
      Word wi=CpmRoot::toWord<T>((*this)[i]);
      wi&=", ";
      res&=wi;
   }
   res&=CpmRoot::toWord<T>((*this)[nd]);
   res&=" }";
   return res;
}

template <class T>
void S<T>::merge(const V< Type >& setList)
{
   Z d=setList.dim();
   for (Z i=1;i<=d;i++) add(setList[i]);
}

template <class T>
T const&  S<T>::first()const
{
   if (n_==0) cpmerror("S<T>::first(): set is void");
   return x_[1];
}

template <class T>
T const&  S<T>::last()const
{
   if (n_==0) cpmerror("S<T>::last(): set is void");
   return x_[n_];
}

template <class T>
bool S<T>::prnOn(ostream& str)const
{
   Z mL=2;
   static Word loc("S<>::prnOn(ostream&)");
   CPM_MA
   cpmwbt;
   CpmRoot::IO<Z>().o(n_,str);
   //cpmp(n_);
   Z i;
   CPM_FOR_ALL{
      if (CpmRoot::wrtTit){
         Word wi("// i=");
         wi&=cpm(i);
```

```
         bool bi=wi.prnOn(str);
            // the method here is to combine Words as Words and
            // write the result to stream (once!)
         cpmassert(bi==true,loc);
      }
      if (!CpmRoot::IO<T>().o((*this)[i],str) ){
         CPM_MZ
         return false;
      }
   }
   cpmwet;
   CPM_MZ
   return true;
}

template <class T>
bool S<T>::scanFrom(istream& str)
{
   Z mL=2;
   static Word loc("S<>::scanFrom(istream&)");
   CPM_MA
   if (!CpmRoot::IO<Z>().i(n_,str)){
      cpmwarning(loc&": invalid stream after reading cardinality");
      CPM_MZ
      return false;
   }
   else{
      if(n_>dimMax || n_<=0)
         cpmwarning(loc&" n_= "&cpm(n_));
      S<T> res;
      T t;
      Z i;
      CPM_FOR_ALL{
         if (!CpmRoot::IO<T>().i(t,str)){
            cpmwarning(loc&": reading failed for i="&cpm(i)&
               " of "&cpm(n_));
            CPM_MZ
            return false;
         }
         res.add(t);
      }
      *this=res;
      CPM_MZ
      return true;
   }
}

#undef CPM_FOR_ALL

} // CpmArrays
```

```
#endif
```

173 cpmset.h

```
//? cpmset.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_SET_H_
#define CPM_SET_H_
/*
   Description: Defines a class of finite sets, free of
      any further typing. Of course, it is implied that also
      the elements are finite sets, and so on. Such sets are called
      'hereditarily finite'.

   History: Started 02-11-13 after preliminary implementation in RUBY
   2005-07-16 a lot of functions added:
      V<Set> getValues(Set const& x)const;
      Set getFunctionValue(Set const& x)const;
      static Set makeNumericFunction(Z n, const F<Z,Z>& f);
      static Set makeCyclicGroup(Z n);
      Z numberOfBraces()const{ return val.size();}
      static Set makeNumber(Z n);
      bool isGroup()const;
      static bool test1(Z n);
      static bool test2(Z n);
      That test2 runs successfully for all arguments <=10
      (for n=10 the Set encoding of S(10) is by 43190 braces
      which translates into 43190 bits) creates confidence that the
      main functions (one needs all of the work horses in the group
      test) are correctly implemented.
      To do: Graphical representation as a mountain chain given by the
      brace level.
   2005-08-02 attribute isValid eliminated, attribute n changed to n_
   2005-11-03 adding to the comments to n_ and add
   2008-04-27 extending 'General considerations'
*/

#include <cpmnumbers.h>
#include <cpmword.h>
#include <cpmv.h>
#include <cpmf.h>
#include <vector>

//////////////////////////// class Set ////////////////////////////////////
/*
General considerations:
2005-08-02
The way how more complex mathemathical entities emerge from simpler ones
shows striking similarities with the way in which more complex biological
entities emerge from simpler ones. At least in the object oriented
```

```
approach to mathematical entities, each such entity has a characteristic
finite system of symbols that determines its 'state' and that can be
compared with the DNA molecule in the nucleus of a living cell or -
sometimes more appropriately - with a gene
(which is a subsection of such a DNA molecule).
Boolean values are the most simple 'genes' in mathematics. Finite lists
of those are the natural numbers in binary coding (see cpmnr.(h,cpp))
'Stone-age coding' by strings of a single character
(see cpmmathfound.(h,cpp)) is a very vasteful encoding of natural
numbers.
In order to build a hierarchy of mathematical notions which is easy to
use one should consider boolean values and natural numbers already
defined before considering sets. Having B and N (natural symbols for the
boolean type and the natural number type) it is very important to define
for arbitrary types T the type V<T> of finite sequences of quantities of
type T. If this all is given, we have a king's way to sets, which then
provide a universal framework for mathematics. Actually, already
N:={0,1,2,...} can be obtained as V<B> e.g. with the correspondence
0 <--> false, 1 <---> true and by the agreement that a V<B> string b
indexed as (b_p,b_{p-1},...,b_1,b_0) corresponds to the number
n = 2^{p} * b_p + 2^{p-1} * b_{p-1} + ... + 2^{1} * b_1 + 2^{0} * b_0.
Here, for each string b with more than one component (i.e. p>0)
we have b_p=1.
*/

namespace CpmArrays{

   using std::string;
   using std::vector;
   using CpmRoot::Z;
   using CpmRoot::Word;
   using CpmFunctions::F;

class Set{  // represents general finite sets
   // The public interface of the class uses the classes bool, Z,
   // V<bool>, V<Set>, and F<Z,Z>. In mathematics one would call a set
   // finite even if each of its finitely many elements would be infinite.
   // So {N,Z,Q,R,C,H} is a finite set. This is obviously not ment here.
   // Set represents what in Mathematics today is called
   // 'Grothendieck universe of hereditarily finite sets'.
   // Here we see, that the C++ notion of a class and instance is very
   // much parallel to the notion of class (as 'not necessarily being a
   // set') and element in Bernays-Neumann set theory. ???
   // C++ does probably not allow to define a class in which Set would
   // be an instance ? .... not yet clear.

   static char a;
      // character which represents '{' in a set
      // theoretical expression
```

```
   static char z;
      // character which represents '}' in a set
      // theoretical expression

// independent data
   string val;
      // representation of *this as a string of characters a and z
      // It is the complete reresentation, thus {} for the void set
// dependent data
   Z n_;
      // number of elements;
      // too dangerous to name this number simply n as I did till
      // 2005-08-01. Actually I had some dubious code using n as a local
      // variable and as a function argument
      // not independent, equals elm.size()

   vector<string> elm;
      // to get to elm one has to remove the outer
      // pair of braces to get to the list of elements. There are no
      // separators in the list and none are needed since proper pairing
      // of the braces uniqely determines the subunits. These subunits
      // than are exactly the val data members of the elements of *this.
      // The process which creates elm from val ensures that the
      // order in which these elements appear in elm agrees with the
      // lexicographic order of a,z-valued strings (a<z, of course).

// end of data

   typedef Set Type;

// data all well typed; therefore the copy constructor and assignment
// operator will be defined correctly by the compiler

   void ini();
      // initializes elm from val

   void iniFromElm();
      // initializes val from elm

   static void unwrap(string&);
      // taking appart the enclosing braces

   explicit Set(const string& str):val(str){ ini();}
      // Making a set from its val data member.
      // Use only if str is well formed. Thus private.

   explicit Set(const vector<string>& vs):elm(vs){ iniFromElm();}
      // Making a set from its elm data member.
      // Use only if vs is well formed. Thus private.
```

```
   string rep()const{ return val;}

public:

   CPM_ORDER_PLAIN
   virtual Set* clone()const{ return new Set(*this);}
   CPM_IO
   Word nameOf()const{ return "Set";}

// constructors.

   Set(void);
      // creates the void set

   explicit Set(V<Set> const& vs);
      // creates a set the elements of which are just the components of
      // vs (clearly, multiply appearing components get used only once,
      // otherwise we would not get a set)

   explicit Set(Z n);
      // The natural n-element set: void for n<=0
      // For n>=1 it has as elements the sets
      // Set::makeNumber(1),....,Set::makeNumber(n)

// cardinality access
   Z card()const{ return n_;}
   Z dim()const{ return n_;} // for uniformity with other array types
   Z size()const{ return n_;} // for uniformity with STL code

// element read access

   Set operator[](Z i)const;
      // Returns the i-th element if  i \in {1,...card()}
      // else error

   Set first()const{ return (*this)[1];}
      // returns the first element of *this

   Set last()const{ return (*this)[n_];}
      // returns the last element of *this

   Set operator()(Z i)const;
      // continuation as constant below and above the limits
      // Thus no error for any value of i

   V<Set> operator()(void)const;
      // returns an array of all elements in natural order

// read access to internals
```

```
   string getVal()const{return val;}
   string getElm(Z i)const{ return elm[i-1];}

// set formation operations

   Set push()const;
   // push is the basic creative method. It returns {uv...w}
   // for *this=uv...w
   // note that the returned set has just one element

   V<Set> pop()const;
    // Let the result be denoted res. If *this has just one element,
    // res.dim()==1 and res[1] is just this element. Otherwise
    // res.dim()==0.
    // For all sets s thus (s.push().pop()).dim()==1 and
    // (s.push().pop())[1]==1

   Set operator |(Set const& s)const;
      // forming the union of *this and s (corresponds to the
      // 'or' operation for indicator functions)

   static Set uni(V<Set> const& vs);
      // union (unfortunately 'union' is a key-word in C++ so it can't be
      // used as a name) of the sets in the family vs

   static Set sec(V<Set> const& vs);
      // section of the sets in the family vs

   Set unionOfElements()const{ return uni(operator()());}
      // since the elements of *this are sets, we are allowed to
      // form the set union of them all. This set then is returned.

   Set sectionOfElements()const{ return sec(operator()());}
      // since the elements of *this are sets, we are allowed to
      // form the set section of them all. This set then is returned.

   Set subSet(const V<bool>& vb)const;
      // Let Set s;
      // s[i] will be an element of s.subSet(vb) iff vb[i]==true

   Set powerSet()const;
      // set of all subsets of *this

   Set add(Set const& s){ return (*this)|(s.push());}
      // adds s as an element (if not yet there)
      // The function f: Set x Set --> Set, (x,y) |--> x.add(y) plays
      // a decisive role in genertating Set:
      // Assume Set s=....
      // Then with
      // V<Set> vs=s();
```

```
      // Z n=vs.dim();
      // we have s=f(...f(f(vs[1],vs[2]),vs[3])...,vs[n])
      // Notice also that {s}=f({},s). So it is quite clear that starting
      // from {} as an element of Set and forming f(x,y) for yet formed
      // sets x,y one obtains any set. Thus in Backus Naur Notation:
      // <Set> ::= {}|f(<Set>,<Set>)

   Set operator &(Set const& s)const;
      // forming the section of *this and s (corresponds to the
      // 'and' operation for indicator functions)

   Set operator -(Set const& s)const;
      // forming *this\s i.e. the set of all those elements of
      // *this which don't belong to s (relative complement)

   Set operator ^(Set const& s)const;
      // forming the symmetric difference of s1 and s2
      // (corresponds to 'exclusive or' for indicator functions)

// boolean operations

   bool isVoid(void)const{ return n_==0;}
      // returns true if *this is void

   bool hasElem(Set const& s)const;
      // returns true iff s is an element of *this

   bool ni(Set const& s)const{ return hasElem(s);}
      // returns true iff s is an element of *this (same as previous
      // function).
      // from LATEX symbol \ni for the mirror image of the epsilon-like
      // 'is element'-symbol

   bool in(Set const& s)const{ return s.hasElem(*this);}
      // returns true iff *this is an element of s
      // from LATEX symbol \in for the epsilon-like 'is element'-symbol

   Z locate(Set const& s)const;
      // returns 0 if s is not an element of *this
      // and its index if it is an element of *this

   bool isSubSetOf(Set const& s)const;
      // returns true iff *this is a subset of s

   bool isSuperSetOf(Set const& s)const;
      // returns true iff s is a subset of *this

   Set wkp(Set const& y)const;
   // generates the Wiener Kuratowski pair
   // {{*this},{*this,y}}=:WKP(*this,y)
```

```
   // notice WKP(x,y):={{x},{x,y}} ==> WKP(x,x)={{x}}
   // see e.g. Paul Taylor: Practical Foundations of Mathematics,
   // Cambridge 1999, p. 121

   Set orderedPair(Set const& y)const{ return wkp(y);}
      // more explicit name for previous function

   Set operator *(Set const& y)const;
      // Cartesian product *this times y
      // x*y={WKP(a,b) | a \in x and b \in b}

   V<Set> getTwoFactors()const;
      // Let res be the return value.
      // Then res.dim()==0 if *this is not a WKP.
      // Else *this=WKP(res[1],res[2])
      // Consider Set p=s1.wkp(s2).wkp(s3).wkp(s4);
      // V<Set> v=p.twoFactors(). Then
      // v[1]=s1.wkp(s2).wkp(s3), v[2]=s4
      // The factorization is unique and splits off always the last
      // factor of a multiple product; we would never get factors like
      // s1.wkp(s2), s3.wkp(s4)

   static Set tupel(V<Set> const& vs);
      // creates a n-tupel (n=vs.dim()) by iterating the WKP
      // construction:
      // Lets write [x,y]:=WKP(x,y)
      // tupel(x,y,z):=[[x,y],z]
      // tupel(x,y,z,z'):=[tupel(x,y,z),z'].
      // This is a well defined procedure in which no definition problems
      // can arrise.

   static Set car(V<Set> const& vs);
      // cartesian product of all these sets
      // vs[1]*vs[2]*...*vs[n] consists of all
      // tupel[e1,...en] with e1 \in vs[1],...en \in vs[n]
      // Further: Set::car(vs)=(...((vs[1]*vs[2])*vs[3])...*vs[n])

   V<Set> getFactors()const;
      // decomposing a tupel into its composing components (factors since
      // tupel formation is analogous to multiplication in some way)
      // tupel(V<Set> const& vs).getFactors()==vs
      // For the result res we have res.dim()==0 if *this is not a tuple

// topic: relations and their algebra

   bool isRelation()const;
      // returns true iff all elements of *this are wkp's

   V<Set> decompose()const;
      // returns a list of subsets
```

```
      // where res[i] is the subset of *this of those elements which
      // are i-tuples

   V<Set> extractRelation()const;
      // If *this contains wkps as elements, the subset of those
      // will be returned as res[1]; res[2] will be the domain of
      // relation res[1], and res[3] will be the range of it.
      // If no element of *this is a wkp, res will be returned as
      // V<Set>(0). Notice that some of the elements of *this
      // which end up in res[1] may be decomposable further, but this
      // is not the task of the present function.

   bool isFunction()const;
      // returns true iff all elements of *this are wkp's
      // and if there apear no two different wkp's which agree
      // in their first component

   V<Set> getValues(Set const& x)const;
      // returns a list of y's such that (x,y) is an element of *this.
      // If no such y exists (e.g. if *this has no ordered pair as
      // element at all) then the returned list has length 0.
      // Notice that returning the void set would not be safe in this
      // case since the void set could be a legal return value of a
      // function.

   Set getFunctionValue(Set const& x)const;
      // returs the uniquely defined function value if it exists
      // and fails if it does not exist

 //  static V<Set> makeFunction(V<Set> const& rel);
      // let rel be a result from extract Relation.
      // If rel.dim()==0 we return V<Set>(0). Else we
      // return the mapping from res[2].powerSet() to res[3].powerSet()
      // which is induced by relation rel[1].

   static Set makeNumericFunction(Z n, const F<Z,Z>& f);
      // returns the the set
      // {(1,f(1)),...,(n,f(n)}

   Word toWord()const{ return Word(val);}

   Z numberOfBraces()const{ return (Z)val.size();}

// coding natural numbers (zero included) as sets

   static Set makeNumber(Z n);
      // Set::makeNumber(0)={}, Set::makeNumber(1)={{}},
      // Set::makeNumber(2)={{{}}}, ...
      // If n is negative, we set makeNumber(n):=makeNumber(-n) to define
      // the function for all values of the argument
```

```
   Z number()const;
      // returns the representation of *this as a standard number if
      // *this is of the form {{{{{... }}}}}....; else it returns -1.
      // Notice that Z allows to hold 'exceptional values' since in this
      // case only values >=0 can appear as regular values. So we can
      // deviate from the establishe method which in this case would be
      // to return a V<Z> of dim 0 in the irregular case and of dim 1 in
      // the regular one which then would hold the regular value as the
      // first component. We will have to return to this method for
      // coding boolean values since there are only two such values
      // which leaves no room for coding 'nil' as a boolean value

   bool isNumber()const{ return number()>=0;}

// coding boolean values as sets

   static Set makeBoolean(bool b)
      { return (b ? makeNumber(1) : makeNumber(0));}

   V<bool> boolean()const
   {
      if (*this==makeNumber(0)) return V<bool>(1,false);
      else if (*this==makeNumber(1)) return V<bool>(1,true);
      else return V<bool>(0);
   }

   bool isBoolean()const
   { return !boolean().isVoid();}

// coding groups as sets

   static Set makeCyclicGroup(Z n);
      // returns the cyclic group of order n, coded as the set
      // ({1,...n},product,inversion,unitelement)
      // Tests whether the functions provided so far are expressive
      // enough for defining this conveniently. The answer is yes!

   bool isGroup()const;
      // returns true if *this is a group, else false

// test functions

   static bool test1(Z n);
      // returns true if the test was done and everything turned out to
      // be correct, otherwise it returns wrong

   static bool test2(Z n);
      // concerned with making a group and testing whether it is one
```

```
   static bool test3(Z n);
      // concerned with making a number and testing whether it is one
};

} // namespace

#endif
```

174 cpmset.cpp

```
//? cpmset.cpp
//? Status of work 2008-10-25.
//?
#include <cpmset.h>
#include <cpmalgorithms.h>
#include <algorithm>

using namespace CpmArrays;
using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmSystem;

using namespace std;


char Set::a='{';
char Set::z='}';

void Set::unwrap(string& str)
{
   Z ns=(Z)str.length();
   if (ns<2) cpmerror("string too short for unwrapping");
   else if (ns==2) str="";
   else{
      str.erase(0,1);
      str.erase(str.length()-2,1);
   }
}

void Set::ini()
{
   Z mL=3;
   Word loc("Set::ini()");
   cpmmessage(mL,loc&" started");
   elm=vector<string>(0);
   n_=0;
   if (val.length()==2){
     // isValid=true;
      cpmmessage(mL,loc&" done, void set created");
      return;
   }
   else{
      string str=val;
      unwrap(str);
         // taking appart the enclosing braces
      Z sum=0;
      while (true){
```

```
         if (str.length()==0) break;
         string ele("");
         while (true){
            char v=str[0];
            str.erase(0,1);
            if (v==a){
               sum++;
               ele+=a;
            }
            else if (v==z){
               sum--;
               ele+=z;
               if (sum==0) break;
            }
            else{
               cpmerror(loc&": unexpected character found");
            }
         }
         elm.push_back(ele);
         n_++;
         cpmmessage(mL+1,"reading element "&cpmwrite(n_)&" done");
      }
   }
   cpmmessage(mL,loc&" done");
}

void Set::iniFromElm()
{
   Z mL=3;
   Word loc("Set::iniFromElm()");
   cpmmessage(mL,loc&" started");
   n_=(Z)elm.size();
   val=string(1,a);
   for (Z i=0;i<n_;i++) val+=elm[i];
   val+=string(1,z);
   cpmmessage(mL,loc&" done");
}

Set::Set()
{
   val=string(1,a)+string(1,z);
   ini();
}

Set::Set(const V<Set>& vs):elm(vs.dim())
{
   Z mL=3;
   Word loc("Set(const V<Set>&)");
   cpmmessage(mL,loc&" started");
   n_=(Z)elm.size();
```

```
   if (n_==0){
      *this=Set();
      cpmmessage(mL,loc&" done, void set made");

   }
   else if (n_==1){
      cpmmessage(mL,loc&" done, one-element set made");
      *this=vs[1].push();
   }
   else{
      for (Z i=0;i<n_;i++){
         string si=vs[i+1].rep(); // nothing to unwrap
         // the representations of the vs[i] have directly be
         // concatenated
         // to form elm of the set to generate
         elm[i]=si;
      }
      std::stable_sort(elm.begin(),elm.end());
      vector<string>::iterator newend;
      newend=std::unique(elm.begin(),elm.end());
         // uniqe works only as needed on ordered arrays
      elm.erase(newend,elm.end()); // not done automatically by unique!
      iniFromElm();
   }
   cpmmessage(mL,loc&" done");
}

Set::Set(Z nI)
{
   if (nI<=0) *this=Set();
   else if (nI==1){
      Set x;
      x=x.push();
      *this=x;
   }
   else{
      V<Set> vres(nI);
      Z i;
      for (i=2;i<=nI;i++){
         vres[i]=vres[i-1].push();
      }
      *this=Set(vres);
   }
}

Set Set::push()const
{
   string valRes=string(1,a)+val+string(1,z);
   return Set(valRes);
}
```

```
V<Set> Set::pop()const
    // s.push().pop() leaves s unchanged. If applied to to a set that has
    // not just one
    // element, *this will be left unchanged with respect to val and elm
    // but with isValid==false
{
   if (card()!=1){
      cpmwarning("Set::pop() created unvalid object");
      return V<Set>(0);
   }
   else{
      string valRes=elm[0];
      return V<Set>(1,Set(valRes));
   }
}

// order operators dirctly from the string representation

bool Set::operator == ( const Set & x)const
{ return val==x.val;}

bool Set::operator != (const Set & x)const
{ return val!=x.val;}

bool Set::operator < (const Set & x)const
{ return val<x.val;}

bool Set::operator > (const Set & x)const
{ return val>x.val;}

bool Set::operator <= (const Set & x)const
{ return val<=x.val;}

bool Set::operator >= (const Set & x)const
{ return val>=x.val;}


// lets use Word::prnOn and Word::scanFrom since than
// my mechanism to support the message passing interface is
// active
bool Set::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("Set",str);
   if (!Word(val).prnOn(str)) return false;
   return true;
}

bool Set::scanFrom(istream& str)
{
```

```
   Word w;
   if (!w.scanFrom(str)) return false;
   val=w.str();
   ini();
   return true;
}

V<Set> Set::operator()(void)const
{
   V<Set> res(n_);
   for (Z i=1;i<=n_;i++){
      res[i]=Set(elm[i-1]);
   }
   return res;
}

Set Set::operator |(const Set& s)const
      // forming the union of *this and s (corresponds to the
      // 'or' operation for indicator functions)
{
   Z ns=s.card();
   vector<string> elms=s.elm;
   vector<string> res(n_+ns);

   vector<string>::iterator limit;
   limit=std::set_union
      (elm.begin(),elm.end(),elms.begin(),elms.end(),res.begin());
   res.erase(limit,res.end());
   return Set(res);
}

Set Set::operator &(const Set& s)const
      // forming the section of *this and s (corresponds to the
      // 'and' operation for indicator functions)
{
   Z ns=s.card();
   vector<string> elms=s.elm;
   Z nRes=(n_<=ns ? n_ : ns);
   vector<string> res(nRes);

   vector<string>::iterator limit;
   limit=std::set_intersection
      (elm.begin(),elm.end(),elms.begin(),elms.end(),res.begin());
   res.erase(limit,res.end());
   return Set(res);
}

Set Set::operator -(const Set& s)const
{
   Z ns=s.card();
```

```
   vector<string> elms=s.elm;
   Z nRes=n_;
   vector<string> res(nRes);
   vector<string>::iterator limit;
   limit=std::set_difference
      (elm.begin(),elm.end(),elms.begin(),elms.end(),res.begin());
   res.erase(limit,res.end());
   return Set(res);
}

Set Set::operator ^(const Set& s)const
{
   Z ns=s.card();
   vector<string> elms=s.elm;
   Z nRes=n_+ns;
   vector<string> res(nRes);
   vector<string>::iterator limit;
   limit=std::set_symmetric_difference
      (elm.begin(),elm.end(),elms.begin(),elms.end(),res.begin());
   res.erase(limit,res.end());
   return Set(res);
}

Set Set::operator[](Z i)const
{
   if (i<1) cpmerror("Set::operator[](Z i): i="&cpmwrite(i));
   if (i>n_) cpmerror
      ("Set::operator[](Z i): i="&cpmwrite(i)&", n="&cpmwrite(n_));
   return Set(elm[i-1]);
}

Set Set::operator()(Z i)const
{
   if (i<1) i=1;
   if (i>n_) i=n_;
   return Set(elm[i-1]);
}

bool Set::hasElem(const Set& s)const
{
   string ss=s.rep();
   for (Z i=0;i<n_;i++){
      if (ss==elm[i]) return true;
   }
   return false;
}

Z Set::locate(const Set& s)const
      // returns 0 if s is not an element of *this
      // and its index if it is an element of *this
```

```
{
   string ss=s.rep();
   for (Z i=0;i<n_;i++){
      if (ss==elm[i]) return i+1;
   }
   return 0;
}

bool Set::isSubSetOf(const Set& s)const
      // returns true iff *this is a subset of s
{
   return n_==((*this)&s).card();
}

bool Set::isSuperSetOf(const Set& s)const
      // returns true iff s is a subset of *this
{
   return n_==(s&(*this)).card();
}

//Set Set::orderedPair(const Set& y)const
Set Set::wkp(const Set& y)const
{
   Set c1=push();
   V<Set> vc2("",*this,y);
   Set c2(vc2);
   V<Set> vRes("",c1,c2);
   return Set(vRes);
}

V<Set> Set::getTwoFactors()const
{
   Z mL=3;
   Word loc("Set::getTwoFactors()");
   cpmmessage(mL,loc&" started");
   if (n_==1){
      V<Set> x=pop();
      if (x.dim()==1){
         Set x1=x[1];
         return V<Set>("",x1,x1);
      }
      else return V<Set>(0);
   }
   else if (n_==2){
      Set c1=first();
      Set c2=last();
      Z n1=c1.card();
      Z n2=c2.card();
      Z n3=n1+n2;
      if (n3!=3){
```

```
         cpmwarning(loc&" no valid result due to n3!=3");
         return V<Set>(0);
      }
      else{
         Set x=c1&c2;
         Set y;
         if (n1==1) y=c2-x; else y=c1-x;
         return V<Set>("",x.pop()[1],y.pop()[1]);
      }
   }
   else{
      return V<Set>(0);
   }
   cpmmessage(mL,loc&" done");
}

Set Set::operator *(const Set& y)const
{
   Z mL=2;
   Word loc("Set::operator *(const Set& y)");
   cpmmessage(mL,loc&" started");
   Z ny=y.card();
   Z nRes=n_*ny;
   Z i,j;
   V<Set> vs=operator()();
   V<Set> vsy=y();
   V<Set> vRes(nRes);
   Z k=0;
   for (i=1;i<=n_;i++){
      for (j=1;j<=ny;j++){
         k++;
         vRes[k]=vs[i].wkp(vsy[j]);
      }
   }
   cpmmessage(mL,loc&
      " done, except of statement return Set(vRes), vRes.dim()="
      &cpmwrite(nRes));
   return Set(vRes);
}

Set Set::tupel(const V<Set>& vs)
{
   Z m=vs.dim();
   if (m==0) return Set();
   else if (m==1) return vs[1];
   else{
      Set res(vs[1]);
      for (Z i=2;i<=m;i++){
         res=res.wkp(vs[i]);
      }
```

```
      return res;
   }
}

V<Set> Set::getFactors()const
{
   Z mL=1;
   Word loc("Set::getFactors()");
   cpmmessage(mL,loc&" started");
   vector<Set> temp; // for fast appending at the end
   Set toFactorize=*this;
   bool go=true;
   Z nFactorsGot=0;
   while(go){
      V<Set> factors=toFactorize.getTwoFactors();
      if (factors.dim() !=2){
         go=false;
      }
      else{
         temp.push_back(factors[2]);
         toFactorize=factors[1];
         nFactorsGot++;
      }
   }
   if (nFactorsGot==0){
      cpmmessage(mL,loc&" done, trivial array returned");
      return V<Set>("",toFactorize); // if no split into two factor was
      // found,
         // the term is one factor himself
   }
   else{
      temp.push_back(toFactorize);
      Z i,m=(Z)temp.size();
      V<Set> res(m);
      for (i=1;i<=m;i++) res[i]=temp[m-i];
      cpmmessage(mL,loc&
         " done, array of dimension "&cpmwrite(m)&" returned");
      return res;
   }
}

Set Set::uni(const V<Set>& vs)
{
   Z i,m=vs.dim();
   if (m==0) return Set();
   else if (m==1) return vs[1];
   else{
      Set res=vs[1];
      for (i=1;i<=m;i++) res=res|vs[i];
      return res;
```

```
   }
}

Set Set::sec(const V<Set>& vs)
{
   Z i,m=vs.dim();
   if (m==0) return Set();
   else if (m==1) return vs[1];
   else{
      Set res=vs[1];
      for (i=1;i<=m;i++) res=res&vs[i];
      return res;
   }
}

Set Set::car(const V<Set>& vs)
{
   Z i,m=vs.dim();
   if (m==0) return Set();
   else if (m==1) return vs[1];
   else{
      Set res=vs[1];
      for (i=1;i<=m;i++) res=res*vs[i];
      return res;
   }
}

Set Set::subSet(const V<bool>& vb)const
{
   if (n_==0) return Set();
   else{
      V<Set> vs=operator()();
         // vs.dim()==n per def
      Z i,m=vb.dim();
      m=(m<=n_ ? m : n_); // inf(m,n)
      vector<Set> temp;
      for (i=1;i<=m;i++){
         if (vb[i]==true) temp.push_back(vs[i]);
            // vb[i] and vs[i] defined by definition of m
      }
      m=(Z)temp.size();
      if (m==0){
         return Set();
      }
      else{
         V<Set> vres(m);
         for (i=1;i<=m;i++) vres[i]=temp[i-1];
         return Set(vres);
      }
   }
```

```
}

Set Set::powerSet()const
      // set of all subsets of *this
{
   Z mL=2;
   Word loc("Set::powerSet()");
   cpmmessage(mL,loc&" started: n="&cpmwrite(n_));
   Z nBytes=sizeof(mL);
   Z nMax=8*nBytes;
   cpmassert(n_<=nMax,loc);
   Z m=Z1(2).pow(n_);
   V<Set> vres(m);
   for (Z i=0;i<m;i++){
      V<bool> vb=CpmAlgorithms::bitsFromZ(i,n_);
      vres[i+1]=subSet(vb);
   }
   cpmmessage(mL,
      "last possible message from "&loc&": collecting subsets done,"
      &" output may take time");
   return Set(vres);
}

bool Set::isRelation()const
{
   V<Set> vs=operator()();
   Z i,d;
   for (i=1;i<=n_;i++){
      d=vs[i].getTwoFactors().dim();
      if (d!=2) return false;
   }
   return true;
}

V<Set> Set::decompose()const
{
   Z maxFac=10;
   V<Set> res(maxFac);
   Z i,d;
   V<Set> vs=operator()();
   for (i=1;i<=n_;i++){
      d=vs[i].getFactors().dim();
      res[d]=res[d].add(vs[i]);
   }
   Z iNeeded=0;
   for (i=1;i<=maxFac;i++){
      Z di=res[i].card();
      if (di>0) iNeeded=i;
   }
   return res.resize(iNeeded);
```

```
}

V<Set>  Set::extractRelation()const
{
   V<Set> res(3);
   Z i,d;
   V<Set> vs=operator()();
   for (i=1;i<=n_;i++){
      V<Set> vsigf=vs[i].getTwoFactors();
      d=vsigf.dim();
      if (d==2){
         res[1]=res[1].add(vs[i]);
         res[2]=res[2].add(vsigf[1]);
         res[3]=res[3].add(vsigf[2]);
      }
   }
   if (res[1].isVoid()) return V<Set>(0);
   else return res;
}

bool Set::isFunction()const
      // returns true iff all elements of *this are wkp's
      // and if there apear no two different wkp's which agree
      // in their first component
{
   V<Set> vs=operator()();
   Z i,d;
   for (i=1;i<=n_;i++){
      d=vs[i].getTwoFactors().dim();
      if (d!=2) return false;
   }
   return true;
}

/*
V<Set> Set::makeFunction(const V<Set>& rel)
      // let rel be a result from extract Relation.
      // If rel.dim()==0 we return V<Set>(0). Else we
      // return the mapping from res[2].powerSet() to res[3].powerSet()
      // which is induced by relation rel[1].
      // not ready
{
   return V<Set>(0);
}
*/
```

175 cpmshapes.h

```
//? cpmshapes.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_SHAPES_H_
#define CPM_SHAPES_H_
/*
Description: Geometrical Shapes for 2D and 3D geometry
*/
#include <cpmdim.h>

namespace CpmDim2{
   #include <cpmshapesx.h>
}

namespace CpmDim3{
   #include <cpmshapesx.h>
}

#endif
```

176 cpmshapes.cpp

```
//? cpmshapes.cpp
//? Status of work 2008-10-25.
//?
#include <cpmshapes.h>
#include <cpmrvector.h>

namespace CpmDim2{
   #include <cpmshapesxcpp.h>
}

namespace CpmDim3{
   #include <cpmshapesxcpp.h>
}
```

177 cpmshapesx.h

```
//? cpmshapesx.h
//? Status of work 2008-10-25.
//?
using CpmFunctions::FncObj;

using CpmRoot::R;
using CpmRoot::Z;
using CpmRootX::R1;
using CpmRootX::Z1;
using CpmFunctions::F;
using CpmArrays::V;

//doc namespace CpmDimx{

//////////////////// class Form<> ////////////////////////////////////////////

template <class Dat>
class Form: public FncObj<Rdim,R>{ // shape determining function objects
   // a more direct version of F1<Rdim,Dat,R> to be used for
   // initialization in Shp<Dat>
   Dat dat;
       // streamable data
 public:
   Form(const Dat& dat_):dat(dat_){}
   R operator()(const Rdim& sp)const{ return dat.indFnc(sp);}
};

//////////////////////// class Shp<> ////////////////////////////////////////

template <class Dat>
class Shp:public SubsetOfSpace{ // shapes

protected:

   Dat dat;

public:

   typedef SubsetOfSpace Base;
   typedef Shp<Dat> Type;
   CPM_IO_V

   virtual SubsetOfSpace* clone()const{ return new Shp(*this);}

   Shp(const Dat& dat_=Dat(), const RefSys& rs=RefSys()):
      Base( F<Rdim,R>(new Form<Dat>(dat_)), rs ), dat(dat_)
   {
```

```
      set_x(dat.getCenter());
      set_r(dat.getRadius());
   }

   Dat getDat()const{ return dat;}
};

template <class Dat>
bool Shp<Dat>::prnOn(ostream& str)const
{
   if (!CpmRoot::writeTitle("Shp<>",str)) return false;
   if (!Base::prnOn(str)) return false;
   if (!dat.prnOn(str)) return false;
   return true;
}

template <class Dat>
bool Shp<Dat>::scanFrom(istream& str)
{
   RefSys rs_;
   if (!rs_.scanFrom(str)) return false;
   Spc x_;
   if (!x_.scanFrom(str)) return false;
   R1 r_;
   if (!r_.scanFrom(str)) return false;
      // x and r not used, but they have to be read
   DisPar dp_;
   if (!dp_.scanFrom(str)) return false;
   Dat dat_;
   if (!dat_.scanFrom(str)) return false;
   *this=Shp<Dat>(dat_,rs_);
   set_dp(dp_);
   return true;
}

//////////////////////////// class Cylinder ////////////////////////////


struct CylinderDat{ // data which determine a cylinder
      // Cylinder
   Rdim x0;
         // coordinate triplet of center
   R1 rho;
         // radius of the cylinder
   R1 hh;
         // half-height of the cylinder along the cylinder axis
   CylinderDat(){}
   CylinderDat(Rdim x0_,R rho_,R hh_):
      x0(x0_),rho(rho_),hh(hh_){}
   typedef CylinderDat Type;
```

```
   CPM_IO
   R getRadius()const{ return sqrt(hh*hh+rho*rho);}
   Rdim getCenter()const{ return x0;}
   R vol()const{
      R arBase=(Spc::dimension()==3 ? CpmRoot::Pi*rho*rho : 2*rho);
      return hh*arBase;
   }
   R indFnc(const Rdim&)const;
};


class Cylinder:public Shp<CylinderDat>{ // cylinder
public:
   typedef Shp<CylinderDat> Base;
   Cylinder(Rdim x0=Rdim(),R rho=0,R hh=0):Base(CylinderDat(x0,rho,hh)){}
   Cylinder(const Base& b):Base(b){}
   R vol()const{ return dat.vol();}
};

//////////////////////// class Boxoid ////////////////////////////////
// intermediate object between box and ellipsoid

struct BoxoidDat{ // data of which determine a boxoid

   Rdim x0;
      // coordinate list of center
   Rdim s;
      // list of half-axes
   R1 p;
      // for p=2 we get an ellipsoid, for larger p we approach the box
      // for p=1 one has a romboid
   BoxoidDat(){}
      // void Boxoid
   BoxoidDat(Rdim x0_, Rdim s_, R p_):x0(x0_),s(s_),p(p_){}
      // Boxoid with prescribed center and sites of the enclosing box.
      // Further, p_ is the form factor p described above
   typedef BoxoidDat Type;
   CPM_IO
   R getRadius()const;
   Rdim getCenter()const{ return x0;}
   R indFnc(const Rdim&)const;
};

class Boxoid:public Shp<BoxoidDat>{// objects between box and ellipsoid

public:
   typedef Shp<BoxoidDat> Base;
   Boxoid(Rdim x0=Rdim(),Rdim s=Rdim(),R p=0):Base(BoxoidDat(x0,s,p)){}
   Boxoid(const Base& b):Base(b){}
      // no explicit volume formula
```

```
};

///////////////// class Ellipsoid ////////////////////////////////////////

struct EllipsoidDat{ // data which determine an ellipsoid

   Rdim x0;
      // coordinate list of center
   Rdim s;
      // list of half-axes

   EllipsoidDat(){}
      // void Ellipsoid
   EllipsoidDat(Rdim x0_, Rdim s_):x0(x0_),s(s_){}
      // Ellipsoid with prescribed center and sites of the enclosing box.
      // Further, p_ is the form factor p described above
   typedef EllipsoidDat Type;
   CPM_IO
   R getRadius()const;
   Rdim getCenter()const{ return x0;}
   R vol()const;
   R indFnc(const Rdim&)const;
};

class Ellipsoid:public Shp<EllipsoidDat>{ // ellipsoids
public:
   typedef Shp<EllipsoidDat> Base;
   Ellipsoid(Rdim x0=Rdim(),Rdim s=Rdim()):Base(EllipsoidDat(x0,s)){}
   Ellipsoid(const Base& b):Base(b){}
   R vol()const{ return dat.vol();}
};

//////////////////////////////// class Box ////////////////////////////
// rectangular parallelepiped

struct BoxDat{ // data which determine a box

   Rdim x0;
      // coordinate list of center
   Rdim s;
      // list of half-axes (=half-sites)

   BoxDat(){}
      // void Box
   BoxDat(Rdim x0_, Rdim s_):x0(x0_),s(s_){}
      // Box with prescribed center and sites of the enclosing box.
      // Further, p_ is the form factor p described above
   typedef BoxDat Type;
   typedef BoxDat CloneBase;
   CPM_IO_V
```

```
      // I/0 functions are virtual in order to allow overwriting in
      // derived classes
   R getRadius()const;
   Rdim getCenter()const{ return x0;}
   R vol()const;
   R indFnc(const Rdim& )const;
};

class Box:public Shp<BoxDat>{ // rectangular parallelepipeds
public:
   typedef Shp<BoxDat> Base;
   Box(Rdim x0=Rdim(),Rdim s=Rdim()):Base(BoxDat(x0,s)){}
      // construction from center and half-sites such that
      // these data refer to the master system of refereence
   Box(const Base& b):Base(b){}
   Box(const V<Spc>& vs, const RefSys& rs);
      // a box with axes parallel to those of rs and enclosing
      // the points of vs in a minimum way.
   R vol()const{ return dat.vol();}
   V<Spc> ran(Z n)const;
   V<Spc> lattice(Z n, R site);
      // divides the box into identically sized cells
      // which are nearly cubic (or quadratic for Spc::dimension=2).
      // Each site s[i] satisfies  s[i] >= site and is as close to size
      // as a genuine fraction of the corresponding site of the Box
      // can be. Returned are nRes centers of cells, where nRes is the
      // minimum of n and the total number of cells. If n is smaller than
      // the total number of cells, the sequence of cell centers starts
      // with the the left lower corner and gets filled according to the
      // scheme (for 3D):
      //  x=corner[1];
      //  for (i=1;i<=ns[1];i++){
      //    x+=dx;
      //    xs[1]=x;
      //    y=corner[2];
      //    for (j=1;j<=ns[2];j++){
      //       y+=dy;
      //       xs[2]=y;
      //       z=corner[3];
      //       for (k=1;k<=ns[3];k++){
      //          l++;
      //          z+=dz;
      //          xs[3]=z;
      //          res[l]=refSys(xs);
      //       }
      //    }
      //  }
};

///////////////// class Sphere ////////////////////////////////
```

```
struct SphereDat{ // data which determine a sphere
      // intermediate object between box and ellipsoid
   Rdim x0;
      // coordinate list of center
   R1 r;
      // radius (= 'half-axis')

   SphereDat(){}
      // void Sphere
   SphereDat(Rdim x0_, R r_):x0(x0_),r(r_){}
      // Sphere with prescribed center and radius.

   typedef SphereDat Type;
   CPM_IO
   R getRadius()const{return r;}
   Rdim getCenter()const{ return x0;}
   R vol()const{return Spc::vol(r);}
   R indFnc(const Rdim& x)const{ return (x-x0).absVal()-r;}
};

class Sphere:public Shp<SphereDat>{ // spheres
public:
   typedef Shp<SphereDat> Base;

explicit Sphere(Rdim x0=Rdim(),R r=0):Base(SphereDat(x0,r)){}
      // constructor of the same design as all before
explicit  Sphere(const BndSph& bs):
   Base(SphereDat(refSys(bs.get_x()),bs.get_r())){}
      // special constructor from a bounding sphere
   Sphere(const Spc& sp, R r):
      Base(SphereDat(sp.toRdim(),cpmabs(r))){}
      // special constructor from center in Spc and radius
   Sphere(const Base& b):Base(b){}
   R vol()const{ return dat.vol();}
};
```

178 cpmshapesxcpp.h

```
//? cpmshapesxcpp.h
//? Status of work 2008-10-25.
//?
/*
   Description: dimension-independent part of implementation
   of geometrical objects
*/

using namespace CpmStd;

//////////////////////////// class Cylinder ////////////////////////////////

bool CylinderDat::prnOn(ostream& out)const
// return preliminary
{
   CpmRoot::writeTitle("CylinderDat",out);
   x0.prnOn(out);
   rho.prnOn(out);
   hh.prnOn(out);
   return true;
}

bool CylinderDat::scanFrom(istream& in)
// return preliminary
{
   x0.scanFrom(in);
   rho.scanFrom(in);
   hh.scanFrom(in);
   return true;
}

R CylinderDat::indFnc(const Rdim& x)const
   {
      using CpmArrays::R_Vector;
      R val1=Iv(x0[1]-hh,x0[1]+hh)(x[1]);
      Z d=Spc::dimension();
      R_Vector y(d-1);
      for (Z i=2;i<=d;i++){
         y[i-1]=x[i]-x0[i];
      }
      R val2=y.absVal()-rho;
      return CpmRootX::sup<R>(val1,val2);
   }

//////////////////////// class Boxoid ////////////////////////////////////////

bool BoxoidDat::prnOn(ostream& out)const
```

```
// return preliminary
{
   CpmRoot::writeTitle("BoxoidDat",out);
   x0.prnOn(out);
   s.prnOn(out);
   p.prnOn(out);
   return true;
}

bool BoxoidDat::scanFrom(istream& in)
// return preliminary
{
   x0.scanFrom(in);
   s.scanFrom(in);
   p.scanFrom(in);
   return true;
}

R BoxoidDat::indFnc(const Rdim& x)const
{
   Z d=x0.dim();
   R s1=0,s2=0;
   for (Z i=1;i<=d;i++){
      R ha=s[i]; // half axis
      R xi=CpmRoot::absVal(x[i]-x0[i]);
      ha=pow(ha,p);
      xi=pow(xi,p);
      s1+=ha;
      s2+=xi/ha;
   }
   R pInv=1./p;
   s1=pow(s1,pInv);
   s2=pow(s2,pInv);
   return s1*(s2-1.);
}

R BoxoidDat::getRadius()const
{
   if (p>2) return s.absVal();
   else{
      Z d=Spc::dimension();
      CpmArrays::R_Vector rv(d);
      for (Z i=1;i<=d;i++){
         R hs=CpmRoot::absVal(s[i]);
         rv[i]=hs;
      }
      return rv.sup();
   }
}
```

```
//////////////////////////// class Ellipsoid ////////////////////////////

bool EllipsoidDat::prnOn(ostream& out)const
// return preliminary
{
   CpmRoot::writeTitle("EllipsoidDat",out);
   x0.prnOn(out);
   s.prnOn(out);
   return true;
}

bool EllipsoidDat::scanFrom(istream& in)
// return preliminary
{
   x0.scanFrom(in);
   s.scanFrom(in);
   return true;
}

R EllipsoidDat::indFnc(const Rdim& x)const
{
   Z d=x0.dim();
   R s1=0,s2=0;
   for (Z i=1;i<=d;i++){
      R ha=s[i]; // half axis
      R xi=x[i]-x0[i];
      ha*=ha;
      xi*=xi;
      s1+=ha;
      s2+=xi/ha;
   }
   s1=sqrt(s1);
   s2=sqrt(s2);
   return s1*(s2-1.);
}

R EllipsoidDat::getRadius()const
{
   Z d=Spc::dimension();
   CpmArrays::R_Vector rv(d);
   for (Z i=1;i<=d;i++){
      R hs=CpmRoot::absVal(s[i]);
      rv[i]=hs;
   }
   return rv.sup();
}

R EllipsoidDat::vol()const
{
   R res= Pi;
```

```
   Z d=Spc::dimension();
   if (d==3) res*=(4./3);
   for (Z i=1;i<=d;i++) res*=s[i];
   return res;
}

//////////////////////////// class Box ////////////////////////////////////

bool BoxDat::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("BoxDat",str);
   cpmp(x0);
   cpmp(s);
   return true;
}

bool BoxDat::scanFrom(istream& str)
{
   cpms(x0);
   cpms(s);
   return true;
}

R BoxDat::vol()const
{
   R res=1;
   Z d=Spc::dimension();
   for (Z i=1;i<=d;i++) res*=(2*s[i]);
   return res;
}

R BoxDat::getRadius()const
{
   R res=0;
   Z d=Spc::dimension();
   for (Z i=1;i<=d;i++) res+=(s[i]*s[i]);
   res=sqrt(res);
   return res;
}

R BoxDat::indFnc(const Rdim& x)const
{
   return SubsetOfSpace::box(x,x0,s);
}

Box::Box(const V<Spc>& vs, const RefSys& rs)
{
   Z i,j,n=vs.dim(),m=dimVec;

   Group g=rs.toMasSys();
```

```
   V<Rdim> vsp(n);
   for (i=1;i<=n;i++){
      Spc pi=vs[i];
      pi*=g;
      vsp[i]=pi.toRdim();
   }
   VV<R> dir(m,n);
   for (i=1;i<=m;i++){
      for (j=1;j<=n;j++){
         dir[i][j]=vsp[j][i];
      }
   }
   V<Iv> ivDir(m);
   for (i=1;i<=m;i++) ivDir[i]=Iv(dir[i]);
   Rdim x0;
   Rdim s;
   for (i=1;i<=m; i++){
      x0[i]=ivDir[i].mean();
      s[i]=ivDir[i].radius();
   }
   Box box(x0,s);
   box*=!g;
   *this=box;
}


V<Spc> Box::ran(Z n)const
// random sequence of n points in the Box
{
   V<Spc> res(n);
   Z i,j,ir=0;
   Rdim xs;
   using CpmRoot::ranVal;
   Z d=Spc::dimension();
   for (i=1;i<=n;i++){
      for (j=1;j<=d;j++){
         R rr=ranVal(R(1.),ir++);
            // values between -1 and +1
         xs[j]=(dat.x0)[j]+rr*(dat.s)[j];
      }
      res[i]=refSys(xs);
   }
   return res;
}

V<Spc> Box::lattice(Z n, R site)
{
   cpmmessage("Box::lattice(...) started");
   if (site<=0 || n<1) cpmerror("Box::lattice(...): bad arguments");
   Z d=Spc::dimension();
```

```
Z i,j,k,l;
Rdim s=dat.s;
V<Z> ns(d);
V<R> ss(d);
Z nCells=1;

for (i=1;i<=d;i++){
   R site_i=2*s[i];
   Z nt=(Z)floor(site_i/site);
   ns[i]=(nt<=1 ? 1 : nt);
   ss[i]=(site_i/ns[i]);
   nCells*=ns[i];
}

Z nRes=cpminf(n,nCells);
   // number of points to be generated

R dx=ss[1];
R dy=ss[2];
R dz=(d==3) ? ss[3] : 0;

Rdim corner;
corner[1]=          (dat.x0)[1]-s[1]-0.5*dx;
corner[2]=          (dat.x0)[2]-s[2]-0.5*dy;
if (d==3) corner[3]=(dat.x0)[3]-s[3]-0.5*dz;

V<Spc> res(nRes);
Rdim xs;
R x,y,z;

l=0;
if (d==2){
   y=corner[2];
   for (i=1;i<=ns[2];i++){
      y+=dy;
      xs[2]=y;
      x=corner[1];
      for (j=1;j<=ns[1];j++){
         if (++l>nRes) break;
         x+=dx;
         xs[1]=x;
         res[l]=refSys(xs);
      }
   }
}
else{
   z=corner[3];
   for (i=1;i<=ns[3];i++){
      z+=dz;
      xs[3]=z;
```

```
         y=corner[2];
         for (j=1;j<=ns[2];j++){
            y+=dy;
            xs[2]=y;
            x=corner[1];
            for (k=1;k<=ns[1];k++){
               if (++l>nRes) break;
               x+=dx;
               xs[1]=x;
               res[l]=refSys(xs);
            }
         }
      }
   }
   cpmmessage("Box::lattice(...) done");
   return res;
}

//////////////////////////// class Sphere ////////////////////////////

bool SphereDat::prnOn(ostream& out)const
// return preliminary
{
   CpmRoot::writeTitle("SphereDat",out);
   x0.prnOn(out);
   r.prnOn(out);
   return true;
}

bool SphereDat::scanFrom(istream& in)
// return preliminary
{
   x0.scanFrom(in);
   r.scanFrom(in);
   return true;
}
```

179 cpmsi.h

```
//? cpmsi.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_SI_H
#define CPM_SI_H
/*
   Description: Physical quantities carrying 'dimension'
*/

#include <cpmq.h>
   // for class Q of rational numbers
   // (coded as numerator/denominator)
#include <cpmvr.h>
   // for forming Vr<Q> := Q_Vector
#include <cpmtypes.h>

namespace CpmPhysics{

   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmRoot::Q;
   using CpmRoot::Word;

enum TYPE { SI, NATCONST, UNSPECIFIED};

class Si{  // class for description of physical quantities
   // (including dimensions) especially with respect to the SI system.
   // Some exploratory methods to deal also with other systems of units,
   // e.g. Planck's units bases on natural constants c,h,G

   typedef CpmArrays::Vr<Q> Q_Vector;
   typedef Si Type;
   typedef R ScalarType;

// independent data
   R val;   // dimensionless number
   Q_Vector qv;
      // x = val * m^d1 * kg^d2 * s^d3 * A^d4 * K^d5 * mol^d6 * cd^d7
      // for di=qv[i]
   TYPE type; // sets the meaning of the dimensional exponents
// dependent data
   bool hasDim;   // false if the quantity is 'dimension-less'
   void adjust();
      // sets the dependent data according to the independent ones
   Q& operator[](Z i){ return qv[i];} // after change one may have to
   // call
      // adjust, so this function should not be public
```

```
public:
// constructors
   Si(R valIn, Q_Vector dimensionsIn, TYPE t=SI);
      // dimensionsIn is cut to length nDim if too long and filled by
      // zero if too short

   explicit Si(R v=1): qv(nDim),val(v),hasDim(0),type(SI){}
      // default Si is the dimension-less 1

   Si( R in_val, Q in_m, Q in_kg,
      Q in_s, Q in_A=Q(), Q in_K=Q(), Q in_mol=Q(), Q in_cd=Q() );
      // constructor from numerical value and fractional dimensions for
      // each of the basic units

   Si( R in_val, Z in_m, Z in_kg,
      Z in_s, Z in_A=0, Z in_K=0, Z in_mol=0, Z in_cd=0 );
      // constructor from numerical value and integer dimensions for each
      // of the basic units
// interfaces (very similar to those of Q)
   CPM_IO
   CPM_ORDER
   CPM_TEST_X
   CPM_DESCRIPTORS
   CPM_CONJUGATION
   CPM_SUM_M
   CPM_PRODUCT_M
   CPM_PRODUCT_SCALAR_M
    // this does not incude Type + ScalarType
   CPM_DIFFERENCE
   CPM_DIVISION

//overloading pow() for type SI
   Si pow(Q y)const;
   Si pow(Z y)const;
   Si sqrt()const{ return pow(Q(Z(1),Z(2)));}

// access methods
   TYPE getType()const{ return type;}
   R operator()(void)const{ return val;}
      // numerical value
   const Q& operator[](Z i)const{ return qv[i];}
// For Si x=...;
// and x.getType()==SI we have
// x=x()*m^x[1]*kg^x[2]*s^x[3]*A^x[4]*K^x[5]*mol^x[6]*cd^x[7]
//
// for x.getType()==NATCONST we presently don't seriously consider
// quantities of dimension of temperature, amount of substance,
// and luminous intensity so that we did not fix natural constants
// describing them. We thus put:
```

```
// x=x()*c^x[1]*h^x[2]*G^x[3]*e^x[4]*K^x[5]*mol^x[6]*cd^x[7]

// express a quantiy in terms of other base quantities

   Si convert(const CpmArrays::V<Si>& newBase, TYPE newType)const;
      // given a array of SI quantities we return a quantity
      // which is formal SI but actually give value and dimension
      // product referring to the quantities from newBase. Tool for
      // implementing the next function

   Si fromSiToNat()const;
      // applying the convert function for converting SI to NATCONST
      // meaning

   static void PlanckUnits(const Word& fileName="natunits.txt");
      // nice demonstration of function Si fromSiToNat()const;

   void test(Z stop=0);
      // old test function, not particularly well designed

   static Z nDim; // always 7 for SI

   static Si m;
   static Si kg;
   static Si s;
   static Si A;
   static Si K;
   static Si mol;
   static Si cd;

   static Si c;
   static Si h;
   static Si h_; // h divided by 2*Pi
   static Si e;
   static Si G;
   static Si u;
   static Si eps0;
   static Si mu0;
};

// derived classes for simple introduction of physical quantities
// Example: terrestial acceleration g may be introduced as
// ACCELERATION g(9.81);
// This fixes the value in SI units as 9.81 and the dimension as that
// of an acceleration: m^1*s^-2

// ************************** L E N G T H  Class **********************

class LENGTH : public Si{      // LENGTH() = m
public:
```

```
   explicit   LENGTH(R valIn=1):Si(valIn,1,0,0,0,0,0,0){}
};

// *************************** M A S S  Class ***************************
class MASS : public Si{    // MASS() = kg
public:
   explicit   MASS(R valIn=1):Si(valIn,0,1,0,0,0,0,0){}
};

// *************************** T I M E Class ***************************
class TIME : public Si{    // TIME() = s
public:
   explicit   TIME(R valIn=1):Si(valIn,0,0,1,0,0,0,0){}
};

// ************** (Electric) C U R R E N T  Class ********************

class CURRENT : public Si{ // CURRENT() = A
public:
   explicit   CURRENT(R valIn=1):Si(valIn,0,0,0,1,0,0,0){}
};

// ************* (Thermodynamic) T E M P E R A T U R E  Class   *********
class TEMPERATURE : public Si{ //  TEMPERATURE() = K
public:
   explicit   TEMPERATURE(R valIn=1):Si(valIn,0,0,0,0,1,0,0){}
};

// *************   (Amount of) S U B S T A N C E Class  *************
class SUBSTANCE : public Si{  // SUBSTANCE() = mol
public:
   explicit   SUBSTANCE(R valIn=1):Si(valIn,0,0,0,0,0,1,0){}
};

// *************** (Luminous) I N T E N S I T Y Class  ***************
class INTENSITY : public Si{  // INTENSITY = cd
public:
   explicit   INTENSITY(R valIn=1):Si(valIn,0,0,0,0,0,0,1){}
};

// *************** (Plane) A N G L E Class  ***************
class ANGLE : public Si{ // unit rad = radian = 1
public:
   explicit   ANGLE(R valIn=1):Si(valIn,0,0,0,0,0,0,0){}
};

// *************** S O L I D  A N G L E Class  ************************
class SOLID_ANGLE : public Si{ // unit sterrad = steradian = 1
public:
   explicit   SOLID_ANGLE(R valIn=1):Si(valIn,0,0,0,0,0,0,0){}
```

```
};

// ************************* V E L O C I T Y   Class ********************

class VELOCITY : public Si{         // m*s^-1
public:
   explicit   VELOCITY(R valIn=1):Si(valIn,1,0,-1,0,0,0,0){}
};

// ************************* A C C E L E R A T I O N   Class ************

class ACCELERATION : public Si{      // m*s^-2
public:
   explicit   ACCELERATION(R valIn=1):Si(valIn,1,0,-2,0,0,0,0){}
};

// ************************** A R E A  ********************************

class AREA : public Si{      // m^2
public:
   explicit   AREA(R valIn=1):Si(valIn,2,0,0,0,0,0,0){}
};


// ************************** V O L U M E ******************************

class VOLUME : public Si{      // m^3
public:
   explicit   VOLUME(R valIn=1):Si(valIn,3,0,0,0,0,0,0){}
};

// ************************* D E N S I T Y   Class **********************

class DENSITY : public Si{              // kg*m^-3
public:
   explicit   DENSITY(R valIn=1):Si(valIn,-3,1,0,0,0,0,0){}
};

// ************************* F O R C E   Class **************************

class FORCE : public Si{      // N == m * kg * s^-2
public:
   explicit   FORCE(R valIn=1):Si(valIn,1,1,-2,0,0,0,0){}
};

// ************************* E N E R G Y  Class *********************

class ENERGY : public Si{      // J = N*m == m^2 * kg * s^-2
public:
   explicit   ENERGY(R valIn=1):Si(valIn,2,1,-2,0,0,0,0){}
```

```
};


// ************************* C H A R G E   Class ************************

class CHARGE : public Si{   // electrical charge == Coulomb [C] = s*A
public:
   explicit   CHARGE(R valIn=1):Si(valIn,0,0,1,1,0,0,0){}
};

// ************************* R E S I S T A N C E   Class ****************

class RESISTANCE : public Si{ // electrical resistance Ohm = V/A
public:
   explicit   RESISTANCE(R valIn=1):Si(valIn,2,1,-3,-2,0,0,0){}
};

// ************************* V O L T A G E   Class **********************

class VOLTAGE : public Si{ // V
public:
   explicit   VOLTAGE(R valIn=1):Si(valIn,2,1,-3,-1,0,0,0){}
};

} // namespace

#endif
```

180 cpmsi.cpp

```
//? cpmsi.cpp
//? Status of work 2008-10-25.
//?
#include <cpmsi.h>
#include <cpmconstphys.h>
#include <cpmrmatrix.h>

using namespace CpmPhysics;
using namespace CpmRoot;
using namespace CpmArrays;

Z Si::nDim=7;

Si Si::m(1.,1,0,0,0,0,0,0);
Si Si::kg(1.,0,1,0,0,0,0,0);
Si Si::s(1.,0,0,1,0,0,0,0);
Si Si::A(1.,0,0,0,1,0,0,0);
Si Si::K(1.,0,0,0,0,1,0,0);
Si Si::mol(1.,0,0,0,0,0,1,0);
Si Si::cd(1.,0,0,0,0,0,0,1);

Si Si::c(c_PHYS,1,0,-1,0,0,0,0);
Si Si::e(e_PHYS,0,0,1,1,0,0,0);
Si Si::h(h_PHYS,2,1,-1,0,0,0,0);
Si Si::h_(h__PHYS,2,1,-1,0,0,0,0);
Si Si::G(G_PHYS,4,-1,-2,0,0,0,0);
Si Si::u(mu_PHYS,0,1,0,0,0,0,0);
Si Si::eps0(eps0_PHYS,-3,-1,4,2,0,0,0);
Si Si::mu0(my0_PHYS,1,1,-2,-2,0,0,0);

void Si::adjust(void)
{
   Z d=qv.dim();
   hasDim=false;
   for (Z i=1;i<=d;i++){
      if (qv[i]!=Q()) { hasDim=true; return;}
   }
   if (type==SI){
      if (d!=nDim)
         cpmerror("Si::adjust(): Si quantity dimension mismatch");
   }
}

Si::Si( R in_val, Q in_m, Q in_kg,
   Q in_s, Q in_A, Q in_K, Q in_mol, Q in_cd ):
      val(in_val),qv(nDim),type(SI)
{
```

```
   val=in_val;
   qv[1]=in_m;
   qv[2]=in_kg;
   qv[3]=in_s;
   qv[4]=in_A;
   qv[5]=in_K;
   qv[6]=in_mol;
   qv[7]=in_cd;
   adjust();
}

Si::Si(R in_val,Z in_m,Z in_kg,Z in_s,Z in_A,Z in_K,Z in_mol,Z in_cd ):
   val(in_val),qv(nDim),type(SI)
{
   qv[1]=Q(in_m);
   qv[2]=Q(in_kg);
   qv[3]=Q(in_s);
   qv[4]=Q(in_A);
   qv[5]=Q(in_K);
   qv[6]=Q(in_mol);
   qv[7]=Q(in_cd);
   adjust();
}

Si::Si(R valIn, Q_Vector qvIn, TYPE t):val(valIn), qv(nDim),type(t)
   // qvIn is cut to length nDim if too long and filled by
   // zero if too short
{
   Z dMin=CpmRootX::inf(nDim,qvIn.dim());
   for (Z i=1;i<=dMin;i++) qv[i]=qvIn[i];
   adjust();
}

Si Si::convert(const V<Si>& newBase, TYPE newType)const
// See UM/SI 95-4-21/1-2 Conversion of Units
{
   const Z nOK=10;
   const R eps=0.001;
   R_Matrix a(nDim,nDim);
   Z i,j;
   for (i=1;i<=nDim;i++) for (j=1;j<=nDim;j++)
      a[j][i]=(R)(newBase[i][j]);    // transposition implied
   a=!a;
   R_Vector x(nDim);
   for (i=1;i<=nDim;i++)
      x[i]=(R)(qv[i]);
   R_Vector y=a*x;  // no transposition here
   Si res;
   for (i=1;i<=nDim;i++){
      Q q(y[i],eps,nOK);
```

```
      res[i]=q;
   }
   R prod=val;
   for (i=1;i<=nDim;i++){
      R ri=(newBase[i]).val;
      R ei=-res[i];
      prod*=::pow(ri,ei);
   }
   res.val=prod;
   res.type=newType;
   res.adjust();
   return res;
}

Si Si::fromSiToNat()const
{
   V<Si> base(nDim);
   base[1]=c;
   base[2]=h;
   base[3]=G;
   base[4]=e;
// natural units for these last three types of quantities
// are not so convincingly determined; let them be!
   base[5]=TEMPERATURE();
   base[6]=SUBSTANCE();
   base[7]=INTENSITY();
   return convert(base,NATCONST);
}

Si& Si::operator+=(const Si& x)
{
   if (type!=x.type)
      cpmerror("Si: += : type mismatch");
   if (qv != x.qv)
      cpmerror("Si: += : dimension mismatch");
   val+=x.val;
   return *this;
}

Si& Si::operator*=(const Si& x)
{
   if (type!=x.type) cpmerror("Si: *= : type mismatch");
   val*=x.val;
   qv+=x.qv;
   adjust();
   return *this;
}

Si& Si::operator*=(const R& s)
{
```

```
   val*=s;
   return *this;
}

Si Si::neg(void)const
{
   return Si(-val,qv);
}

Si Si::inv(void)const
{
   if (val==0.) cpmerror("Si::inv(): val=0");
   return Si(1./val,-qv);
}

Si Si::net(Z i)const
{
   if (i==0) return Si(0,qv);
      // there is a additive neutral for each value of qv !
   else return Si(1);
}

R Si::absSqr()const{ R r=cpmabs(val); return r*r;}

R Si::absVal()const{ return cpmabs(val);}

Si Si::con(void)const{ return *this;}

R Si::disVal(const Si& z)const
{
   if (qv!=z.qv) return 1; // quantities of different dimension
      // are maximally different
   else return CpmRoot::disVal(val,z.val);
}

namespace{

R pow_(R x, Z k)
// k-th power of x realized as successive multiplication
{
   if (x==0.){
      cpmwarning("Si::pow_(...) negative power of 0 set to 0");
      return 0.;
   }
   Z n;
   if (k<0){ n=-k; x=1./x;} else n=k;
   R res=x;    // one factor
   for (int i=1;i<n;i++) res*=x; // n-1 multiplications
       // thus in toto n factors
   return res;
```

```
}

} // namespace

Si Si::pow(Q q)const
{
   R resVal;
   if (q.denominator()==1){
      Z qn=q.numerator();
      resVal=pow_(val, qn);
      return Si(resVal,qv*q);
   }
   else if (val<0){
      cpmerror("Si::pow(Q): broken power of a negative value");
      return *this; // never come here
   }
   else{
      resVal=::pow(val,(R)q);
      return Si(resVal,qv*q);
   }
}

// overloading pow() with SI base and int exponent
Si Si::pow(Z y)const
{
   R resVal;
   resVal=pow_(val, y);
   return Si(resVal,qv*Q(y));
}

Word Si::nameOf(void)const{ return Word("Si");}

Word Si::toWord()const
{
   Word res=cpmwrite(val);
   if (!hasDim) return res;

   V<Word> u(nDim);

   if (type==SI){
      u[1]="m";
      u[2]="kg";
      u[3]="s";
      u[4]="A";
      u[5]="K";
      u[6]="mol";
      u[7]="cd";
   }
   else if (type==NATCONST){
      u[1]="c";
```

```
      u[2]="h";
      u[3]="G";
      u[4]="e";
      u[5]="K";
      u[6]="mol";
      u[7]="cd";
   }
   else {
      u[1]="e1";
      u[2]="e2";
      u[3]="e3";
      u[4]="e4";
      u[5]="e5";
      u[6]="e6";
      u[7]="e7";
   }
   for (Z i=1;i<=nDim;i++){
      Q qi=qv[i];
      if (qi!=Q()) res&=" * "&u[i]&"^"&qi.toWord();
   }
   return res;
}

bool Si::prnOn( ostream& str ) const
{
   CpmRoot::writeTitle("Si",str);
   cpmp(val);
   cpmp(qv);
   cpmp((Z)type);
   return true;
}

bool Si::scanFrom (istream& str)
{
   cpms(val);
   cpms(qv);
   Z zt;
   cpms(zt);
   type=(TYPE)zt;
   adjust();
   return true;
}

Z Si::hashVal() const
{
   R a=val;
   for (Z i=1;i<=nDim; i++) a+=cpmabs((R)qv[i]);
   return CpmRoot::hashVal<R>(a);
}
```

```
Si Si::ranVal(Z j)const
// ranVal should not change the qv, since one
// would get incompatible test values for addition
   {
      R valR=cpmran(val,j);
      return Si(valR,qv,type);
   }

Si Si::testVal(Z tsz)const
{
   R r=1;
   Q q(Z(1));
   r=CpmRoot::testVal(r,tsz);
   Q q0=q.testVal(tsz);
   Si res(*this); // res has the same type as *this
   res.val=r;
   Z i,j=tsz;
   for (i=1;i<=nDim;i++) (res.qv)[i]=q0.ranVal(j++);
   res.adjust();
   return res;
}


Z Si::com(const Si& z) const
{
   if (val<z.val) return 1;
   if (val>z.val) return -1;

   if (qv<z.qv) return 1;
   if (qv>z.qv) return -1;

   return 0; // *this==v
}

void Si::test(Z stop)
 {
   const Z tvs=16;
   Si q(1);
   q.testVal(tvs);
   cout<<endl<<"testVal = "<<q;
   //if (stop) getchar();
   Z i=137;
   Si q1=q.ranVal(i++);
   cout<<endl<<" First randomly generated test object q1: "<<endl;
   cout<<q1;
   //if (stop) getchar();

   Si q2=q.ranVal(i++);
   cout<<endl<<" Second randomly generated test object q2: "<<endl;
   cout<<q2;
```

```
 //if (stop) getchar();

 Si q3=q.ranVal(i++);
 cout<<endl<<" Third randomly generated test object q3: "<<endl;
 cout<<q3;
 //if (stop) getchar();

 using CpmRoot::ranVal;
 using CpmRoot::absVal;

 R sum=0;
 R r;
 R r1=ranVal(r,i++);
 R r2=ranVal(r,i++);

 cout<<endl<<"Test of *,+,- :";
 Si q4=(r1+r2)*q3;
 Si q5=-q3*r1+r2*(-q3);
 q5+=q4;

 cout<<endl<<"q5="<<q5<<" should have value = 0 ";

 sum+=q5.absVal();

 cout<<endl<<" errorSum= "<<sum<<" 0 is OK";
// if (stop) getchar();

 cout<<endl<<"Test of ! * /:";
 q4=!(q1*q2*q3);
 q5=!q3*!q1*!q2;
 q5/=q4;
 cout<<endl<<"q5="<<q5<<" should be number 1 ";
 Si q6;
 q6=q6.net(1);
 q5-=q6;
 cout<<endl<<"q5="<<q5<<" should be number 0 ";
 sum+=q5.absVal();
 cout<<endl<<" errorSum= "<<sum<<" 0 is OK";
 //if (stop) getchar();

 cout<<endl<<"Demo of calculations with natural constants";
  LENGTH m(1);
 TIME s(1);
 Si cVal=c*s/m; // SI::c is static
 cout<<endl<<"value of c = "<<cVal;

 ACCELERATION g(9.81);
 TIME t(2);
 Si s1=(g/2)*t*t;
 cout<<endl<<" s1= "<<s1<< " should be the length 19.62 m";
```

```
   cout<<endl<<"Test of file I/O: ";

   Si q7,q8,q9;

   ofstream out("testdata.txt");
   if (!out) cpmerror("Si::test(): output file cannot be opened");
   out<<q1;
    out<<q2;
     out<<q3;
   out.close();

   ifstream in("testdata.txt");
   if (!in) cpmerror("Si::test(): input file cannot be opened");
   in>>q7;
     in>>q8;
     in>>q9;

   cout<<endl<<" This was read from file: "<<endl
   <<q7<<endl<<q8<<endl<<q9<<endl<<" looks normal ?";
   in.close();
   sum+=(q7-q1).absVal();
   sum+=(q8-q2).absVal();
   sum+=(q9-q3).absVal();
   cout<<endl<<" errorSum= "<<sum<<", should be close to 0";
   cout<<endl<<" End of test of class "<<"Si"<<endl;
 }

void Si::PlanckUnits(const Word& fileName)
{
   const Z nx=8;
   ofstream out(-fileName);
   V<Si> x(nx);
   x[1]=LENGTH();
   x[2]=MASS();
   x[3]=TIME();
   x[4]=CURRENT();
   x[5]=VOLTAGE();
   x[6]=RESISTANCE();
   x[7]=Si::eps0;
   x[8]=Si::u;
   V<Word> w(nx);
   w[1]="LENGTH";
   w[2]="MASS";
   w[3]="TIME";
   w[4]="CURRENT";
   w[5]="VOLTAGE";
   w[6]="RESISTANCE";
   w[7]="eps0";
   w[8]="u";
   for (Z i=1;i<=nx;i++){
```

```
      Word wi=w[i]&": ";
      wi&=x[i].toWord();
      wi&=" = ";
      wi&=x[i].fromSiToNat().toWord();
      cout<<endl<<wi;
      out<<endl<<wi;
   }
}
```

181 cpmsmoothlocalinterpol.h

```
//? cpmsmoothlocalinterpol.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_SMOOTHLOCALINTERPOL_H_
#define CPM_SMOOTHLOCALINTERPOL_H_
/*
   Description : Smooth interpolation. Gives a smooth interpolating
        curve for any finite list of prescribed real values.
        The list of x-values has to be in ascending order but needs not
        to be aequidistant. (Smooth means that derivatives of any order
        exist.) For details, see the comment to function sli.

*/
#include <cpmtypes.h>
#include <cpmv.h>

namespace CpmArrays{

   using CpmRoot::R;
   using CpmRoot::Z;
   using CpmRootX::R1;
   using CpmRootX::Z1;
   using CpmRoot::Word;

//////////////////// class IpolControl ////////////////////////////

class IpolControl{ // interpolation control
// save arrangement of the parameters which control interpolation
// The meaning of parameters m and method is as in function slint()
// 'smooth local interpolation'(see Kodak-TR 278844A) Here is the
// explanation:
// m is the order of the interpolation method (m=1,2,..., m<=n, n= number
// of data points) i.e. the number of data points entering the basic
// local interpolation function (m=3 for quadratic interpolation).
// method is 0 or 2 for polynomial interpolation and 1 for Bulirsch Stoer
// rational diagonal interpolation.
// The number of points used in interpolation is m if x is
// more than m/2 points away from the boundary. If method=2, things are
// arranged such, that coming closer to the boundary reduces the number
// of points which are used. This is not only useful for interpolation;
// more important is the improvement in extrapolation properties.
// For universal usage there is included normal (not smooth)
// linear interpolation as method=3.
// Notice that such a expensive operation as testing the (potentially
// very long) list of x-values should not be done with each computation
// of an interpolated value. So, the two data monoTest and eps will come
// into action not before interpolation gets embeded in a class (which
```

```
// will be R_func) where preparatory activities can be delegated to
// a constructor which has to be called only once for a certain set of
// data.
   Z1 m;
   Z1 method;
// the next two data will be used in R_Func
   Z1 monoTest;
   R1 eps;
   void adjust();
   typedef IpolControl Type;
public:
   CPM_IO
   IpolControl(Z mI=4, Z methodI=0, Z monoTestI=0):
   m(mI),method(methodI),monoTest(monoTestI),eps(1e-6)
   { adjust();}

   explicit IpolControl(const Word& w);
      // More convenient constructor for normal applications.
      // w=polygon,first_order,second_order,third_order,fourth_order,
      // fifth_order, sixth_order.
      // This is broken linear (non-smooth) for w="polygon", and smooth
      // interpolation (method=0) of respective order else.
      // Monotony test of x-list is enabled since reading from file or
      // careless programming may easily create lists that fail to
      // be monotone.

   Z get_m()const{return m;}
   Z get_method()const{return method;}
   Z get_monoTest()const{return monoTest;}
   void set_monoTest(bool b){ monoTest=(b==true ? 1:0);}
   R get_eps()const{ return eps;}
   void set_eps(R epsIn){ eps=epsIn;}
};

R locIpol(R x, const V<R>& xList, const V<R>& yList,
   const IpolControl& c=IpolControl());
   // 'modern form with class for controlling parameters

R locIpol(R ic, V<R> const& yList, IpolControl const& c=IpolControl());
   // local interpolation
   // Here ic is a 'continuous index'. The interpolated function f
   // has the property f(x)=yList[(Z)x] if x is integer (by value,
   // not by type).
   // This is here for uniformity with
   // R interpolate(R ic, R_Vector const& x) in cpmvectormore.h.
   // If this function has to be called many times for the same argument
   // yList, one should consider to create xList once ( xList[i]=i)
   // and to call the previous function.

R locIpolPos(R x, const V<R>& xList, const V<R>& yList,
```

```
   const IpolControl& c=IpolControl());

void prepIpol(V<R>& x, V<R>& y, R eps=1e-6);
// name stands for 'prepare Interpolation'
// Tests whether the list of x's is stricly increasing as expected for
// interpolation purposes. If two x values are very close together they
// get merged and the y-value replaced by the mean

} // namespace

#endif
```

182 cpmsmoothlocalinterpol.cpp

```
//? cpmsmoothlocalinterpol.cpp
//? Status of work 2008-10-25.
//?
#include <cpmsmoothlocalinterpol.h>

using CpmRootX::parity;

namespace CpmArrays{

void IpolControl::adjust()
{
   if (m<=0) cpmerror("IpolControl(...) m<=0");
   if (m>=20) cpmerror("IpolControl(...) m>=20");
   if (method<0) cpmerror("IpolControl(...) method<0");
   if (method>3) cpmerror("IpolControl(...) method>3");
      // limit 20 is ad hoc but large values of
      // m make no sense
}

IpolControl::IpolControl(const Word& w):monoTest(1),method(0),eps(1e-6)
 // w=polygon,first_order,second_order,third_order,...sixth_order
{

   if (w=="polygon"){
      m=1; method=3;
   }
   else if (w=="first_order"){
      m=2;
   }
   else if (w=="second_order"){
      m=3;
   }
   else if (w=="third_order"){
      m=4;
   }
   else if (w=="fourth_order"){
      m=5;
   }
   else if (w=="fifth_order"){
      m=6;
   }
   else if (w=="sixth_order"){
      m=7;
   }
   else{
      cpmerror("IpolControl(const Word& w): unvalid value for w");
   }
```

```
}

bool IpolControl::prnOn(ostream& str)const
{
   CpmRoot::writeTitle("IpolControl",str);
   cpmp(m);
   cpmp(method);
   cpmp(monoTest);
   cpmp(eps);
   return true;
}

bool IpolControl::scanFrom(istream& str)
{
   cpms(m);
   cpms(method);
   cpms(monoTest);
   cpms(eps);
   return true;
}

namespace{  // anonymus

const R HUGE_UM=1.e25;
const R TINY=1./HUGE_UM;

/*
Z parity(Z k)
{
   if (k%2) return -1; else return 1;
}
*/

void polint(V<R> xa, V<R> ya, R x, R& y, R& dy)
{
   Z n=xa.dim();
   cpmassert(n==ya.dim(),"polint(...)");
   Z i,m,ns=1;
   R den,dif,dift,ho,hp,w;
   dif=cpmabs(x-xa[1]);
   V<R> c(n);
   V<R> d(n);
   for (i=1;i<=n;i++) {
      if ( (dift=cpmabs(x-xa[i])) < dif) {
         ns=i;
         dif=dift;
      }
      c[i]=ya[i];
      d[i]=ya[i];
   }
```

```
   y=ya[ns--];
   for (m=1;m<n;m++) {
      for (i=1;i<=n-m;i++) {
         ho=xa[i]-x;
         hp=xa[i+m]-x;
         w=c[i+1]-d[i];
         if ( (den=ho-hp) == 0.0) cpmerror("Error in routine POLINT");
         den=w/den;
         d[i]=hp*den;
         c[i]=ho*den;
      }
      y += (dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
   }
}

void ratint(V<R> xa, V<R> ya, R x, R& y, R& dy)
{
   Z n=xa.dim();
   cpmassert(n==ya.dim(),"ratint(...)");
   Z m,i,ns=1;
   R w,t,hh,h,dd;
   V<R> c(n);
   V<R> d(n);
   hh=cpmabs(x-xa[1]);
   for (i=1;i<=n;i++) {
      h=cpmabs(x-xa[i]);
      if (h == 0.0) {
         y=ya[i];
         dy=0.0;
         return;
      } else if (h < hh) {
         ns=i;
         hh=h;
      }
      c[i]=ya[i];
      d[i]=ya[i]+TINY;
   }
   y=ya[ns--];
   for (m=1;m<n;m++) {
      for (i=1;i<=n-m;i++) {
         w=c[i+1]-d[i];
         h=xa[i+m]-x;
         t=(xa[i]-x)*d[i]/h;
         dd=t-c[i+1];
         if (dd == 0.0){
            y=HUGE_UM;
            dy=HUGE_UM;
             cpmwarning("Problem in routine RATINT");
            return;
         // this is changed 91-11-12 for improved flexibility
```

```
         }
         dd=w/dd;
         d[i]=c[i+1]*dd;
         c[i]=t*dd;
      }
      y += (dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
   }
   return;
}

/********************** my functions ******************************/

R s_log(R x)
// s_log stands for save log: argument is allows to be negative
{
   static R x_min=1e-50;
   static R y_min=log(x_min);
   if (x<x_min) return y_min;
   return log(x);
}

R s_exp(R x)
// s_exp stands for save exponential function, argument may be
// arbitrarily
// large
{

   static R y_max=1e50;
   static R x_max=log(y_max);
   if (x>x_max) return y_max;
   return exp(x);
}

R hill(R x, R p)
// For any positive parameter p, the function x |---> hill(x,p)
// is a smooth (i.e. derivatives of any order exist) non-increasing
// function. It has the constant value 1 for x<=0 and the constant value
// 0 for x>=1.  So, it is a smooth down-hill connecting level 1 with
// level
// 0.
// The whole curve is radially symmetrical to the point (x,y)=(0.5,0.5).
// p=1 gives a nice shape. p=0.5 seems to give smaller absolut
// interpolation
// error but finer 'ripple' of the error curve.
// The idea behind is the following: The negative hyperbolic tangent
// function
// is already similar to such a hill. A transformation of the argument
// brings the x-values 0 and 1 to infinity before they are inserted
// into the hyperbolic tangent. The normal expression for the latter
// is transformed in order to call exp only once. Futher, to avoid
```

```
// overflow riscs in exp, this function is called only for negative
// arguments.p=0.810747 minimizes the L2-Norm of the second derivative.
{
   if (x<=0) return(1.);
   if (x>=1) return (0.);
   R y=2*p*(x-0.5)/(x*(x-1));
   if (y<=0) {
      y=exp(y); y=y/(1+y);
   }
   else {
      y=exp(-y); y=1/(1+y);
   }
   // so the argument of exp is always negativ
   // hence no overflow errors can result by calling exp.
   // Since the values of exp are never negative, 1+y is never
   // zero.
   return(y);
}

/*

R hill(R x, R p)
// For any positive parameter p, the function x |---> hill(x,p)
// is a smooth (i.e. derivatives of any order exist) non-increasing
// function. It has the constant value 1 for x<=0 and the constant value
// 0 for x>=1.  So, it is a smooth down-hill connecting level 1 with
// level
// 0.
// The whole curve is radially symmetrical to the point (x,y)=(0.5,0.5).
// p=1 gives a nice shape.
// experimental simple replacement
{
   if (x<=0) return(1.);
   if (x>=1) return (0.);
   R x_=(1-x);
   R x2_=x_*x_;
   return x2_/(x2_+x*x);
}
*/


Z lonei(Vl<R> x, R x_)
// lonei stands for lowest neighbour.
// Let x[] be an ascending list x[0] < x[1] <...< x[n-1]. For a given
// real number x_, we find the largest integer j such that x[j] <= x_.
// This j is the return value of the function. If all x[...] are larger
// than x_, we return -1.
// error eliminated 92-8-4
{
   Z n=x.dim();
```

```
   cpmassert(n>0,"lonei(...)");
   Z n_=n-1; R xb=x[0]; R xe=x[n_]; // for shortness

   if (x_>=xe) return(n_);
   if (x_<xb)   return(-1);
   Z k=  (Z)(n_*(x_-xb)/(xe-xb));       // j=k for equidistant lists x[]
   if ( x_==x[k] ) return k;          // for equidistant lists
                              // this is the normal exit
   Z j=k;
   if ( x_ > x[k] ) {                 // find j by increasing k
      while ( x_ >= x[j+1] ) j++;     // x_>=x[j] has to be achieved
                              // if this holds for the next j,
                              // go to the next j
   }
   else {                              // find j by decreasing k.
      while ( x_ < x[j] ) j--;        // Since x_ has to be >= x[j],
                              // the condition x_<x[j] says
                              // that j has to be changed.
                              // error till 92-8-4
   }
   return j;
}

void range(R k, Z n, Z m, Z& l, Z& r, Z method)
/* k is integer or half-integer=m of my paper, m is P of my paper
*l is L(m,P), *r is R(m,P)
*/
{

   Z ir=(Z)(k+0.5*m);      // primarily (Z)(k+0.5*(m-1)+0.5)
   Z il=ir-m+1;

   if (method==2){ // variable order when approaching boundary
      Z flag=0;
      if (il<0) {il=0; flag=1;}
      if (ir>=n) {ir=n-1; flag=2;}
      if (parity(ir-il+1)!=parity(m)){
         if (flag==1) ir--;   // 93-1-19 changed ir++ in ir--
         if (flag==2) il++;   // 93-1-19 changed ir-- in ir++
            // this preferrs using less points near the boundary
            // compared with using more points; seems to be better for
            // robust extrapolation properties
      }                         // insure that ir-il+1 remains of
                          // the same parity as m    91-11-8
   }
   else {
      if (il<=0){ il=0; ir=m-1;}
      if (ir>=(n-1)){ ir=n-1; il=n-m;}       // constant order
      // since variable order is not useful for rational interpolation
      // (since especially used for extrapolation)
```

```
   }
   if (ir < il) ir=il;  // no real reason, should not happen anyway
   r=ir; l=il;
}

R pu(Vl<R> x, Z n, R x_, R k, R w_0, R w_1)
/* pu stands for partition of unity. x is the list of tabulated points.
x_ is the x-value for which the function pu is to be evaluated (value=
return value). k is the index, integer or half-integer, which indicates
what member of the partition of unity is considered. w_0, w_1 are weights
which modify the original pu in order to account for the expected error
of the local interpolation functions. This will be used if local
interpolation functions are rational and either due to poles or to
particularities of the Bulirsch Stoer algorithm not always give
contributions of equal reliability. If both weights are set equal,
no modification of the originally designed pu results. w_0=w_1=0. gives
fastest execution.
*/
{
   static R hillPar=0.810747;     // seems to be better than P=1
   Z kc,ku,kl;
   R xc,xn,y,y_;
   kc=(Z)(k+0.1);
   if( k==(R)kc ){ // then k is integer
      xc=x[kc];
      if (x_>xc){
         kc++;
         if ( kc>=n )  return 1;
      }
      else {
         kc--;
         if ( kc<0) return 1;
      }
      xn=x[kc];
   }
   else { // then in our application k is half-integer
      ku=(Z)(k+0.6); kl=ku-1;
      if ( ku>=n || kl<0) return 1;
      xc=0.5*(x[ku]+x[kl]);
      if (x_>xc){
         ku++; kl++;
         if (ku>=n) return 1;
      }
      else {
         ku--; kl--;
         if (kl<0 || ku<0 ) return 1;
      }
      xn=0.5*(x[ku]+x[kl]);
   }
   y=(x_-xc)/(xn-xc);
```

```
   if (w_0==0. && w_1==0.)  goto re;
   y_=y*w_1+(1-y)*w_0;
   if (y_==0.) goto re;
   y*=(w_1/y_);
re:   return hill(y,hillPar);
}

R lip(Vl<R> x, Vl<R> y, Z n, Z m, R x_, R k, R& acc, Z method)
/* lip stands for local interpolation. It returns the value of the local
interpolating function, indexed by k (integer or half-integer) at x_.
x,y are the global interpolation data of length n. m of them are used
for local interpolation (according to 'method' this number may change
near the boundary). acc is a measure for the 'expected error'in the
interpolation, see Press, Flannery,..p.90....
method is a parameter which determines whether polynomial or rational
interpolation is to be used. Further, it determines the way how
interpolation near the boundary is handled.
*/
{

   R y_;
   Z j,l,r,n_;
   range(k,n,m,l,r,method);
   n_=1+r-l;
   if (n_==1) return y[l]; // this is the trivial case of constant
      // interpolation
   V<R> a(n_);
   V<R> b(n_);
   Z i_c=1;
   for (j=l; j<=r; j++){
      a[i_c]=x[j]; b[i_c]=y[j]; i_c++;
   }
   if (method == 1){
      ratint(a,b,x_,y_,acc);
   }
   else {
      polint(a,b,x_,y_,acc);
   }
   return y_;
}

R lip_pos(Vl<R> x, Vl<R> y, Z n, Z m, R x_, R k, R& acc, Z method)
/* lip_pos is in every respect identical to lip with the only exception
that it interpolates the logarithms of the function (by the
method selected by parameter method) and gives back the exp of the
logarithms. Since s_log is used for the logarithm, the values are allowed
to be negative, but in this case the result is not meaningful.
*/
{
   R y_;
```

```
   Z j,l,r,n_;
   range(k,n,m,l,r,method);
   n_=1+r-l;
   if (n_==1) return y[l]; // this is the trivial case of constant
      // interpolation
   V<R> a(n_);
   V<R> b(n_);
   Z i_c=1;
   for (j=l; j<=r; j++){
      a[i_c]=x[j]; b[i_c]=s_log(y[j]); i_c++;
   }
   if (method == 1){
      ratint(a,b,x_,y_,acc);
   }
   else {
      polint(a,b,x_,y_,acc);
   }
   return s_exp(y_);
}

} // anonymus

void unique(Vl<R>& x, Vl<R>& y, R eps=1e-6)
// tests whether the list of x's is stricly increasing as expected for
// interpolation purposes. If two x values are very close together they
// get merged and the y-value replaced by the mean

{
   Z mL=4;
   static Word loc("unique(Vl<R>& x, Vl<R>& y)");
   cpmmessage(mL,loc&" started");
   Z nx=x.dim();
   Z ny=y.dim();
   cpmassert(nx==ny,loc);
   Z i;
   R lastx=x[0];
   R lasty=y[0];
   Vl<bool> select(nx,true);
   for (i=1;i<nx;i++){
      R valx=x[i];
      R valy=y[i];
      R diffx=CpmRoot::disVal<R>(valx,lastx);
      R diffy=CpmRoot::disVal<R>(valy,lasty);
      if (diffx<eps){
         select[i]=false;
         x[i-1]=0.5*(valx+lastx);
         y[i-1]=0.5*(valy+lasty);
         if (diffx>0 || diffy>0){
            cpmwarning(loc&" points merged: "&
               " diffx="&cpmwrite(diffx)&" diffy="&cpmwrite(diffy));
```

```
         }
      }
      else if(valx<lastx){
         cpmerror(loc&" valx<lastx holds for valx="&cpmwrite(valx)&
            " and lastx="&cpmwrite(lastx));
      }
      lastx=valx;
      lasty=valy;
   }
   x=x.select(select);
   y=y.select(select);
   cpmmessage(mL,loc&" done");
}

void prepIpol(V<R>& x, V<R>& y, R eps)
{
   unique(x.c(),y.c(),eps);
}

R slint(R x_, const Vl<R>& x, const Vl<R>& y, Z m, Z method)
// sli stands for smooth local interpolation
// x is the array of tabular x-values, indexed 0,1,...n-1, strictly
// increasing y is the array of tabular y-values, indexed 0,1,...n-1
// x_ is the x-value for which the interpolated value ( return value of
// the function) has to be calculated. n =1,2,3,   is the number of
// tabular points. m is the 'order of the method' (m=1,2,..., m<=n).
// (apart from working near the boundary it is the number of points
// used in interpolation). method is 0 or 2 for polynomial interpolation
// and 1 for Bulirsch Stoer rational diagonal interpolation.
// The number of points used in interpolation is m if x_ is
// more than m/2 points away from the boundary. If method=2, things are
// arranged such, that coming closer to the boundary
// reduces the number of points which are used. This is only useful
// polynomial interpolation, there better extrapolation properties are
// achieved. For universal use there is included normal (not smooth)
// linear interpolation as method=3.
{
   const R epsilon=2.e-2;        // determines from what absolut
   // inaccuracy, reported from ratint on we begin to distrust the
   // result of rational interpolation.
   // should be an argument of the function, but only used if method=1

   Z n=x.dim();
   if (n!=y.dim()){
      cpmerror("slint(): x.dim() != y.dim()");
      return 0;
   }
   if (n==0){
      cpmwarning("slint(): x.dim() = 0");
      return 0;
```

```
   }
   if (n==1) return y[0];

   if (m<1) m=1;
   if (m>n) m=n;
   R w,ip_l,ip_r,acc_l,acc_r,k_;
   Z k=lonei(x,x_);
   if (method==3){
      if (k>n-2) {
         k=n-2;
      }
      else if (k==-1){
         k=0;
      }
      else{
         ;
      }
      R p=(x_-x[k])/(x[k+1]-x[k]);
      return y[k]+(y[k+1]-y[k])*p;
   }
   if ( parity(m)==-1 ){ // then m is odd
      if (k==-1) return lip(x,y,n,m,x_,0.,acc_l,method);
      if (k==n) return lip(x,y,n,m,x_,(R)(n-1),acc_l,method);
      k_=(R)k;
   }
   else { // then m is even
      if (x_<=0.5*(x[0]+x[1]))
         return lip(x,y,n,m,x_,0.5,acc_l,method);
      if (x_>=0.5*(x[n-2]+x[n-1]))
         return lip(x,y,n,m,x_,n-1.5,acc_l,method);
      // "=" in if (...)   introduced 91-11-8
      R center=0.5*(x[k]+x[k+1]);
      if (x_<= center){
         k_=k-0.5;
      }
      else {
         k_=k+0.5;
      }
   }

   ip_l=lip(x,y,n,m,x_,k_, acc_l, method);
   ip_r=lip(x,y,n,m,x_,k_+1,acc_r,method);
   if (method==1){ // to reduce the influence of poles
      R ip_i=y[k]+(y[k+1]-y[k])*(x_-x[k])/(x[k+1]-x[k]);
         // this is a linearly interpolated value
      R w_l=pu(x,n,x_,k_,0.,0.);
      R w_r=1.-w_l;
      R err_r=cpmabs(acc_r)/epsilon;
      R err_l=cpmabs(acc_l)/epsilon;
//      R err_i=fabs(ip_l-ip_r)/epsilon;
```

```
//      w_l+=err_l; w_r+=err_r;
      R w_i=0.5*(err_l+err_r); w_i*=w_i; // seems to work well
      R norm=1./(w_l+w_i+w_r);
      w_l*=norm; w_i*=norm; w_r*=norm;
      return (w_l*ip_l+w_i*ip_i+w_r*ip_r);
   }
      // If the expected error (acc as accuracy is misleading, but will
      // not be changed) for the left function is large, this increases
      // the weight ('credibility') of the right function. The epsilon
      // mechanism is introduced such that only considerable expected
      // errors influence the weights of left and right functions.
      // Some power, or other function of the a_l, a_r might be
      // useful to form.
   w=pu(x,n,x_,k_,1.0,1.0);
   return (w*ip_l+(1-w)*ip_r);
}

R slint_pos(R x_, const Vl<R>& x, const Vl<R>& y, Z m, Z method)
/*
sli_pos is in every respect identical to sli with the only exception
that it interpolates the logarithms of the function (by the method
selected by parameter method) and gives back the exp of the logarithms.
IMPORTANT: Since s_log is used for the logarithm, the values are allowed
to be zero or negative, but in this case the result is  n o t
meaningful.
Also, the only partially sucessful provisions against the influence of
poles in case of method==1 are eliminated.
*/

{
   Z n=x.dim();
   if (n!=y.dim()){
      cpmerror("slint(): x.dim() != y.dim()");
      return 0;
   }
   if (n==0){
      cpmwarning("slint(): x.dim() = 0");
      return 0;
   }
   if (n==1){
      R res=y[0];
      if (res<0) cpmerror(
         "slint_pos(...): y.dim()=1, y[0]="&cpmwrite(res)&" <0."
         );
      return res;
   }
   R w,ip_l,ip_r,acc_l,acc_r,k_;
   Z k=lonei(x,x_);
    if (m<1) m=1;
    if (m>n) m=n;
```

```
   if (method==3){
      if (k>n-2) {
         k=n-2;
      }
      else if (k==-1){
         k=0;
      }
      R y_=s_log(y[k]), y__=s_log(y[k+1]);
      return s_exp((y_+(y__-y_)*(x_-x[k])/(x[k+1]-x[k])));
   }
   if ( parity(m)==-1 ){ // then m is odd
      if (k==-1) return lip_pos(x,y,n,m,x_,0.,acc_l,method);
      if (k==n) return lip_pos(x,y,n,m,x_,(R)(n-1),acc_l,method);
      k_=(R)k;
   }
   else { // then m is even
      if (x_<=0.5*(x[0]+x[1]))
          return lip_pos(x,y,n,m,x_,0.5,acc_l,method);
      if (x_>=0.5*(x[n-2]+x[n-1]))
         return lip_pos(x,y,n,m,x_,n-1.5,acc_l,method);
      // "=" in if (...)   introduced 91-11-8
      R center=0.5*(x[k]+x[k+1]);
      if (x_<= center){
         k_=k-0.5;
      }
      else {
         k_=k+0.5;
      }
   }
   ip_l=lip_pos(x,y,n,m,x_,k_, acc_l, method);
   ip_r=lip_pos(x,y,n,m,x_,k_+1,acc_r,method);
   w=pu(x,n,x_,k_,1.0,1.0);
   return (w*ip_l+(1-w)*ip_r);
}

// 'modern' form

R locIpol(R x_, const V<R>& x, const V<R>& y,
   IpolControl const& c)
{
   return slint(x_,x.c(),y.c(),c.get_m(),c.get_method());
}

R locIpolPos(R x_, const V<R>& x, const V<R>& y,
   IpolControl const& c)
{
   return slint_pos(x_,x.c(),y.c(),c.get_m(),c.get_method());
}

R locIpol(R ic, V<R> const& yList, IpolControl const& c)
```

```
{
   Z i,d=yList.dim();
   V<R> xList(d);
   for (i=1;i<=d;++i) xList[i]=i;
   return locIpol(ic,xList,yList,c);
}

} // namespace
```

183 cpmsr.h

```
//? cpmsr.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_SR_H_
#define CPM_SR_H_
/*
   Description: Defines a template class Set<T> of sets
       the elements of which are of type T.
      Intented as a basis for a mathematics style definition of
           associative lists.
      T is assumed to show full KAGUM functionality and
      Set<T> will show the same property

   History: Started 95-3-13
       95-3-15 transformed in a class derived from Obj
      97-9-5 transformed as to fit the CTL framework
      99-7-27 transformed as to fit the CPM framework
      02-5-24 implementation changed to STL, Test_set carried out
      2005-09-10 implementation purged from io_clone and r_clone
*/

#include <cpms.h>

namespace CpmArrays{

   using CpmRoot::Word;

template <class T>
class Sr: public S<T>{  // S with a rich interface
   // represents sets, all the elements of which are of type T.

public:

   typedef Sr<T> Type;
   typedef S<T> Base;

// constructors.

   Sr(void):S<T>(){}
         // constructor for the void set (n=0 added 95-11-24)

   Sr(const T& t):S<T>(t){}
         // constructor for the set {t}

   Sr(const S<T>& s):S<T>(s){}
      // down-cast constructor
```

```
   Sr<T> condense(void)const;
      // forms a new set obtained by shrinking to a single representative
      // element every equivalence class with respect to function equiv,
      // which itself is controlled by the static element acc

// Infrastructure functions

   virtual S<T>* clone()const{ return new Sr(*this);}

   CPM_IO
   CPM_TEST_X
   CPM_DESCRIPTORS

   static R acc;

protected:

    static bool equiv(const T& t1, const T& t2)
    { return CpmRoot::disVal<T>(t1,t2)<acc;}

};

// test functions based on set theoretical identities

 template <class T>
   void idem1(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>&s)
      { lhs=(s&s); rhs=s;}

   template <class T>
   void idem2(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s)
      { lhs=(s|s); rhs=s;}

   template <class T>
   void comm1(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>& s2)
      { lhs= (s1&s2); rhs=(s2&s1);}

   template <class T>
   void comm2(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>& s2)
      {   lhs= (s1|s2); rhs=(s2|s1);}

   template <class T>
   void assoc1(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>& s2,
    const Sr<T>& s3)
      { lhs= ((s1&s2)&s3); rhs=(s1&(s2&s3));}

   template <class T>
   void assoc2(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>& s2,
    const Sr<T>& s3)
      { lhs= ((s1|s2)|s3); rhs=(s1|(s2|s3));}
```

```
   template <class T>
   void distr1(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>& s2,
    const Sr<T>& s3)
      { lhs= (s1|(s2&s3)); rhs=((s1|s2)&(s1|s3)) ;}

   template <class T>
   void distr2(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>& s2,
    const Sr<T>& s3)
      { lhs= (s1&(s2|s3)); rhs=((s1&s2)|(s1&s3)) ;}

   template <class T>
   void diffdistr1(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>&
    s2, const Sr<T>& s3)
      { lhs= (s1-(s2|s3)); rhs=((s1-s2)&(s1-s3)) ;}

   template <class T>
   void diffdistr2(Sr<T>& lhs, Sr<T>& rhs, const Sr<T>& s1, const Sr<T>&
    s2, const Sr<T>& s3)
      { lhs= (s1-(s2&s3)); rhs=((s1-s2)|(s1-s3)) ;}

// more technical tests

   template <class T>
   int addElements(const Sr<T>& s, const T& t);
      // adding and finding an element

   template <class T>
   R writeRead(const Sr<T>& s);
      // read write test (to and from file)

// Implementation of Sr<T>

#define CPM_FOR_ALL for (i=1;i<=Base::dim();i++)

template <class T>
R Sr<T>::acc=1e-3;

template <class T>
Sr<T> Sr<T>::condense(void)const
{
   Vo<T> xr=Base::toV();
   Vo<T> x1=(static_cast< V<T> >(xr).condense(equiv)).first;
   return S<T>(x1);
}

template <class T>
Word Sr<T>::nameOf(void)const
{
   Word a="Sr<";
   Word b=CpmRoot::Name<T>()(T());
```

```
   return a&b&">";
}

template <class T>
R Sr<T>::absSqr(void)const
//
{
   R s=0;
   Z i;
   CPM_FOR_ALL{
      s+=CpmRoot::AbsSqr<T>()((*this)[i]);
   }
   return s;
}

template <class T>
R Sr<T>::absVal(void)const
//
{
   return ::sqrt(absSqr());
}

/*
// looks natural but behaves not well in comparing two sets where the
// second is a noisy copy of the first (e.g. ofter writing to file with
// limited numerical accuracy and reading back. In this case the next
// version is superior. This illustrates the point that the notion of sets
// depends strongly on the notation of equality (and vice versa).

template <class T>
R Sr<T>::disVal(const Sr<T>& s)const
{
   Sr<T> sa=(*this).condense();
   Sr<T> sb=s.condense();
   Sr<T> sc=sa^sb;

   R s1=sa.absVal();
   R s2=sb.absVal();
   R s3=sc.absVal();
   return CpmRootX::distFunc(s1,s2,s3);
}
*/

template <class T>
R Sr<T>::disVal(const Sr<T>& s)const
{
   if ( Base::n_ != s.card()) return 1;
   Vo<R> d(Base::n_);
   Z i;
   CPM_FOR_ALL d[i]=CpmRoot::DisVal<T>()((*this)[i],s[i]);
```

```
   d.order();
   return d[Base::n_]; // max of d
}

template <class T>
Sr<T> Sr<T>::ranVal(Z j)const
{
   Z i;
   Sr<T> res=*this;
   Z j0=1000;
   Z j_=(j!=0 ? CpmRoot::ranVal<Z>(j0,j) : 0);
   Z j1=j;
   bool b=true;
   V<T> xr(2*Base::n_);
   V<bool> vr(2*Base::n_);
   CPM_FOR_ALL{
      xr[i]=CpmRoot::ranVal<T>(Base::x_[i],j1);
      vr[i]=CpmRoot::ranVal<bool>(b,j1);
      j1+=j_;
      // random generator for T and B used
   }
   vr[1]=1; // that not all become false for small n
   return S<T>(xr,vr);
}

template <class T>
Sr<T> Sr<T>::testVal(Z tsz)const
{
   Z nt,i;
   nt=CpmRoot::testVal<Z>(nt,tsz);
   T tTest;
   tTest=CpmRoot::testVal<T>(tTest,tsz);
   V<T> xr(nt);
   for (i=1;i<=nt;i++){
      xr[i]=CpmRoot::ranVal<T>(tTest,i);
   }
   return S<T>(xr);
}

template <class T>
Z Sr<T>::hashVal(void)const
{
   Z i,h=0;
   h+=Base::dim();
   CPM_FOR_ALL h+=CpmRoot::hashVal<T>((*this)[i]);
   return h;
}

template <class T>
bool Sr<T>::prnOn(ostream& str)const
```

```
{
   cpmwbt;
   bool res=Base::prnOn(str);
   cpmwet;
   return res;
}

template <class T>
bool Sr<T>::scanFrom(istream& str)
{
   S<T> st;
   if (!st.scanFrom(str)) return false;
   *this=Sr<T>(st);
   return true;
}

template <class T>
Word Sr<T>::toWord()const
{
   Word res="{ ";
   for (Z i=1;i<Base::n_;i++){
      Word wi=CpmRoot::toWord<T>((*this)[i]);
      wi&=", ";
      res&=wi;
   }
   res&=CpmRoot::toWord<T>((*this)[Base::n_]);
   res&=" }";
   return res;
}

template <class T>
int addElements(const Sr<T>& s, const T& t)
   // Tests whether an element added to a set is retrieved as an element
   {
      Sr<T> s1=s;
      s1.add(t);
      return s1.hasElem(t);
   }

template <class T>
R writeRead(const Sr<T>& s)
   // Tests whether a set can be read again when written to file
   {
      Word filename=s.nameOf()+".dat";
      Word name=s.nameOf();
      ofstream out(-filename);

      if (!out) cpmerror(name,"test: output file cannot be opened");
      out<<s;
      out.close();
```

```
      Sr<T> sr;
      R res;
      ifstream in(-filename);
      if (!in) cpmerror(name, "test: input file cannot be opened");
      in>>sr;
      in.close();
      res=s.disVal(sr);
      return res;
    }
#undef CPM_FOR_ALL
} // CpmArrays

#endif
```

184 cpmstate.h

```
//? cpmstate.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_STATE_H_
#define CPM_STATE_H_
/*
   Description: Development and test of a class
      that represent quantum mechanical pure states.
*/

#include <cpmvl.h>
#include <cpmc.h>
#include <cpmtypes.h>
#include <cpminterfaces.h>
#include <cpmrfunc.h>
#include <cpmconstphys.h>
#include <cpmgraph.h>

using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::C;
using CpmGraphics::Frame;
using CpmGraphics::Graph;
using CpmLinAlg::Z2;
using namespace std;
using namespace CpmArrays;
using namespace CpmFunctions;

// interfaces as declaration macros

#define CPM_PURE_VECTOR_SPACE_INTERFACE(X,S)\
public:\
   X operator - (void)const;\
   X& operator *=(S);\
   X& operator +=(const X&);\
   X& operator -=(const X&);\
   X operator *(S)const;\
   X operator +(const X&)const;\
   X operator -(const X&)const;

#define CPM_PURE_SCALAR_INTERFACE(X,S)\
public:\
    X& operator *=(S);\
   X operator *(S)const;

#define CPM_SCALAR_PRODUCT_INTERFACE(X,S)\
public:\
```

```
   X con(void)const;\
   S operator |(const X&)const;\
   R normSquared(void)const{ return (operator|(*this)).real();}\
   R norm(void)const{ return ::sqrt(normSquared());}\
   R absVal(void)const{ return norm();}\
   R absSqr(void)const{ return normSquared();}

namespace CpmQuantumMechanics{

class QMBase{ // quantum mechnics on a linear lattice, basics

protected:

// control quantities
   static Z verbose;
      // controls messages

   static R phiStep;
      // Limit to the phase angle that a single sep of the
      // direct midpoint integrator is allowed to achieve.
      // 0.01 seems to be a good value (corresponds to 0.572 angular
      // degrees. Or is it 3.6 degrees which would better agree with
      // experience in mechanical equations)

// independent physical quantities

   static R c;
      // velocity of light in vacuum

   static R hDirac;
      //=CpmPhysics::h__PHYS;
      // My reference for using the name 'hDirac' for h-bar
      // is Landau Lifschitz: Lehrbuch der Zheoretischen Physik,
      // Band 3, Seite 20, Fußnote

   static R mass;
      //e.g. CpmPhysics::me_PHYS

   static R size;
      // length of the spatial periodicity interval

// dependent physical quantities

   static R hDiracInv;
      // 1/hDirac

   static R hPlanck;
      // hDirac*2*pi, the original of 1900

   static C ihDirac;
```

```
      // C(0,hDirac)

   static C ihDiracInv;
      // C(0,-1/hDirac)

   static R ihDiracInv2;
      // -1/(hDirac*hDirac);

// Dependent physical quantities that also depend on the number d
// of discretization-points. This number d could probably also be
// usefully a static data member of the present class. I hesitate to do
// treat it this way since this would reduce flexibility (see function
// State::resize(Z) and the requirement for d to be even in cases
// where the momentum operator is used) and would induce extended code
// revision.

   static Vl<R> positions(Z d);
      // discrete positions associated with an interval [-size/2,size/2]
      // subdivided into d congruent sub-intervals. The positions
      // returned here are the midpoints of these intervals. Notice that
      // the spacing of these mid-points is size/d.
      // Notice that this picture differs a bit from the naive one in
      // which one would consider d equidistant points where the first
      // and the last one coincide with the boundaries of the interval.
      // Then the d points would divide the interval into d-1 congruent
      // subintervals and the point spacing thus would be size/(d-1).
      // Wave functions are value distributions by nature and thus are
      // associated with intervals rather than points. Therefore we don't
      // use the naive picture.

   static Vl<R> posNor(Z d);
      // 'positions normalized'. As above with size==1

   static Vl<R> frequencies(Z d);
      // spatial frequencies associated with a linear lattice of the kind
      // described in the comment to function positions.
      // See Press et a. p. 503 between eq. (12.1.8) and (12.1.9).
      // This function requires d to be an even positive number.
      // The maximum value is the Nyquist frequency fc=d/(2*size).
      // The shortest 'representable wavelength' on the lattice is
      // 1/fc = (2*size)/d. There is a mass mc for which this smallest
      // wavelength just equals its Compton-wavelength (=hDirac/(m*c)).
      // (this actually is abuse of language; hPlank/(m*c) is the real
      // Compton wavelength (Weizel and Källen agree).
      // T psi = -(hDirac^2/(2*m)) psi"
      // If we use mc for m and the numerical difference formula for
      // step size = lattice spacing s (s=size/d), we get
      // (T psi)[i] =
      // (2*psi[i]-psi[i+1]-psi[i-1])*d^2*hDirac^2/(2*mc*size^2) =
      // (2*psi[i]-psi[i+1]-psi[i-1])*(hDirac*c/s).
```

```
      // The factor (hDirac*c/s)is a energy.
      // It is the energy uncertainty associated with a limitation
      // of any quantum system to a spatial interval of length s
      // hDirac ~ dE*dt = dE *(s/c) ==> dE  ~ hDirac*c/s.
      // In a system of units with hDirac=c=1 we thus have
      // T psi = s*(2*psi[i]-psi[i+1]-psi[i-1])/(s*s) =
      // -s*psi" = -(size/d)*psi". If our system also sets s=1
      // (discretization length as unit) then T psi = - psi".
      // Notice that masses and Compton wavelengths are inversely
      // proportional. Since we adjusted to the smallest possible
      // wave length, we also adjusted for the largest reasonable mass.
      // So all masses to be dealt with on the lattice of length size
      // and d points should be  s m a l l e r  than mc.
      //  m <= mc= d*hDirac/(2*size*c) = hDirac/(2*s*c).
      // Again, in a system of units in which s=1 (together with
      // hDirac=1,c=1) the maximum representable mass is 1/2. Using such
      // a
      // system one has T psi=-psi" for the wave function of this maximum
      // mass. This canonical result would not be obtained if we
      // had related the mass to the minimum wave length via the
      // histotically 'true' Compton wavelength formula.

public: // the following four functions need to be public; this can't
   // do any harm since they don't allow to make changes

   static R mMax(Z d);
      // hDirac*d/(2*size*c) = pMax/(2*Pi*c). This mass has a better
      // relation to the motion on our lattice (by giving the
      // Schrödinger equation the form of the most simple form
      // in natural units) than the 6.28 times larger mass
      // pMax/c. Proposal for discretization: All masses should
      // be smaller than this value. See the extended comment to
      // static Vl<R> frequencies(Z d) for this issue.

   static R pMax(Z d);
      // pMax=hPlanck*d/(2*size)= (hPlank/2)/ds (ds=size/d)
      //  = pi*hDirac/ds.
      // The momentum associated with the highest spatial frequency fc
      // which can be  represented with equi-distant positions with
      // spacing size/d. Notice that fc=frequencies(d)[d/2]=d/(2*size).

   static R vMax(Z d);
      // vMax(d)=pMax(d)/mass
      // velocity associated with the momentum pMax(d)
      // Notice that a boost operation is essentially a shift operation
      // in momentum space so that state components that get shifted
      // (formally) to values larger than the upper end of the momentum
      // range will enter the momentum range at the other end.
      // Thus applying a boost with vMax to a Gaussian wave packet (at
      // rest) will result in a wave packet with momentum components
```

```
      // arround fc and -fc. Under time evolution this packet will split
      // into two beams; for boosts with velocities close to vMax, these
      // two beams are also obvious but their relative strength varies
      // with v.

   static R eMax(Z d);
      // eMax=(2/m)*(hDirac/ds)^2, ds=(size/d); maximum kinetic energy in
      // free motion if the Hamiltonian is realized by the wellknown
      // differencing scheme for the second derivative.
      // If one writes this as p^2/(2*m), we get p=2*hDirac/ds
      // This p, which also is a kind of maximal momentum turns out to be
      // smaller than pMax as defined above. It is smaller in the way
      // 2 is smaller than pi. This is the least-surprising direction ---
      // the opposit would say that the fastest possible motion would
      // have momenta exceeding pMax. For the Hamilton operator defined
      // in momentum space (by discrete Fourier transformation) no such
      // discrepency would occur.

   static R tRef(Z d);
      // size/vMax(d)
      // time needed for traversing the distance size when travelling at
      // speed vMax(d)

   static R dtLimit(Z d);
      // time needed for traversing the distance size/d when travelling
      // at speed vMax(d)

   static R dtStabilityLimit(Z d);
      // Stability limit for the direct midpoint method
      // dtStabilityLimit*eMax=2*hDirac
      // Actually here the actual norm of the Hamilton operator should
      // stand instead of the free contributions eMax.

   static R lambdaOsc(Z d);
      // (hPlanck*d/(size^2))^2 / (2*mass)
      // Consider the Hamiltonian of a harmonic oscillator:
      // Observable p("p",d), x("x",d);
      // Observable h=p*p/(2*mass)+x*x*lambda;
      // Then, for lambda=QMBase::lambdaOsc(d), a state of maximum
      // momentum will just have the kinetic energy lambda*(size/2)^2.
      // A fictitious state having this momentum and being localized near
      // x=0, would oscillate with an amplitude size*0.5.
      // Let us write  mass = xi*mMax, then one gets
      // lambdaOsc(Z d) = ((2*Pi)^2 * c * hDirac)/(xi * size^3),
      // which is independent of d. This is not too surprising since d is
      // inherent in our parameterization of the  mass as xi*mMax.

protected:

   static C facPhi;
```

```
   static C facPsi;

   static void adjust();
      // used to update all quantities that depend on hDirac

// Some functions needed to build expressions
public:
   static void accFunc(C&, const C&);
      // z1 -> z1+=z2

   static C conjFunc(const C& );
      // z.con()

   static C negFunc(const C&);
      // -z

   static C sumFunc(const C&, const C&);
      // z1+z2

   static C diffFunc(const C&, const C&);
      // z1-z2

   static C prodFunc(const C&, const C&);
      // z1*z2

   static C prodFuncReal(const C&, const C&);
      // z1*Re(z2)

   static C prodFuncConj1(const C&, const C&);
      // z1.con()*z2

   static C prodFuncConj2(const C&, const C&);
      // z1*z2.con()

   static C divFunc(const C&, const C&);
      // z1/z2

   static R rt(Z i); //  i<0 ? 0 : ::sqrt(R(i))

   static void setUnits(Word const& w);
      // w ="SI" sets SI-Unites everything else sets "natural units"
      // with hDirac==c==1. All dependent data get set accordingly.

   static void setVerbose(Z v){ verbose=v;}
   static Z getVerbose(){ return verbose;}

   static void setSize(R s){ size=s;}
   static R getSize(){ return size;}

   static void setMass(R m){ mass=m;}
```

```
   static R getMass(){ return mass;}

   static void setPhiStep(R ps){ phiStep=ps;}
   static R getPhiStep(){ return phiStep;}

   static R get_c(){ return c;}
   static R get_hDirac(){ return hDirac;}
   static R get_hPlanck(){ return hPlanck;}
   static R get_hDiracInv(){ return hDiracInv;}
   static C get_ihDirac(){ return ihDirac;}
   static C get_ihDiracInv(){ return ihDiracInv;}
   static R get_ihDiracInv2(){ return ihDiracInv2;}

   static R step(R n1,R n2)
   { return n2==0 ? CpmRootX::hugeNumber : 0.999*phiStep*n1/n2;}
   // a building block for auto-setting the step size for the
   // direct midpoint integrator
};

class Observable;

class State: public QMBase{ // finite-dimensional QM state space
   // Finite-dimensional Quantum mechanical state space with Hilbert
   // space structure and 'periodic boundary conditions'. Notice however,
   // that a fixed value of the dimension is not part of the present
   // concept.
   // So creation and destruction operators can be defined without the
   // artificial modification deviced by Fritz Bopp and nevertheless
   // satisfy their characteristic commutation relations exactly.
   // Implementation aims at best efficiency that can achieved with Cpm.
   // Here the direct pointer operations in Vl<> are essential.
   // Without aiming at efficiency one could build on C_Vector or on
   // Va<C>.

   friend class Observable;

   Z adjust(Z i)const;
      // brings i ito the range 0<=i<dim();
      // for reasons of utmost efficiency we let indexes start with 0
      // in quantum mechanical context

   Vl<C> rep_;
      // lean and efficient array with indexes starting with 0

   Z1 n1_,n2_,n3_;
      // numbers (normally 0 that describe a layout of the rep_.dim()
      // lattice points as containing a loop. Experimental

   State(const Vl<C>& rep, Z n1, Z n2, Z n3):
      rep_(rep),n1_(n1),n2_(n2),n3_(n3){}
```

```
      // creating

   explicit State(const V<C>& v):rep_(v.c()){}

   V<C> toArray()const{ return V<C>(rep_);}

public:

   typedef State Type;
   CPM_IO
   CPM_PURE_VECTOR_SPACE_INTERFACE(Type,C)
   CPM_SCALAR_PRODUCT_INTERFACE(Type,C)
   CPM_PURE_SCALAR_INTERFACE(Type,R)
      // Hilbert space with anti-unitary operator con()
   CPM_NAM(State)

   State():rep_(){}

   explicit State(Z d):rep_(d){}
      // creates zero vector of dimension d.

   State(Z d, Z n1, Z n2, Z n3);
      // constructor that allows to define the
      // branch structure.

   State(Z d, Z i):rep_(d){ rep_[i]=C::one;normalize_();}

   State(Z d, C z):rep_(d,z){ normalize_();}
      // creates a 'd-dimensional' normalized state
      // the components of which are all the same
      // positive real multiple of z

   State(Word const& idle, const V<C>& v):rep_(v.c()){normalize_();}

   State(const R_Func& f, Z d);
      // f is a function defined on the interval [-1/2,1/2] and creates a
      // normalized 'wave function' such that f(-1/2) corresponds to
      // rep_[0] and f(1/2) to rep_[d-1].

   State(const F<R,C>& f, Z d);
      // f is a function defined on the interval [-1/2,1/2] and creates a
      // normalized 'wave function'

   State(const R_Vector& arr, Z d);
      // The array arr will be used for building a Bernstein
      // polynomial, which the will define the state

   State toMomSpc()const;
      // to momentum space (FFT)
```

```
State frmMomSpc()const;
   // from momentum Space (inverse FFT)

State shift(R s)const;
   // active transformation translates a wave function peaked at
   // 0 into one peaked at s. So for s>0 translation on
   // screen is to the right. (Was in the opposite direction till
   // 2004-10-13)

State boost(R v)const;
   // active transformation that makes a particle initially
   // at rest moving with velocity v, where v>0 means
   // motion from left to right on screen.
   // (Was in the opposite direction till 2004-10-13)

State splitBoost(R v)const
   // returns a superposition of states 'going in different
   // directions' 'beam-splitter states'
{
   State res=boost(v)+boost(-v);
   res.normalize_();
   return res;
}

C getLean(Z i)const
   { return rep_.cui(i);}
   // fast read access to components without index range check

void setLean(Z i, const C& z)
   { rep_.cui(i)=z;}
   // fast write access to matrix elements without index range check

Vl<C> getRep()const{ return rep_;}
   // fast return of the main data

C operator()(const Z& i)const{ return rep_[adjust(i)];}
   // lets a state be used as a function, which is defined on the
   // set Z

C& operator[](Z i){ return rep_[adjust(i)];}
   // Indexes start here with 0 as in rep_ for utmost efficiency.
   // standard component access, no limit on indexes since always
   // periodic continuation is enforced

C const& operator[](Z i)const{ return rep_[adjust(i)];}
   // standard component access, no limit on indexes since always
   // periodic continuation is enforced

C& cui(Z i){ return rep_.cui(i);}
   // fast and unsave index access
```

```
C const& cui(Z i)const{ return rep_.cui(i);}
   // fast and unsave index access

C const& read(Z i, Outside os)const{ return rep_.read(i,os);}
   // fast and unsave index access

Z dim(void)const{return rep_.dim();}
   //: dimension
   // returns the dimension of the state space

void setBra_(Z n1, Z n2, Z n3);
   //: set branches

void set_(R r){ rep_.set_(C(r));}
   // setting all components of *this equal to C(r,0)

void set_(C z){ rep_.set_(z);}
   // setting all components of *this equal to z

void reg_(){ Z iLast=rep_.dim()-1;
   C val=(rep_[0]+rep_[iLast])*0.5; rep_[0]=val;
   rep_[iLast]=val;}

Type resize(Z newDim)const
{ return State(rep_.resize(newDim),n1_,n2_,n3_);}
   // returns the State of dimension Max(0,newDim) which
   // is naturally defined by *this

Type reduce(Z d=32)const{ return d<dim() ? resize(d) : *this;}
   // the result never exceeds *this by dimension

Type makeReal()const;
   // returns a State with all components of *this
   // replaced vy their absolute value

Type symmetrize()const;
   // returns a state which has each component i
   // replaced by the mean of i and dim()-1-i

Type smooth(bool normalizeRes=true)const;
   // returns a smoothed  version of *this.
   // If the argument has value true, the returned state, in addition,
   // is normalized (to norm 1).
   // Smoothing is by the kernel 1/4 1/2 1/4 where periodic
   // treatment of the boundary is used

void smt_(Z n=1)
   //: smooth
   // Changes *this into a smoothed, normalized version.
```

```
   // The argument gives the number of times the basic
   // smoothing operation (kernel 1/4 1/2 1/4) is performed.
{
   for (Z i=1;i<=n;++i) rep_.fh_(C(0.25),C(0.5),C(0.25),CYCLIC);
   nor_();
}

Type operator ||(const Type&)const;
   // tensor product; member function is preferred due to more
   // transparent behavior in the presence of namespaces

Type symPrd(const Type&)const;
   // symmetrized tensor product (for bosons)
   // Normalization factor is chosen such, that a product
   // of unit vectors has a unit vector as a result.

Type antSymPrd(const Type&)const;
   // anti-symmetrized tensor product (for fermions)
   // Normalization factor is chosen such, that a product
   // of unit vectors has a unit vector as a result.

Type crt()const;
   // creator = creation operator

Type des()const;
   // destructor (not in the sense of C++) = destruction operator.
   // The basic property of crt and des is:
   //    psi.crt().des()-psi.des().crt()==psi
   // which corresponds to
   //    [a,a*]=1
   // taking into account the order of application given by
   // postfix notation.

Type phi()const
{ return (crt()+des())*facPhi;}

Type phi2()const
   // normal ordered product
{  return (crt().crt()+des().des())*0.5+des().crt();}

Type psi()const
{ return (crt()-des())*facPsi;}

Type psi2()const
   // normal ordered product
{  return (crt().crt()+des().des())*(-0.5)+des().crt();}

Type sinParHam(R2_Func const& pot, R t=0, R lambda=1.)const;
   // 'single particle Hamiltonian'
   // returns a State which results from *this by application of
```

```
   // a single-body hamiltonian with position and time dependent
   // potential pot. The potential is a functions of two variables
   // (instance of R2_Func), where the second argument is time.
   // The first argument is a function of particle position
   // and of time.
   // Particle position here refers to a coordinate system
   // in which the whole position range is [-1/2,+1/2].
   // This is the actual position range for QMBase::size=1, which thus
   // is the natural choice when the present function is to be used.
   // If the coupling constant lambda is 0, no computation of the
   // potential will be done. We then have the Hamiltonian of a
   // free particle. Since the implementation uses a finte difference
   // formula for the second derivative, the dimension of *this
   // is not restricted to powers of 2.

Type freeParHam()const;
   // Like previous function but no potential

R normalize_(R r=1);
   // returns the norm of *this and changes *this into
   // a State of norm r by multiplying with a real factor.
   // If *this was the zero State, it will be turned into
   // a 'standard State' of norm r

R nor_(R r=1){ return normalize_(r);}
   //: normalize
   // 'modern' name

R variation()const;
   // returns the curve length if *this is represented as
   // the polygon rep_[0]->rep_[1]->,...->rep_[d-1] in C
   // It is a measure for the geometrical complexity of the state

R_Vector toRealForm()const;
   // returns a 2*d vector

void fromRealForm(const R_Vector& r);
   // changes *this into the one resuling from the real vector r

Z2 pos(Z j)const;
   //: position

void show(Frame& f, Z mult_i=8, Z mult_j=4,
   Z iShift=0, Z jShift=0, bool ignorePhases=false)const;
   // one component of rep_ will be represented as a rectangle of
   // mult_i*mult_j pixels where i refers to the vertical direction
   // and j to the horizontal (as in matrices). The whole image will
   // be shifted by (iShift,jShift) with respect to the left upper
   // corner of the graphics window f. Actually the components of rep_
   // are not converted to a screen color directly but before get
```

```
   // renormalized by a real factor chosen such that the dominant
   // component of rep_ gets an absolute value of 1. If the true
   // values of the components of a state s are to be shown use
   // Graph g;
   // g.show(s.toRealForm())
   // If ignorePhases differs from 0, what is represented is the
   // probability density instead of the complex-valued wave function.

void showAsCurves(Graph& g)const;
   // auto-scaled polygon representation of real part (red), imaginary
   // part (green) and absolute value of the wave function

void show(Z d1, Z d2, Frame& f, Z mult_i=4, Z mult_j=4,
   Z iShift=0, Z jShift=0)const;
   // represents the array of states got from
   // factorize(Z d1, Z d2)

R sup()const;
   // max of absolute value of components

bool orthogonal(Vl<State>& x, R eps=1e-5)const;
   // The return value indicates whether *this is othogonal to all
   // components of x. If this value is true, x will be appended to
   // x, so that one can collect vectors into a orthogonal base.

static V<State> E_Schmidt(V<State>& x, R eps=1e-6);
   // returns an orthonormal family which spans the same space
   // as family x. Made by E. Schmidt's orthogonalization;
   // eps controles the termination of the process and helps to
   // return vectors which are made 'out of numerical noise'.
   // More specifically we assume that x.dim() is an even number 2*n
   // and that there are just n complex-linearly independent members
   // in the family x. (all 2*n are real-orthonormal in our
   // application). Thus we try to return just n vectors. As is well
   // known, E. Schmidt's procedure tends to strongly accumulate
   // round-off errors for families of more than, say, 20 members.
   // So, we do a semi-understood reordering for larger families which
   // seems to cure all deseases

static void show(Graph& g, V<State> const& x);
   // shows a family of states as curves

static R orthoTest(const V<State>& x);
   // returns a very small value if the family is orthonormal,
   // and a much larger one else

void adjPhase(R eps);
   // adjust phase such that the component of largest absolute
   // value is real and positive. Actually the component of
   // largest absolute value is not always uniquely determined
```

```
   // independent of numerical noise. Therefore we give a
   // control parameter to determine a set of components that
   // will be under consideration together with the one of
   // largest absolute value: This set consists of those components
   // c of *this such that  disVal(|c|,|cMax|)<eps ,
   // cMax: largest component.
   // From all these components under consideration, we make
   // the one with the smallest index positive.

void dePhase(R p1, R p2, Z meth);
   // experimental device to enforce collapse of a wave packet
   // meth=1 results in stochastical phase jitter with maximum phase
   // change 2*pi*p1. p2 has no influence then
   // method<=0: no action. meth: 1,2,3,4 defined, see
   // implementation in cpmstate.cpp

void dePhase_(R p){ dePhase(p,0,1);}
   // 'modern' name, since is a non-constant method

State corGrn(Z nPhi, Z nBins)const;
   //: coarse grained
   // Returns a state in which complex numbers only habe nPhi
   // different (and equi-spaced) values
   // and each |rep_[i]|^2 is an integer multiple of nBins.
   // added 2007-09-08

R_Vector probabilities()const;

State toState(bool ignorePhases)const
{ return ignorePhases ? makeReal() : *this ;}

Vl< Vl<C> > factorize(Z d1, Z d2)const;
   // returns the state as a matrix in a manner that a state
   // being a tensor product State(d1)||State(d2)
   // brings the factors into evidence

State ranVal(Z j=0)const;
   // random generator. The calling instance gives as the only
   // information its dimension

R disVal(State const& st)const
{
   R a=(*this).absVal();
   R b=st.absVal();
   R d=(*this-st).absVal();
   return CpmRoot::distFunc(a,b,d);
}

bool col_(IvZ const& ivz, R p=1);
   //: collapse
```

```
      // If called, the intended action happens with probability p.
      // The intended action is collapsing the wave function
      // to either the domain given by ivz or to the complement
      // with a probability that is consistent with the state
};

} // namespace

#endif
```

185 cpmstate.cpp

```
//? cpmstate.cpp
//? Status of work 2008-10-25.
//?
/*
   cpmstate.cpp
   History: Started 98-10-12

*/

#include <cpmobservable.h>
#include <cpmimagingtools.h>
#include <cpmfft.h>
#include <cpmgraph.h>
#include <cpmv.h>
#include <cpms.h>
#include <cpmrvector.h>

/*************** struct QMBase ****************************************/

using CpmQuantumMechanics::QMBase;
using CpmQuantumMechanics::State;
using CpmQuantumMechanics::Observable;
using CpmGraphics::Frame;
using CpmGraphics::Graph;
using CpmGraphics::Z2;
using CpmImaging::Color;
using namespace std;
using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmArrays;

using CpmFunctions::R_Func;

namespace{
   R aux_=1./sqrt(2.);
}

R QMBase::phiStep=0.025;
Z QMBase::verbose=1;

R QMBase::c=1;
R QMBase::hDirac=1;
R QMBase::hDiracInv=1./hDirac;
R QMBase::hPlanck=hDirac*2*Pi;
C QMBase::ihDirac=C(0,hDirac);
C QMBase::ihDiracInv=C(0,-hDiracInv);
R QMBase::ihDiracInv2=-hDiracInv*hDiracInv;
```

```
void QMBase::adjust()
{
   hPlanck=hDirac*2*Pi;
   hDiracInv=1./hDirac;
   ihDirac=C(0,hDirac);
   ihDiracInv=C(0,-hDiracInv);
   ihDiracInv2=-hDiracInv*hDiracInv;
}

void QMBase::setUnits(Word const& w)
{
   if (w=="SI"){
      hDirac=CpmPhysics::h__PHYS;
      c=CpmPhysics::c_PHYS;
   }
   else{
      hDirac=1;
      c=1;
   }
   adjust();
}

R QMBase::mass=1;
R QMBase::size=1;

R QMBase::mMax(Z d)
{
   return hDirac*d/(2*size*c);
}

R QMBase::pMax(Z d)
{
  return hPlanck*d/(2*size);
}

R QMBase::vMax(Z d)
{
  return hPlanck*d/(2*size*mass);
}

R QMBase::eMax(Z d)
// eMax=(2/m)*(hDirac/ds)^2, ds=(size/d);
// maximum kinetic energy in free motion.
{
   R ds=size/d;
   R y=hDirac/ds;
   return (2/mass)*y*y;
}
```

```
R QMBase::tRef(Z d)
{
   return (2*size*size*mass)/(hPlanck*d);
}

R QMBase::dtLimit(Z d)
{
   R ds=size/d;
   return 2*ds*ds*mass/hPlanck;
}

R QMBase::dtStabilityLimit(Z d)
{
   return 2*hDirac/eMax(d);
}

R QMBase::lambdaOsc(Z d)
      // (hPlanck*d/(size^2))^2 / (2*mass)
{
   R val=hPlanck*d/(size*size);
   return val*val*0.5/mass;
}

Vl<R> QMBase::positions( Z d)
{
   Z mL=3;
   Word loc( "Vl<R> positions(" );
   loc&=cpm(d);
   loc&=", ";
   loc&=(cpm(size)&")");
   CPM_MA
   cpmassert(d>1,loc);
   Vl<R> res(d);
   R delta=size/d;
   R xi=(-size+delta)*0.5;
   for (Z i=0;i<d;i++){
      res[i]=xi;
      xi+=delta;
   }
   CPM_MZ
   return res;
}

Vl<R> QMBase::posNor(Z d)
{
   Z mL=3;
   Word loc="Vl<R> posNor(Z)";
   CPM_MA
   cpmassert(d>1,loc);
   Vl<R> res(d);
```

```
   R delta=1./d;
   R xi=(delta-1)*0.5;
   for (Z i=0;i<d;i++){
      res[i]=xi;
      xi+=delta;
   }
   CPM_MZ
   return res;
}

Vl<R> QMBase::frequencies( Z d)
{
   Z mL=1;
   Word loc( "QMBase::Vl<R> frequencies(" );
   loc&=cpm(d);
   loc&=", ";
   loc&=(cpm(size)&")");
   CPM_MA
   cpmassert(d>1,loc);
   Z i,dh=d/2; // d half
   cpmassert(d==dh*2,loc);
   // assures that n is even
   R delta=size/d; // sampling spacing resulting from dividing an
     // interval of length size into d equal parts
   R fc=0.5/delta; // Nyquist frequency (see Press et al. p. 500...)
   R fSpacing=2*fc/d;
   Vl<R> res(d);
   R val=0;
   for (i=0;i<=dh;i++){
      res[i]=val;
      val+=fSpacing;
   }
   val=-fc+fSpacing;
   for (i=1;i<dh;i++){
      res[dh+i]=val;
      val+=fSpacing;
   }
   CPM_MZ
   return res;
}

R QMBase::rt(Z i){return i>=0 ? ::sqrt(R(i)) : 0.;}
void QMBase::accFunc(C& z, const C& zAdd){ z+=zAdd;}
C QMBase::conjFunc(const C& z){ return z.con();}
C QMBase::negFunc(const C& z1){  return -z1;}
C QMBase::sumFunc(const C& z1, const C& z2){ return z1+z2;}
C QMBase::diffFunc(const C& z1, const C& z2){ return z1-z2;}
C QMBase::prodFunc(const C& z1, const C& z2){ return z1*z2;}
C QMBase::prodFuncConj1(const C& z1, const C& z2){return z1.con()*z2; }
C QMBase::prodFuncConj2(const C& z1, const C& z2){ return z1*z2.con();}
```

```
C QMBase::prodFuncReal(const C& z1, const C& z2){ return z1*z2.real();}
C QMBase::divFunc(const C& z1, const C& z2){ return z1/z2;}
C QMBase::facPhi=C(aux_,0); // 1/sqrt(2)
C QMBase::facPsi=C(0,aux_); // i/sqrt(2)

/************************* class State ********************************/

// State::shift(...), is defined in cpmobservables.cpp
// since we better have observables available as tools.

State::State(const R_Func& f, Z d):rep_(d)
{
   Iv iv(-0.5,0.5);
   V<R> xList=iv.centers(d);
   for (Z i=0;i<d;i++) rep_[i]=C(f(xList[i+1]),0);
   normalize_();
}

State::State(const CpmFunctions::F<R,C>& f, Z d):rep_(d)
{
   R xL=-0.5, xR=0.5;
   V<R> xList=CpmImaging::ImagingTools::pixelCenters(xL,xR,d);
   for (Z i=0;i<d;i++) rep_[i]=f(xList[i+1]);
   normalize_();
}

State::State(const R_Vector& arr, Z d)
{
   R_Func fb=R_Func::Bernstein(arr, R2(-0.5,0.5));
   *this=State(fb,d);
}

State::State(Z d, Z n1, Z n2, Z n3):rep_(d),n1_(n1),n2_(n2),n3_(n3)
{
   Z mL=3;
   Word loc("State(Z,Z,Z,Z)");
   CPM_MA
   cpmassert(n1_>=0,loc);
   cpmassert(n2_>=0,loc);
   cpmassert(n3_>=0,loc);
   cpmassert(d>=0,loc);
   Z s=n1_+n2_+n3_;
   cpmassert(s<=d,loc);
   CPM_MZ
}

void State::setBra_(Z n1, Z n2, Z n3)
{
   Z mL=3;
   Word loc("State::setBra_(Z,Z,Z)");
```

```
   CPM_MA
   n1_=n1;
   n2_=n2;
   n3_=n3;
   cpmassert(n1_>=0,loc);
   cpmassert(n2_>=0,loc);
   cpmassert(n3_>=0,loc);
   Z d=rep_.dim();
   cpmassert(d>=0,loc);
   Z s=n1_+n2_+n3_;
   cpmassert(s<=d,loc);
   for (Z i=0;i<s;++i) rep_.cui(i)=C(0); // clean the interferometer
   CPM_MZ
}

Z State::adjust(Z i)const
{
   Z d=rep_.dim();
   while( i>=d) i-=d;
   while(i<0) i+=d;
   return i;
}

R State::normalize_(R r)
{
   Z d=rep_.dim();
   if (d==0) return 0.;

   R r0=norm();
   if (r0==0){
      State aux(d);
      aux[0]=C(r);
      *this=aux;
   }
   else{
      *this*=(r/r0);
   }
   return r0;
}

State State::makeReal()const
{
   Z d=rep_.dim();
   State res(d);
   for (Z i=0;i<d;++i)
      res.rep_.cui(i)=C(rep_.cui(i).absVal());
   return res;
}

State State::symmetrize()const
```

```
{
   Z d=rep_.dim();
   State res(d);
   for (Z i=0,ic=d-1;i<d;++i,--ic){
      res.rep_.cui(i)=(rep_.cui(i)+rep_.cui(ic))*0.5;
   }
   return res;
}

State State::smooth(bool normalizeRes)const
{
   Z d=rep_.dim();
   State res(d);
   for (Z i=0;i<d;++i){
      res.rep_.cui(i)=
         (rep_.cyc(i-1)*0.25+rep_.cui(i)*0.5+rep_.cyc(i+1)*0.25);
   }
   if (normalizeRes) res.normalize_();
   return res;
}

// fast implementation using natural iteration over pointers

State State::con(void)const
{
   return State(rep_.fa(conjFunc),n1_,n2_,n3_);
}

C State::operator |(const State& x2)const
{
   return rep_.fAcc2(QMBase::prodFuncConj1,QMBase::accFunc,x2.rep_);
}

R State::variation()const
{
   Z d=dim();
   Z d_=d-1;
   R res=0;
   for (Z i=0;i<d_;i++) res+=(rep_[i+1]-rep_[i]).absVal();
   return res;
}

State State::operator - (void)const
{
   return State(rep_.fa(negFunc),n1_,n2_,n3_);
}

State& State::operator +=(const State& x)
{
   return *this=*this+x;
```

```
}

State& State::operator -=(const State& x)
{
   return *this=*this-x;
}

State& State::operator *=(C z)
{
   rep_.fc_(prodFunc,z);
   return *this;
}

State& State::operator *=(R r)
{
   rep_.fc_(prodFuncReal,C(r));
   return *this;
}

State State::operator +(const State& x2)const
{
  return State(rep_.fe2(QMBase::sumFunc,x2.rep_),n1_,n2_,n3_);
}

State State::operator -(const State& x2)const
{
   return State(rep_.fe2(QMBase::diffFunc,x2.rep_),n1_,n2_,n3_);
}

State State::operator *(C x2)const
{
  State res(*this);
  return res*=x2;
}

State State::operator *(R x2)const
{
  State res(*this);
  return res*=x2;
}

bool State::prnOn(ostream& str)const
{
   cpmwt("State");
   Z d=rep_.dim();
   cpmp(d);
   for (Z i=0;i<d;i++) cpmp(rep_.cui(i));
   return true;
}
```

```
bool State::scanFrom(istream& str)
{
   Z d;
   cpms(d);
   rep_=Vl<C>(d);
   for (Z i=0;i<d;i++) cpms(rep_.cui(i));
   return true;
}

State State::operator ||(const State& s2)const
// beautyful symmetry of computation
{
   Z d1=dim();
   Z d2=s2.dim();
   State res(d1*d2);
   Z k=0;
   for (Z i=0;i<d1;i++){
      C vali=rep_[i];
      for (Z j=0;j<d2;j++){
         res[k]=vali*s2.getLean(j);
         k++;
      }
   }
   return res;
}

State State::sinParHam(R2_Func const& pot, R t, R lambda)const
{
   Z mL=3;
   Word loc("State::sinParHam(R2_Func,R,R)");
   CPM_MA
   R hD2=hDirac*hDirac;
   Z i,d=dim();
   R dxNorm=1./d;
   R dxInv2=d/size; // (1/dx)^2, dx=size/d
   dxInv2*=dxInv2;
   R fac=-hD2*dxInv2/(2*mass);
      // since the - is here, we have to take the second derivative and
      // n o t  its negative. So we take the coefficients directly from
      // Bronstein ... p. 768 Tabelle 7.13
   State res(d);
   Outside per=CYCLIC; // periodic
   for (i=0;i<d;++i){
      C ri=(rep_(i+1,per)+rep_(i-1,per)-rep_(i)*2.)*fac;
      res[i]=ri;
   }
   // now res=Kinetic energy operator applied to *this.
   // What follows are the potential contributions.
   if (lambda!=0){
      Vl<R> x0=posNor(d);
```

```
      for (i=0;i<d;++i){
         R xi=x0[i];
         C xit(xi,t);
         R pxit=pot(xit)*lambda;
         if (pxit!=0) res[i]+=rep_(i)*pxit;
      }
   }
   CPM_MZ
   return res;
}

/****************** old and good
State State::freeParHam()const
{
   Z mL=3;
   Word loc("State::freeParHam()");
   CPM_MA
   R hD2=hDirac*hDirac;
   Z i,d=dim();
   R dxNorm=1./d;
   R dxInv2=d/size; // (1/dx)^2, dx=size/d
   dxInv2*=dxInv2;
   R fac=-hD2*dxInv2/(2*mass);
   State res(d);
   Outside per=CYCLIC; // periodic
   for (i=0;i<d;++i){
      C ri=(rep_(i+1,per)+rep_(i-1,per)-rep_(i)*2.)*fac;
      res[i]=ri;
   }
   CPM_MZ
   return res;
}
*/

Z2 State::pos(Z i)const
// we assume 0<=i<dim()
{
// we assume n3_<=n2_ (not clear whether really needed).
// We have to shift all the points but the n_1+n_2 first ones.
// The shift is uniform for the next n3_ ones
// and uniform (but different) for all following ones
// condition for s1: n1_+n2_+s1=n_1, thus s1=-n2_
// condition for s2: n1_+n2_+n3_+s2=n_1+n_2, thus s1=-n3_
   if (n2_==0 || n3_==0) return Z2(i,1);
   Z l1=n1_+n2_, l2=l1+n3_, s1=-n2_, s2=-n3_; // +1 experiment
   if (i<l1) return Z2(i,1);
   else if (i<l2) return Z2(i+s1,2);
   else return Z2(i+s2,1);
}
```

```
State State::freeParHam()const
// trying to implement the branching logic
{
   Z mL=3;
   const R facL=3.;
      // factor in the Laplacian at the branching point.
      // 3 seems graph theoretically more canonical than 2
      // I had 2 first
   Word loc("State::freeParHam()");
   CPM_MA
   R hD2=hDirac*hDirac;
   Z i,d=dim();
   R dxNorm=1./d;
   R dxInv2=d/size; // (1/dx)^2, dx=size/d
   dxInv2*=dxInv2;
   R fac=-hD2*dxInv2/(2*mass);
   State res(d,n1_,n2_,n3_);
   Outside per=CYCLIC; // periodic
   if (n1_==0 || n2_==0){ // then no branching
      for (i=0;i<d;++i){
         C ri=rep_(i+1,per)+rep_(i-1,per)-rep_.cui(i)*2.;
         res.cui(i)=ri;
      }
      CPM_MZ
      return res*fac;
   }
// there are 5 special points:
   Z p1=n1_-1;
   Z p2=p1+n2_;
   Z p3=p2+1;
   Z p4=p2+n3_;
   Z p5=p4+1;
   for (i=0;i<p1;++i){
      C ri=rep_.cui(i+1)+rep_(i-1,per)-rep_.cui(i)*2.;
      res.cui(i)=ri;
   }
   for (i=p1+1;i<p2;++i){
      C ri=rep_.cui(i+1)+rep_.cui(i-1)-rep_.cui(i)*2.;
      res.cui(i)=ri;
   }
   for (i=p3+1;i<p4;++i){
      C ri=rep_.cui(i+1)+rep_.cui(i-1)-rep_.cui(i)*2.;
      res.cui(i)=ri;
   }
   for (i=p5+1;i<d;++i){
      C ri=rep_(i+1,per)+rep_.cui(i-1)-rep_.cui(i)*2.;
      res.cui(i)=ri;
   }
   res.cui(p1)=rep_.cui(p3)+rep_.cui(p1+1)+rep_.cui(p1-1)-rep_.cui(p1\
   )*facL;
```

```
   res.cui(p2)=rep_.cui(p2-1)+rep_.cui(p5)-rep_.cui(p2)*2.;
   if (n3_>1){
      res.cui(p3)=rep_.cui(p1)+rep_.cui(p3+1)-rep_.cui(p3)*2.;
      res.cui(p4)=rep_.cui(p5)+rep_.cui(p4-1)-rep_.cui(p4)*2.;
   }
   else{ // n3_==1 is slightly special
      res.cui(p3)=rep_.cui(p1)+rep_.cui(p5)-rep_.cui(p3)*2.;
      res.cui(p4)=rep_.cui(p5)+rep_.cui(p1)-rep_.cui(p4)*2.;
   }
   res.cui(p5)=rep_.cui(p5+1)+rep_.cui(p4)+rep_.cui(p2)-rep_.cui(p5)*facL;
   CPM_MZ
   return res*fac;
}

Vl< Vl<C> > State::factorize(Z d1, Z d2)const
{
   Z mL=3;
   Word loc("State::factorize(Z,Z)");
   cpmmessage(mL,loc&" started");
   Z i,j,d=dim();
   cpmassert(d==d1*d2,loc);
   Z k=0;
   Vl< Vl<C> > res(d1,Vl<C>(d2));
   for (i=0;i<d1;i++){
      for (j=0;j<d2;j++){
         res.cui(i)[j]=rep_[k];
         k++;
      }
   }
   cpmmessage(mL,loc&" done");
   return res;
}

State State::symPrd(const State& s2)const
{
   return ((*this)||s2 + s2||(*this))*C(0.5);
}

State State::antSymPrd(const State& s2)const
{
   return ((*this)||s2 - s2||(*this))*C(0.5);
}

State State::crt()const
{
   Z n=dim();
   Z nr=n+1;
   State res(nr);
   for (Z i=1;i<nr;++i) res[i]=rep_[i-1]*rt(i);
   return res;
```

```
}

State State::des()const
{
   Z n=dim();
   if (n==0) return *this;
   Z nr=n-1;
   State res(nr);
   for (Z i=0;i<nr;++i) res[i]=rep_[i+1]*rt(i+1);
   return res;
}

R_Vector State::toRealForm()const
{
   Z i,i2,d=dim();
   Z realDim=2*d;
   R_Vector res(realDim);
   C val;
   for (i=1;i<=d;i++){
      i2=i+d;
      val=rep_.cui(i-1);
      res[i]=val[1];
      res[i2]=val[2];
   }
   return res;
}

void State::fromRealForm(const R_Vector& r)
{
   Word loc("State::fromRealForm(R_Vector)");
   Z i,i2,realDim=r.dim();
   Z d=realDim/2;
   cpmassert(2*d==realDim,loc);
   rep_=Vl<C>(d);
   R valr,vali;
   for (i=1;i<=d;i++){
      i2=i+d;
      valr=r[i];
      vali=r[i2];
      rep_.cui(i-1)=C(valr,vali);
   }
}

State State::toMomSpc()const
{
   Z1 d(dim());
   cpmassert(d.isPowOfTwo(),"State::toMomSpc()");
   V<C> temp=toArray();
   temp=CpmFourier::fft(temp,1);
   return State(temp);
```

```
}

State State::frmMomSpc()const
{
   Z1 d(dim());
   cpmassert(d.isPowOfTwo(),"State::frmMomSpc()");
   V<C> temp=toArray();
   temp=CpmFourier::fft(temp,-1);
   return State(temp);
}

R State::sup()const
{
   Z d=dim();
   R res=rep_.cui(0).absVal();
   for (Z i=1;i<d;i++){
      R r=rep_.cui(i).absVal();
      if (r>res) res=r;
   }
   return res;
}

/*************** old and good *************************
void State::show(Frame& f, Z mult_i, Z mult_j, Z is, Z js,
   bool ignorePhases)const
// rather fast implementation
{
   R valMax=sup();
   if (ignorePhases) valMax*=valMax;
   R valMaxInv=1./valMax;
   Z i,j,d=dim();
   Z d_=d*mult_i;
   Z jShift=1+js;

   Z iState=0;
   Z iCounter=0;
   Z iEff=is;
   Color c;

   for (i=0;i<d_;i++){
      iEff++;
      if (iCounter==0){
         C val=rep_.cui(iState);
         if (ignorePhases) val=C(val.absSqr());
         val*=valMaxInv;
         c=Color(val);
      }
      iCounter++;
      if (iCounter==mult_i){
         iCounter=0; // triggers new evaluation of c in the next loop
```

```
         iState++;
      }
      for (j=0;j<mult_j;j++){
         f.put(Z2(iEff,j+jShift),c);
      }
   }
}
*/

// new version taking n1_,n2_,n3_ into account
void State::show(Frame& f, Z mult_i, Z mult_j, Z is, Z js,
   bool ignorePhases)const
{
   bool lean=(n2_==0 || n3_==0);
   Z ff=lean? 1 : 4;
   mult_j*=ff;
   R valMax=sup();
   if (ignorePhases) valMax*=valMax;
   R valMaxInv=1./valMax;
   Z i,d=dim();
   Z d_=d*mult_i;
   Z jShift=(1+js)*ff;

   Z iState=0;
   Z iCounter=0;
   Z iEff=is;
   Color c;
   Z multjh= lean ? mult_j :  mult_j/2;
   Z mi= lean ? mult_i : (n2_*mult_i)/n3_;

   for (i=0;i<d;i++){
      C val=rep_.cui(i);
      if (ignorePhases) val=C(val.absSqr());
      val*=valMaxInv;
      c=Color(val);
      Z2 z2=pos(i);
      Z sj=0,p1=z2[1],p2=z2[2];
      Z multAct;
      if (p2==1){
         multAct=mult_i;
         p1*=multAct;
      }
      else{
         multAct=mi;
         p1=n1_*mult_i+(p1-n1_)*multAct;
         sj=multjh;
      }
      Z pa=p1;
      for (Z ki=0;ki<multAct;++ki){
         for (Z kj=0;kj<multjh;++kj){
```

```
            f.put(Z2(pa,kj+sj+jShift),c);
         }
         pa++;
      }
   }
}

void State::show(Z d1, Z d2, Frame& f, Z mult_i, Z mult_j,
      Z iShift, Z jShift)const
{
   Vl< Vl<C> > vvc=factorize(d1,d2);
   for (Z i=0;i<d1;i++){
      State s(vvc[i],n1_,n2_,n3_);
      s.show(f,mult_i,mult_j,iShift,jShift);
      jShift+=mult_j;
   }
}

bool State::orthogonal(Vl<State>& x, R eps)const
      // The return value indicates whether *this is orthogonal to all
      // components of x. If this value is true, x will be appended to
      // x, so that one can collect vectors into a orthogonal base.
{
   Z i,n=x.dim();
   R ti=norm();
   for (i=0;i<n;i++){
      R xi=x[i].norm();
      if ((*this|x[i]).absVal()>eps*xi*ti ) return false;
   }
   V<State> x1(x); // notice that operator & is not defined in Vl<>
   x1&*this;
   x=x1.c();
   return true;
}

void State::adjPhase(R eps)
{
   Z mL=3;
   Word loc("State::adjPhase(R)");
   CPM_MA
   Z i,d=dim();
   Vo<R> absv(d);
   for (i=0;i<d;i++) absv[i+1]=rep_.cui(i).absVal();
   V<Z> per=absv.permutationForIncreasingOrder();
   Z iMax=per[d];
   R absMax=absv[per[d]];
   if (absMax==0) return;
   S<Z> is;
   for (i=1;i<=d;i++){
      if (CpmRoot::disVal<R>(absv[i],absMax)<=eps) is.add(i);
```

```
   }
   cpmassert(is.card()>0,loc);
   Z iMaxEff=is[1]; // sets are always ordered !
   iMaxEff--; // returning to zero-offset arrays
   C cEff=(*this)[iMaxEff];
   R r=cEff.absVal();
   cpmassert(r>0,loc);
   C phase=cEff/r;
   phase=phase.con();
   for (i=0;i<d;i++) rep_.cui(i)*=phase;
   CPM_MZ
}

V<State> State::E_Schmidt(V<State>& x, R eps)
   // returns an orthonormal family which spans the same space
   // as family x. Made by E. Schmidt's orthogonalization. Notice that
   // we do not assume that the family x be linear independent.
   // The procedure goes through family x in the order of indexing.
   // It is evident that the procedure is most stable if the states
   // are ordered in the sense of increasing complexity (structured-ness;
   // e.g. number of peaks, or total variation.
   // Built-in are features specific to the situation of
   // spectralRepresentation
{
   static const Z nCrit=8;
      // if the degeneracy of an eigenvalue is larger than this, the
      // standard orthonormalization procedure begins to become very
      // susceptible to numerical errors. We the do reordering

   Z mL=3;
   Word loc="State::E_Schmidt(V<State>,R)";
   CPM_MA
   Z d=x.dim();
   Z dh=d/2;
   cpmassert(2*dh==d,loc);
   if (dh>nCrit){
      Vo<R> var(d);
      for (Z i=1;i<=d;i++) var[i]=-x[i].variation(); // '-' important
      V<Z> per=var.permutationForIncreasingOrder();
         // due to the '-' in the definition of var, we order for
         // decreasing variation. My intuition was to do the opposite.
         // When this was observed not to work, I tried the present
         // version without any expectation for making things better.
         // they became much better, though.
       x=x.permute(per);
   }

   V<State> u(dh);
   Z iTest=1,iOK=0;
   while (iTest<=d && iOK<dh){
```

```
      State vTest=x[iTest];
      State next=vTest;
      for (Z i=1;i<=iOK;i++) next-=u[i]*(u[i]|vTest);
      iTest++;
      R rn=next.normalize_();
      if (rn>eps){
         iOK++;
         u[iOK]=next;
      }
   }
   if (iOK<dh){
      Z iMissing=dh-iOK;
      Word mes="State::E_Schmidt(...) "&cpm(iMissing);
      mes&=" states missing of "&cpm(dh);
      cpmmessage(mes);
   }
   CPM_MZ
   return u;
}

void State::show(Graph& g, V<State> const& x)
      // shows a familie of states as curves
{
   Z mL=3;
   Word loc("State::show(Graph,V<State>)");
   CPM_MA
   Z n=x.dim();
   cpmassert(n>0,"State::show(...)");
   Z d=x[1].dim();
   R_Vector xl(d);
   R_Matrix yl(n,d);
   Z i,j;
   for (j=1;j<=d;j++) xl[j]=j;
   for (i=1;i<=n;i++) for (j=1;j<=d;j++) yl[i][j]=x[i][j-1].absVal();
   g.show(xl,yl);
   CPM_MZ
}

void State::showAsCurves(Graph& g)const
{
   using namespace CpmImaging;
   using namespace CpmArrays;
   Z i,d=dim();
   V<Color> cl(3);
   cl[1]=BLACK; // color for absolute value
   cl[2]=RED;  // color for real part
   cl[3]=GREEN;   // color for imaginary part

   R_Matrix res(4,d);
   for (i=1;i<=d;i++){
```

```
      res[1][i]=i-1;
      C val=rep_[i-1];
      res[2][i]=val.absVal();
      res[3][i]=val[1];
      res[4][i]=val[2];
   }
   g.clear();
   g.addText("a State: Real part red, Imaginary part green)");
   g.show(res,cl);
}

R State::orthoTest(const V<State>& x)
{
   Z d=x.dim();
   R_Matrix sp(d);
   for (Z i=1;i<=d;i++) for (Z j=1;j<=d;j++)
      sp[i][j]=(x[i]|x[j]).absVal();
   R_Matrix unit=sp.net(1);
   R res=(sp-unit).absVal();
   if (res>1e-3) cpmdata<<sp;
   return res;
}

namespace{

   R f_dePhase(R x)
   {
      return 2*CpmRoot::Pi*(1-x*x);
   }
}

void State::dePhase(R p1, R p2, Z meth )
{
   Z d=dim(),i;
   if (meth<=0) return;
   R refAngle=CpmRoot::Pi2*p1;
   if (meth==1){ // stochastical phase jitter
      for (i=0;i<d;i++){
         R phi_i=CpmRoot::ranVal(refAngle);
         C omega_i=C::expi(phi_i);
         rep_[i]*=omega_i;
      }
   }
   else if (meth==2){
      R alpha=p2*sqrt(1./d);
      for (i=0;i<d;i++){
         C psi_i=rep_[i];
         R phi_i=CpmRoot::ranVal(refAngle);
         C omega_i=C::expi(phi_i);
         rep_[i]+=omega_i*alpha; // adding a complex vector
```

```
            // of lenght controlled by p2
      }
      R rb=normalize_();
      if (rb==0)
         cpmerror("void State::dePhase(R,R,Z): unexpected rb==0");
   }
   else if (meth==3){
      C omega=C::expi(refAngle);
      // complex number of unit modulus
      Vl<R> w(d);
      for (i=0;i<d;i++){
         C xi=omega*rep_[i];
         // each value of the wave function gets multiplied by omega
         R wi=xi.real();
         // we take the real part of the 'rotated' wave function.
         // Depending on the phase of xi, this may be swmall even
         // for large values of |xi|

         wi=pow(cpmabs(wi),p2);
         rep_[i]*=wi;
      }
      R rb=normalize_();
      if (rb==0)
         cpmerror("void State::dePhase(R,R,Z): unexpected rb==0");
   }
   else{
      ;
   }
}

R_Vector State::probabilities()const
{
   Z d=dim();
   R_Vector res(d);
   for (Z i=0;i<d;i++){
      R r=rep_[i].absVal();
      res[i+1]=r*r;
   }
   return res;
}

State State::ranVal(Z j)const
{
   Z d=dim();
   C z(1,1);
   Vr<C> vc(d,z);
   V<C> resPrel=vc.ranVal(j);
   return State(resPrel);
}
```

```
State State::boost(R v)const
{
   Z d=dim();
   State res=*this;
   Vl<R> pos=QMBase::positions(d);
   R fac=v*QMBase::mass*QMBase::hDiracInv;
   for (Z i=0;i<d;++i){
      R val=fac*pos[i];
      res.rep_[i]*=C(::cos(val),::sin(val));
   }
   return res;
}

State State::corGrn(Z nPhi, Z n)const
{
   using CpmGeo::RAD;
   R nInv=1./n;
   Z i,d=dim();
   Vl<C> r(d);
   for (i=0;i<d;++i){
      R rho,phi;
      rep_.cui(i).toPolar(rho,phi);
      Angle a(phi,RAD); // degrees are default
      a.qun_(nPhi); // quantize the angle
      R ci=rho*rho;
      ci=nInv*cpmround(ci*n);
      R ri=::sqrt(ci);
      r.cui(i)=C(ri,a,"polar");
   }
   State res(r,n1_,n2_,n3_);
   res.nor_();
   return res;
}

bool State::col_(IvZ const& ivz, R p)
{
   static bool trapped=false;
   if (trapped) return false;
   R r=CpmRoot::randomR();
   if (r>p ) return false;
   IvZ iz=rep_.dom()&ivz;
   R w=0;
   Z i;
   for (i=iz.b();i<=iz.e();++i){
      w+=rep_.cui(i).absSqr();
   }
   bool res=false;
   r=CpmRoot::randomR();
   if (r<w){ // then the ivz-components survive
      // the others get to zero, 'state is trapped'
```

```
      for (i=rep_.b();i<=rep_.e();++i){
         if (!iz.hasElm(i)) rep_.cui(i)=C(0);
      }
      nor_();
      trapped=true;
      return true;
   }
   else return false;
}
```

186 cpmstate2.h

```
//? cpmstate2.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_STATE2_H_
#define CPM_STATE2_H_
/*
   Description: Development and test of a class that represent quantum
      mechanical pure states of 2-particle systems, where each particle
      lives on a 1-dimensional toroidal space.
*/

#include <cpmstate.h>
#include <cpmtrj.h>

using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::C;
using CpmGraphics::Frame;
using CpmGraphics::Graph;
using namespace std;
using namespace CpmArrays;
using namespace CpmFunctions;

namespace CpmQuantumMechanics{

   struct State2Data{ // data that define a State2 object
      R1 alpha;
      R1 beta;
      R1 mu;
      R1 eps;
      R1 p;
      R1 p1,p2,b1,b2;
      R1 m1;
      R1 m2;
      R1 size1;
      R1 size2;
      Z1 box1;
      Z1 box2;
      bool crossway;
         // If this is true, the discrete positions of the particles are
         // arranged perpendicular to each other (in this case one may
         // interprete the state also as referring to a single particle
         // in 2-dimensional space and the distance between particles
         // replaced by the distance from the origin.
         // If crossway==false, the discrete positions are arranged on
         // the same line, with the centers of the respective intervals
         // coinciding.
```

```
      State2Data():m1(1.),m2(1.),size1(1.),size2(1.){}
         // notice R1's always initialized as 0
   };

class State2{//fixed-dimensional Quantum mechanical 2-particle state space
   // Fixed-dimensional Quantum mechanical 2-particle state space with
   // Hilbert space structure, selectable boundary conditions,
   // and parameterized potential
protected:
   Z m_,n_;
      // numbers of lattice points
   R sm_,sn_; // lattice spacing in m_-direction and n_-direction
      // respectively
   R lpm_,lpn_;
      // lpm_: lowest position in m_-direction
      // lpn_: lowest position in n_-direction
      // see function ini() for the exact definition
      // used for a fast implementation of functions x and y.
   Vl< Vl<C> > rep_;
      // lean and efficient double indexed array with indexes starting
      // with 0 the range of the first index will be mapped to the
      // x-range where particle 1 lives, and the range of the second index
      // will be mapped to the y-range, where the second particle lives
      // (See UM 2006-05-16 p. 1 of my yellow concept booklet).
      // It is the function void mark(Graph& f, Z ipolMeth=1)const;
      // which does this mapping.
   Z box1_,box2_;
      // 0: values of rep_ for out-of range indexes give as default
      // object
      // 1: values given by cyclic continuation
      // else  : values given by constant continuation

   Iv iv1_,iv2_;

   void ini_()
   {
      sm_=d_.size1/m_; sn_=d_.size2/n_;
      lpm_=(-d_.size1+sm_)*0.5; lpn_=(-d_.size2+sn_)*0.5;
      // thus lpm_=-d_.size1/2 + 0.5*sm_, i.e. one half-space above the
      // lower end of the biotope [-d_.size1/2,d_.size1/2]
      box1_=d_.box1;
      box2_=d_.box2;
      iv1_=Iv(d_.size1*(-0.5),d_.size1*(0.5));
      iv2_=Iv(d_.size2*(-0.5),d_.size2*(0.5));
   }

   static State2Data d_;
      // Thus all instances of State2 share the data collected in d_
   State2(Vl<C> const& psi1, Vl<C> const& psi2);
      // creating a tensor-product of two one-particle states
```

```
   State2(Z d1, Z d2):m_(d1),n_(d2),rep_(m_,Vl<C>(n_,C())){ini_();}
public:
   State2(Vl< Vl<C> > const& psi):rep_(psi){ini_();}
   typedef State2 Type;
   CPM_IO
   CPM_NAM(State2)
   CPM_SCALAR_PRODUCT_INTERFACE(Type,C)

   State2():m_(0),n_(0),rep_(m_,Vl<C>(n_)){ini_();}
      // lean constructor which does no allocation for components

   State2(State const& psi1, State const& psi2);

   State red1(State const& psi)const;
      // reduce first index via psi

   State red2(State const& psi)const;
      // reduce second index via psi

   Vl< Vl<C> > getRep()const{ return rep_;}

   C operator()(Z i, Z j)const{ return rep_.cui(i).cui(j);}
      // efficient but unsafe reading

   void put_(Z i, Z j, C v){ rep_.cui(i).cui(j)=v;}
   // efficient but unsafe writing

   Z dim1()const{ return m_;}
   Z dim2()const{ return n_;}
   Z d1()const{ return m_;}
   Z d2()const{ return n_;}
   Iv iv1()const{ return iv1_;}
   Iv iv2()const{ return iv2_;}

   State2 dot()const;
      // hamilton operator divided by ihDirac. Applied to a state
      // it gives the time derivative of this state. This is what
      // the name tries to suggest.

   State2 V12()const;
      // returns the state resulting from applying the
      // interaction energy operator defined by v12
      // This makes it easy to compute expectation values
      // of this interaction operator, and this helpd to
      // set the coupling constant d_.alpha meaningfully.

   R x(Z i)const{ return lpm_+i*sm_;}
      // i \in {0,...m_-1} position in R, symmetric to the origin
      // filling the interval [-d_.size1/2,+d_.size1/2]
   R y(Z j)const{ return lpn_+j*sn_;}
```

```
   // j \in {0,...n_-1} position in R, symmetric to the origin
   // filling the interval [-d_.size2/2,+d_.size2/2]
R dis(R xi, R yj)const
   // particle distance if particle 1 is at xi and particle 2 is at yj
   // Depends on wether the biotops of the particles are orthogonal
   // to each other (d_.crossway==true) or lie on the same line
   // In both cases the centers of the biotopes are assumed to
   // coincide.
{
   return d_.crossway ? R2(xi,yj).absVal() : cpmabs(xi-yj);
}

R v12(R r)const
{
   return ::exp(-d_.mu*r)*::pow(d_.eps+r,-1-d_.p);
}

R pot12(R r)const
{
   return d_.alpha==0. ? 0: d_.alpha*v12(r);
}

R pot1(R x)const
{
   R x1=iv1().x(0.25);
   return d_.b1==0 ? 0. : d_.b1*::pow(cpmabs(x-x1),d_.p1);
}

R pot2(R x)const
{
   R x2=iv2().x(0.75);
   return d_.b2==0 ? 0. : d_.b2*::pow(cpmabs(x-x2),d_.p2);
}

R V1(R r)const
{
   return d_.alpha==0. ? 0:
      d_.alpha*v12(r);
}

R V2(R r2)const{ return d_.beta==0 ? 0. : d_.beta*r2;}
   // for efficiency this function gets input of the form y^2

void mark(Graph& f, Z ipolMeth=0, R gamma=0.5, R secFac=1.1)const;
   // showing the wave function in a way that the x-axis refers to
   // particle 1 and the y-axis ( oriented vertically upward!)
   // If the last argument is >0 it triggers an automatic
   // bright-level adjustment only in the first call and then
   // holds this level so that changes in brightness of individual
   // pixels can't occur as the result of a new level adjustment.
```

```
// basic linear structure in most efficient form
   State2& operator *=(C z);
   State2& operator *=(R r);
   State2& operator +=(State2 const& s);
   State2& operator -=(State2 const& s);

   State2 operator *(C z)const
      { State2 res(*this); return res*=z;}
   State2 operator *(R r)const
      { State2 res(*this); return res*=r;}
   State2 operator +(State2 const& s )const
      { State2 res(*this); return res+=s;}
   State2 operator -(State2 const& s )const
      { State2 res(*this); return res-=s;}

   R nor_(R r=1);
      // turns *this into a vector of length r and returns the norm of
      // the original in order not to loose information. If *this was
      // the zero vector, it will not be changed
   R norm();
      // returns the length of *this
   R sup()const;
      // max of rep_[i][j].absVal()
   void smooth_();
   // applies a next neighbor local averaging operation
   // to rep_. Does not conserve the norm although the weights
   // are optimum for doing so.
   void concentrate_();
   // multiplies each component of rep_ with the square
   // of its absolute value and renormalizes to unit norm.
   // This amplifies these parts of the wave function where
   // the values are close to the maximum value.
   // No change of the phase.
   void con_();
      // complex conjugation, makes motion reversal
   void backGrn_(R level=1e-6);
      // background
      // changes the state in a new one for which to each component
      // the real level gets added. Finally normalizing
// Accessing static member d_
   static void set_d(State2Data const& d){ d_=d;}
   static const State2Data& get_d(){ return d_;}
      // reading access to internal data
   static void setAlpha(R alpha){ d_.alpha=alpha;}
   static void setBeta(R beta){ d_.beta=beta;}
   static R alpha(){ return d_.alpha;}
   static R beta(){ return d_.beta;}
   R ds1()const{ return sm_;}
      // discretization length for particle 1
```

```
   R ds2()const{ return sn_;}
      // discretization length for particle 2
   R m1()const{ return d_.m1;}
      // mass of particle 1
   R m2()const{ return d_.m2;}
      // mass of particle 2
   R dtSup(Z p=100, R tWait=0, R gamma=0.1)const;
      // upper limit for the time step according to
      // the new insights from the preparation of the
      // publication. Tries to estimate the largest
      // energy eigenvalue (better: absolute value of this)
      // Seems to work now after many frustrations. (2006-10-26)
};

class State2Trj{ // State2 trajectory
protected:
   R t_;
      // time
   R dt_;
      // time step
   R dtL_;
      // natural time step
   State2 x_;
      // Schroedinger wave function
   State2 v_;
      // time derivative of the Schroedinger wave function
   R1 jAmp1_, jAmp2_;
      // jitter amplitudes (active if >0) for jAmpi_==1 the phase of the
      // wavefunction becomes randomized completely.
      // Initialized as 0!
public:
   bool enfNor_;
      // enforce normalization
   bool autoStep_;
   R normTol_;
      // if <=0 no action. Otherwise norm conservation within
      // this tolerance will be enforced
   Z stepMet_;
      // step method (3 for the one based on acceleration)
      // 2, and 1 for simplifications of increasing level
      // of naivity
   Z revMet_;
      // reversal method (0 bad method based on re-initialization
      // 1 exact method

   void setPhaseJitter_(R a1, R a2){jAmp1_=a1; jAmp2_=a2;}

   State2Trj(State2 const& psi);
   virtual void step_();
   C enr()const{ return (x_|v_)*QMBase::get_ihDirac();}
```

```
      // energy expectation value
   C xv()const{ return (x_|v_);}
      // similar to energy expectation value
   R xx()const{ return x_.absSqr();}
   State2 x()const{ return x_;}
   State2 v()const{ return v_;}
  // State2 a()const{ return a_;}
   R t()const{ return t_;}
   R dt()const{ return dt_;}
   void set_dt_(R dt){ dt_=dt;}
   R dtLimit()const;
      // working values for dt in step_(dt) are equal or smaller
      // as this value. dtLimit=2*ds*ds*m/hPlanck for mass m,
      // discretization length ds. The reason:
      // ds/dtLimit = QMBase::vMax(n).

   void smooth_(Z n=1){
      for (Z i=0;i<n;i++) x_.smooth_();
      if (n>0){
         x_.nor_();
         v_=x_.dot();
      }
   }

   void concentrate_(){
      x_.concentrate_();
      v_=x_.dot();
   }

   void con_(){
      x_.con_();
      if (revMet_==1){ // exact method
        v_.con_();
        v_*=C(-1);
      }
      else  v_=x_.dot();
         // approximate method, only rough motion reversal
   }

   void phaseJitter_(R relAmp1=1e-3,R relAmp2=1e-3);
      // multiplies each component by exp(i*phi) with -phi0<=phi<=phi0
      // and phi0=2*pi*relAmp1; not yet final
};

class State2Trj2: public State2Trj{ // State2 trajectory 2
   TrjDMI<State> x1_,x2_;
public:
   State2Trj2(State const& x1, State const& x2);
   virtual void step_();
   void step1_(){ x1_.step_(dt_);}
```

```
   void step2_(){ x2_.step_(dt_);}
   State x1()const{ return x1_.x();}
   State x2()const{ return x2_.x();}
  // X2<State, State> dot()const;
   void mark(Graph& f, Z ipolMeth=0, R gamma=0.5, R secFac=1.1)const;
};

} // namespace

#endif
```

187 cpmstate2.cpp

```
//? cpmstate2.cpp
//? Status of work 2008-10-25.
//?
/*
   cpmstate2.cpp
   see cpmstate2.h

*/
#include <cpmstate.h>
#include <cpmstate2.h>
#include <cpmobservable.h>
#include <cpmimagingtools.h>
#include <cpmmultiimage.h>
#include <cpmfft.h>
#include <cpmgraph.h>
#include <cpmv.h>
#include <cpms.h>
#include <cpmrvector.h>
#include <cpmvectormore.h>


using CpmQuantumMechanics::QMBase;
using CpmQuantumMechanics::State;
using CpmQuantumMechanics::State2;
using CpmQuantumMechanics::State2Data;
using CpmQuantumMechanics::State2Trj;
using CpmQuantumMechanics::State2Trj2;
using CpmQuantumMechanics::Observable;


State2Data State2::d_;

/*********************** class State2 *********************************/

State2::State2(Vl<C> const& s1, Vl<C> const& s2):
   m_(s1.dim()),n_(s2.dim()),rep_(m_,Vl<C>(n_))
{
   Z i,j;
   for (i=0;i<m_;++i){
      C si=s1[i];
      for (j=0;j<n_;++j){
         C sij=si*s2[j];
         rep_.cui(i).cui(j)=sij;
      }
   }
   ini_();
}
```

```
State2::State2(State const& s1, State const& s2)
{
   Vl<C> sr1=s1.getRep();
   Vl<C> sr2=s2.getRep();
   *this=State2(sr1,sr2);
}

State2 State2::V12()const
{
   Z mL=2;
   Word loc("State2::V12()");
   CPM_MA
   Z i,j;
   State2 res(m_,n_);
   for (i=0;i<m_;++i){
      R xi=x(i);
      for (j=0;j<n_;++j){
         R yj=y(j);
         C repij=rep_.cui(i).cui(j);
         R rij=dis(xi,yj);
         res.rep_.cui(i).cui(j)=repij*v12(rij);
      }
   }
   CPM_MZ
   return res;
}

/************ old and good ***************************
State2 State2::dot()const
// In a representative run, replacing the checked indexing (using [])
// by the unchecked one (using .cui()) gave 14% speed improvement.
// Thus cui should be used whenever one is sure that the program will
// not violate index limits.
{
   Z mL=2;
   Word loc("State2::dot()");
   CPM_MA
   Z i,j;
   R hDirac=QMBase::get_hDirac();
   R m1=d_.m1;
   cpmassert(m1!=0,loc);
   R m2=d_.m2;
   cpmassert(m2!=0,loc);
   cpmassert(m_>0,loc);
   cpmassert(n_>0,loc);
   cpmassert(d_.size1>0,loc);
   cpmassert(d_.size2>0,loc);
   R h1=d_.size1/m_; // not m_-1 see cpmstate.h
      // comment to static Vl<R> positions(Z d);
```

```
   R h2=d_.size2/n_;
   R f1=hDirac/h1;
   f1*=(-f1);
   R f2=hDirac/h2;
   f2*=(-f2);
   R fac1=f1/(2*m1);
   R fac2=f2/(2*m2);
   R facs=(fac1+fac2)*(-2);
   C fac=QMBase::get_ihDiracInv();
   State2 res(m_,n_);
   Outside box1=(Outside)box1_;
   Outside box2=(Outside)box2_;
   for (i=0;i<m_;++i){
      R xi=x(i);
      for (j=0;j<n_;++j){
         R yj=y(j);
         C repij=rep_.cui(i).cui(j);
         C cij=repij*facs+
         (rep_.read(i-1,box1).read(j,box2)+rep_.read(i+1,box1).
            read(j,box2))*fac1+
         (rep_.cui(i).read(j-1,box2)+rep_.cui(i).read(j+1,box2))*fac2;
         R rij=dis(xi,yj);
         R v1=V1(rij);
         R v2=V2(yj*yj);
         C vij=repij*(v1+v2);
         cij+=vij;
         cij*=fac;
         res.rep_.cui(i).cui(j)=cij;
      }
   }
   CPM_MZ
   return res;
}
*/

/*
State2 State2::dot()const
// this version is for reflecting walls instead of toroidal topology
// works! (2007-06-02)
{
   Z mL=2;
   Word loc("State2::dot()");
   CPM_MA
   Z i,j;
   R hDirac=QMBase::get_hDirac();
   R m1=d_.m1;
   cpmassert(m1!=0,loc);
   R m2=d_.m2;
   cpmassert(m2!=0,loc);
   cpmassert(m_>0,loc);
```

```
   cpmassert(n_>0,loc);
   cpmassert(d_.size1>0,loc);
   cpmassert(d_.size2>0,loc);
   R h1=d_.size1/m_; // not m_-1 see cpmstate.h
      // comment to static Vl<R> positions(Z d);
   R h2=d_.size2/n_;
   R f1=hDirac/h1;
   f1*=(-f1);
   R f2=hDirac/h2;
   f2*=(-f2);
   R fac1=f1/(2*m1);
   R fac2=f2/(2*m2);
   R facs=(fac1+fac2)*(-2);
   C fac=QMBase::get_ihDiracInv();
   State2 res(m_,n_);
   for (i=0;i<m_;++i){
      R xi=x(i);
      for (j=0;j<n_;++j){
         R yj=y(j);
         C repij=rep_.cui(i).cui(j);
         C cij=repij*facs+
         // using operator() which continues the array with 0's (on
         // reading)
         (rep_(i-1)(j)+rep_(i+1)(j))*fac1+
         (rep_(i)(j-1)+rep_(i)(j+1))*fac2;
         R rij=dis(xi,yj);
         R v1=V1(rij);
         R v2=V2(yj*yj);
         C vij=repij*(v1+v2);
         cij+=vij;
         cij*=fac;
         res.rep_.cui(i).cui(j)=cij;
      }
   }
   CPM_MZ
   return res;
}
*/

State2 State2::dot()const
// new version with symmetric treatment of external potentials
// for 1 and 2 directions
{
   Z mL=2;
   Word loc("State2::dot()");
   CPM_MA
   Z i,j;
   R hDirac=QMBase::get_hDirac();
   R m1=d_.m1;
   cpmassert(m1!=0,loc);
```

```
   R m2=d_.m2;
   cpmassert(m2!=0,loc);
   cpmassert(m_>0,loc);
   cpmassert(n_>0,loc);
   cpmassert(d_.size1>0,loc);
   cpmassert(d_.size2>0,loc);
   R h1=d_.size1/m_; // not m_-1 see cpmstate.h
      // comment to static Vl<R> positions(Z d);
   R h2=d_.size2/n_;
   R f1=hDirac/h1;
   f1*=(-f1);
   R f2=hDirac/h2;
   f2*=(-f2);
   R fac1=f1/(2*m1);
   R fac2=f2/(2*m2);
   R facs=(fac1+fac2)*(-2);
   C fac=QMBase::get_ihDiracInv();
   State2 res(m_,n_);
   Outside box1=(Outside)box1_;
   Outside box2=(Outside)box2_;
   for (i=0;i<m_;++i){
      R xi=x(i);
      R v1=pot1(xi);
      for (j=0;j<n_;++j){
         R yj=y(j);
         C repij=rep_.cui(i).cui(j);
         C cij=repij*facs+
         (rep_.read(i-1,box1).read(j,box2)+rep_.read(i+1,box1).
            read(j,box2))*fac1+
         (rep_.cui(i).read(j-1,box2)+rep_.cui(i).read(j+1,box2))*fac2;
         R rij=dis(xi,yj);
         R v12=pot12(rij);
         R v2=pot2(yj);
         C vij=repij*(v1+v12+v2);
         cij+=vij;
         cij*=fac;
         res.rep_.cui(i).cui(j)=cij;
      }
   }
   CPM_MZ
   return res;
}

/*
State2 State2::dot()const
// this version is for reflecting walls instead of toroidal topology
// works! (2007-06-02)
{
   Z mL=2;
   Word loc("State2::dot()");
```

```
   CPM_MA
   Z i,j;
   R hDirac=QMBase::get_hDirac();
   R m1=d_.m1;
   cpmassert(m1!=0,loc);
   R m2=d_.m2;
   cpmassert(m2!=0,loc);
   cpmassert(m_>0,loc);
   cpmassert(n_>0,loc);
   cpmassert(d_.size1>0,loc);
   cpmassert(d_.size2>0,loc);
   R h1=d_.size1/m_; // not m_-1 see cpmstate.h
      // comment to static Vl<R> positions(Z d);
   R h2=d_.size2/n_;
   R f1=hDirac/h1;
   f1*=(-f1);
   R f2=hDirac/h2;
   f2*=(-f2);
   R fac1=f1/(2*m1);
   R fac2=f2/(2*m2);
   R facs=(fac1+fac2)*(-2);
   C fac=QMBase::get_ihDiracInv();
   State2 res(m_,n_);
   for (i=0;i<m_;++i){
      R xi=x(i);
      for (j=0;j<n_;++j){
         R yj=y(j);
         C repij=rep_.cui(i).cui(j);
         C cij=repij*facs+
         // using operator() which continues the array with 0's (on
         // reading)
         (rep_(i-1)(j)+rep_(i+1)(j))*fac1+
         (rep_(i)(j-1)+rep_(i)(j+1))*fac2;
         R rij=dis(xi,yj);
         R v1=V1(rij);
         R v2=V2(yj*yj);
         C vij=repij*(v1+v2);
         cij+=vij;
         cij*=fac;
         res.rep_.cui(i).cui(j)=cij;
      }
   }
   CPM_MZ
   return res;
}
*/

State2& State2::operator *=(C z)
{
   for (Z i=0;i<m_;i++){
```

```
      rep_.cui(i).fc_(QMBase::prodFunc,z);
   }
   return *this;
}

State2& State2::operator *=(R r)
{
   for (Z i=0;i<m_;i++){
      rep_.cui(i).fc_(QMBase::prodFuncReal,C(r));
   }
   return *this;
}

State2& State2::operator +=(const State2& psi)
{
   for (Z i=0;i<m_;i++){
      rep_.cui(i).ff_(QMBase::sumFunc,psi.rep_.cui(i));
   }
   return *this;
}

State2& State2::operator -=(const State2& psi)
{
   for (Z i=0;i<m_;i++){
      rep_.cui(i).ff_(QMBase::diffFunc,psi.rep_.cui(i));
   }
   return *this;
}

C State2::operator |(State2 const& psi)const
{
   C res;
   for (Z i=0;i<m_;i++){
      res+=rep_.cui(i).fAcc2(QMBase::prodFuncConj1,
         QMBase::accFunc,psi.rep_.cui(i));
   }
   return res;
}

R State2::sup()const
{
   R res=-1;
   R val;
   Z i,j;
   for (i=0;i<m_;i++){
      for (j=0;j<n_;j++){
         val=rep_.cui(i).cui(j).absVal();
         if (val>res) res=val;
      }
   }
```

```
   return res;
}

R State2::norm()
{
   return absVal();
}

R State2::nor_(R r)
{
   R res=absVal();
   if (res>0){
      R fac=1./res;
      operator *=(fac);
   }
   return res;
}

void State2::con_()
 // complex conjugation, makes motion reversal
{
   Z i,j;
   for (i=0;i<m_;i++){
      for (j=0;j<n_;j++){
         rep_.cui(i).cui(j).con_();
      }
   }
}

void State2::concentrate_()
{
   Z i,j;
   for (i=0;i<m_;i++){
      for (j=0;j<n_;j++){
         R fac=rep_.cui(i).cui(j).absSqr();
         rep_.cui(i).cui(j)*=fac;
      }
   }
   nor_(1.);
}

void State2::backGrn_(R level)
      // changes the state in a new one for which to each component
      // the real level gets added. Finally normalizing
{
   C z(level);
   Z i,j;
   for (i=0;i<m_;i++){
      for (j=0;j<n_;j++){
         rep_.cui(i).cui(j)+=z;
```

```
      }
   }
   nor_(1.);
}

State2 State2::con(void)const
{
   cpmerror("State2::con(void)const not implemented");
   return State2();
}

void State2::mark(Graph& gr, Z ipolMethod, R gamma, R secFac)const
{
   Color::setGamma(gamma);
   bool aspRatFromState=false;
   bool periodic=true;
   gr.mark(rep_,Iv(),ipolMethod,aspRatFromState,periodic,secFac);
      // fast graphics function (made 2006-07-18)
}

State State2::red1(State const& psi)const
{
   Z mL=2;
   Word loc("State2::red1(State)");
   CPM_MA
   cpmassert(psi.dim()==m_,loc);
   State res(n_);
   for (Z j=0;j<n_;++j){
      C cj;
      for (Z i=0;i<m_;++i){
         cj+=psi.cui(i)*rep_.cui(i).cui(j);
      }
      res.cui(j)=cj;
   }
   CPM_MZ
   return res;
}

State State2::red2(State const& psi)const
{
   Z mL=2;
   Word loc("State2::red2(State)");
   CPM_MA
   cpmassert(psi.dim()==n_,loc);
   State res(m_);
   for (Z i=0;i<m_;++i){
      C ci;
      for (Z j=0;j<n_;++j){
         ci+=rep_.cui(i).cui(j)*psi.cui(j);
      }
```

```
      res.cui(i)=ci;
   }
   CPM_MZ
   return res;
}

void State2::smooth_()
// does not conserve the norm although the weights are
// optimum for doing so
{
   Vl< Vl<C> > rep(rep_);
   Z i1,i2,i1p,i1m,i2p,i2m;
   R fac=1./16;
   Outside box1=(Outside)box1_;
   Outside box2=(Outside)box2_;
   for (i1=0; i1<m_; i1++){
      for (i2=0; i2<n_; i2++){
         i1p=i1+1; i1m=i1-1;
         i2p=i2+1; i2m=i2-1;
         C s0=rep.cui(i1).cui(i2);
         C s1=rep.read(i1m,box1).read(i2,box2)+
            rep.read(i1p,box1).read(i2,box2);
         C s2=rep.cui(i1).read(i2m,box2)+
            rep.cui(i1).read(i2p,box2);
         C s3=rep.read(i1p,box1).read(i2m,box2)+
            rep.read(i1m,box1).read(i2p,box2);
         C s4=rep.read(i1p,box1).read(i2p,box2)+
            rep.read(i1m,box1).read(i2m,box2);
         C res=(s0*4.0+(s1+s2)*2.0+s3+s4)*fac;
         rep_.cui(i1).cui(i2)=res;
      }
   }
}

R State2::dtSup(Z p, R tWait, R gamma)const
{
   Z mL=1;
   Word loc("State2::dtSup(Z)");
   CPM_MA
   cpmassert(p>0,loc);
   State2 psi=*this;
   psi.nor_();
   R_Vector mem(p);
   Graph gr;
   R pInv=1./p;
   for (Z i=1;i<=p;++i){
      psi=psi.dot();
      R mi=psi.nor_(); // the value is the norm!
      mem[i]=mi; // the value is the norm!
      cpmvalue("enr",mi,2);
```

```
      if (tWait>0){
         psi.mark(gr,0,gamma);
         gr.display();
         cpmwait(tWait);
      }
      if (i==1){
         R n1=mem[i];
         cpmdebug(n1);
      }
      cpmprogress("x",i*pInv,3);
   }
   R_Vector stat=CpmArrays::statistics(mem);
   R sigma=stat[3];
   R maxVal=stat[10];
   cpmdebug(maxVal);
   R epsMax=maxVal+sigma;
   cpmdebug(epsMax);
   R res=2./epsMax;
   CPM_MZ
   return res;
}

//////////////// class State2Trj ////////////////////////////////////////

State2Trj::State2Trj(State2 const& psi):
x_(psi),v_(x_.dot()),enfNor_(false),normTol_(-1),stepMet_(0),revMet_(1)
{
   dtL_=dtLimit();
}

void State2Trj::step_()
{
   R tau=0.5*dt_;
   t_+=tau;
   if (stepMet_==-2){ // not a time stepping method,
      // only to visualize iterated application of dot
      v_=x_.dot();
      x_=v_;
   }
   else if (stepMet_==-1){ // Euler
      State2 acc=x_.dot().dot();
      x_+=(v_*dt_+acc*(dt_*tau));
      v_+=(acc*dt_);
   }
   else if (stepMet_==0){
      // lean implementation of the direct midpoint method
      x_+=v_*tau;
      v_+=x_.dot().dot()*dt_;
      x_+=v_*tau;
      phaseJitter_(jAmp1_,jAmp2_);
```

```
}
else if (stepMet_==-3){
   // Askar leap-frog
   x_+=v_*tau;
   v_=x_.dot()*2-v_;
  // v_+=x_.dot().dot()*dt_;
   x_+=v_*tau;
   phaseJitter_(jAmp1_,jAmp2_);
}
else if (stepMet_==1){ // needs no v_ as data member
   v_=x_.dot();
   x_+=(v_*dt_);
}
else if (stepMet_==2 || stepMet_==3){
   State2 xm=x_+v_*tau; // mid-point state, needs v_
   if (stepMet_==3){ // normal step, uses acceleration
      // old-fashioned implementation of the direct midpoint
      // method which is stepMet_==0 in modern form
      State2 dv=xm.dot().dot()*dt_;
      State2 dx=v_*dt_+dv*tau;
      x_+=dx;
      v_+=dv;
   }
   else if (stepMet_==2){ // uses velocity only
      v_=xm.dot();
      State2 dx=v_*dt_;
      x_+=dx;
   }
}
else if (stepMet_==4){ // uses x_,v_,a_
   // method of choice for time independent Hamiltoians
        // lean implementation of the direct midpoint method
   x_+=v_*tau;
   v_+=x_.dot().dot()*dt_;
   x_+=v_*tau;
}
else cpmerror("State2Trj::step_(): undefined step method");
t_+=tau;
if (enfNor_){
   R xn=x_.nor_();
   if (xn==0.) cpmerror("zero norm in State2Trj::step_()");
   v_*=(1./xn);
}
else if (normTol_>0){
   R xn=x_.absVal();
   if (cpmabs(xn-1.)>normTol_){
      R xnInv=1./xn;
      x_*=xnInv;
      v_*=xnInv;
   }
```

```
   }
   else if (autoStep_){
      R xn=x_.absVal();
      if (xn>1.0001 || xn<0.95){ // xn>1 signals that Nyquist is not
      // satisfied
         // so this calls for action
         smooth_();
         dt_*=0.9;
         cpmmessage("dt_/dtL_="&cpm(dt_/dtL_));
      }
   }
   else ;
}

R State2Trj::dtLimit()const
{
   R ds1=x_.ds1(),ds2=x_.ds2(),m1=x_.m1(),m2=x_.m2();
   R fac=2./QMBase::get_hPlanck();
   R r1=ds1*ds1*m1;
   R r2=ds2*ds2*m2;
   return fac*CpmRootX::inf<R>(r1,r2);
}

void State2Trj::phaseJitter_(R relAmp1,R relAmp2)
      // multiplies each component by exp(i*phi) with -phi0<=phi<=phi0
      // and phi0=2*pi*relAmpl;
{
   if (relAmp1<=0. && relAmp2<=0.) return;
   Z m=x_.dim1();
   Z n=x_.dim2();
   Vl<C> phi1(m,C(1));
   Vl<C> phi2(n,C(1));
   Z i,j;
   if (relAmp1>0){
      R a1=CpmRoot::Pi*relAmp1;
      Iv iv1(-a1,a1);
      for (i=0;i<m;++i) phi1.cui(i)=C::expi(iv1.ran());
   }
   if (relAmp2>0){
      R a2=CpmRoot::Pi*relAmp2;
      Iv iv2(-a2,a2);
      for (j=0;j<n;++j) phi2.cui(j)=C::expi(iv2.ran());
   }
   for (i=0;i<m;i++){
      C ci=phi1[i];
      for (j=0;j<n;j++){
         C cij=ci*phi2[j];
         x_.put_(i,j,x_(i,j)*cij);
         v_.put_(i,j,v_(i,j)*cij);
      }
```

```
   }
}

//////////////// class State2Trj2 ////////////////////////////////////////
/*

State2Trj2::State2Trj2(State const& x1, State const& x2):
State2Trj(State2(x1,x2)),x1_(x1,TrjCon()),x2_(x2,TrjCon())
{}


void State2Trj2::step_()
{
   x_=State2(x1_,x2_);
   v_=x_.dot();
   State2Trj::step_();
   State x1m=x1_.step_();
   State x2m=x2_.step_();
   State x1=x_.red2(x2m);
   State x2=x_.red1(x1m);
   x1.nor_();
   x2.nor_();
   x1_=x1;
   x2_=x2;
}


void State2Trj2::step_()
{
   X2<State,State> dpsi=dot();
   x1_+=dpsi.c1()*dt_;
   x2_+=dpsi.c2()*dt_;
   x1_.nor_();
   x2_.nor_();
   x_=State2(x1_,x2_);
}

void State2Trj2::mark(Graph& f, Z ipolMeth, R gamma, R secFac)const
{
   Z mL=2;
   Word loc("State2Trj2::mark(Graph,Z,R,R)");
   CPM_MA
   Frames frs(f,1,2);
   Graph g1(frs(1));
   x1_.x().showAsCurves(g1);
   Graph g2(frs(2));
   x2_.showAsCurves(g2);
   cpmwait(0.5);
   CPM_MZ
}
```

```
X2<State, State> State2Trj2::dot()const
{
   Z mL=2;
   Word loc("State2Trj2::dot()");
   CPM_MA
   Z i,j;
   R hDirac=QMBase::get_hDirac();
   State2Data d_=x_.get_d();
   R m1=d_.m1;
   R m2=d_.m2;
   Z d1=x_.d1();
   Z d2=x_.d2();
   R h1=d_.size1/d1;
   R h2=d_.size2/d2;
   R f1=hDirac/h1;
   f1*=(-f1);
   R f2=hDirac/h2;
   f2*=(-f2);
   R fac1=f1/(2*m1);
   R fac2=f2/(2*m2);
   C fac=QMBase::get_ihDiracInv();

   State y1(d1);
   State y2(d2);
   Outside box1=DEFAULT;
   Outside box2=DEFAULT;

   for (i=0;i<d1;++i){
      R xi=x_.x(i);
      R v1_i=x_.pot1(xi);
      C x1_i=x1_.cui(i);
      C y1_i=fac1*(x1_.read(i-1,box1)-2*x1_i+x1_.read(i+1,box1));
      y1_i+=x1_i*v1_i;
      C sum_i;
      for (j=0;j<d2;++j){
         R yj=x_.y(j);
         R rij=x_.dis(xi,yj);
         R vij=x_.pot12(rij);
         C x2_j=x2_.cui(j);
         sum_i+=x2_j.absSqr()*vij;
      }
      sum_i*=x1_i;
      y1_i+=sum_i;
      y1.cui(i)=fac*y1_i;
   }

   for (j=0;j<d2;++j){
      R yj=x_.y(j);
```

```
      R v2_j=x_.pot2(yj);
      C x2_j=x2_.cui(j);
      C y2_j=fac2*(x2_.read(j-1,box2)-2*x2_j+x2_.read(j+1,box2));
      y2_j+=x2_j*v2_j;
      C sum_j;
      for (i=0;i<d1;++i){
         R xi=x_.x(i);
         R rij=x_.dis(xi,yj);
         R vij=x_.pot12(rij);
         C x1_i=x1_.cui(i);
         sum_j+=x1_i.absSqr()*vij;
      }
      sum_j*=x2_j;
      y2_j+=sum_j;
      y2.cui(j)=fac*y2_j;
   }
   CPM_MZ
   return X2<State,State>(y1,y2);
}
*/
```

188 cpmsys2.h

```
//? cpmsys2.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_SYS2_H_
#define CPM_SYS2_H_
/*
   Description: see sys2x.h
*/
#include <cpmpardynsys.h>
#include <cpmsys2dat.h>
#include <cpmextractor.h>
// notice that the topic is Pala2, i.e. the second 'menu item'
// of program PaLa. The dimension number is appended as _2 or
// _3

#if defined(CPM_DIMDEF2)

#define CPM_DIM CpmDim2
#define CPM_CAMERA CpmCamera2
#define CPM_RIGID_BODY CpmRigidBody2
#define CPM_PHYSICS CpmPhysics2
#define CPM_TOP CpmPala2_2

#include <cpmsys2x.h>

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS
#undef CPM_RIGID_BODY
#undef CPM_TOP

#endif

#if defined(CPM_DIMDEF3)

#define CPM_DIM CpmDim3
#define CPM_CAMERA CpmCamera3
#define CPM_RIGID_BODY CpmRigidBody3
#define CPM_PHYSICS CpmPhysics3
#define CPM_TOP CpmPala2_3

#include <cpmsys2x.h>

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS
#undef CPM_RIGID_BODY
```

```
#undef CPM_TOP

#endif

#endif // guard
```

189 cpmsys2.cpp

```
//? cpmsys2.cpp
//? Status of work 2008-10-25.
//?
// simulating variable namespace via preprocessor as a means to
// avoid code doublication by hand

#include <cpmsys2.h>

#if defined(CPM_DIMDEF2)

#define CPM_DIM CpmDim2
#define CPM_CAMERA CpmCamera2
#define CPM_PHYSICS CpmPhysics2
#define CPM_RIGID_BODY CpmRigidBody2
#define CPM_TOP CpmPala2_2

#include <cpmsys2cpp.h>
   // 'cpp file which has to be included'

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS
#undef CPM_RIGID_BODY
#undef CPM_TOP

#endif

#if defined(CPM_DIMDEF3)

#define CPM_DIM CpmDim3
#define CPM_CAMERA CpmCamera3
#define CPM_PHYSICS CpmPhysics3
#define CPM_RIGID_BODY CpmRigidBody3
#define CPM_TOP CpmPala2_3

#include <cpmsys2cpp.h>

#undef CPM_DIM
#undef CPM_CAMERA
#undef CPM_PHYSICS
#undef CPM_RIGID_BODY
#undef CPM_TOP

#endif
```

190 cpmsys2aps.h

```
//? cpmsys2aps.h
//? Status of work 2008-10-25.
//?
// There was a Sys1 first, which now is obsolete.
// Implementation of Sys2Appl is essentially the implementation of
// Sys2Appl::doTheWork(). This happens essentially in cpmsys2apscpp.h

#include <cpminifilebasapp.h>
#include <cpmsys2.h>

namespace CpmPala2_2{

   using CpmApplication::IniFileBasedApplication;

////////////////////// Sys2Appl ////////////////////////////////////

class Sys2Appl: public IniFileBasedApplication {
// Sys2 application 2D

public:
   Sys2Appl(const Word& w):IniFileBasedApplication(w)
   {
      cpmmessage("CpmPala2_2::Sys2Appl("&w&") done");
   }
   void doTheWork();
};

} // CpmPala2

namespace CpmPala2_3{

   using CpmApplication::IniFileBasedApplication;

////////////////////// Sys2Appl ////////////////////////////////////

class Sys2Appl: public IniFileBasedApplication {
// Sys2 application 3D

public:
   Sys2Appl(const Word& w):IniFileBasedApplication(w)
   {
      cpmmessage("CpmPala2_3::Sys2Appl("&w&") done");
   }
   void doTheWork();
};

} // CpmPala3
```

191 cpmsys2aps.cpp

```
//? cpmsys2aps.cpp
//? Status of work 2008-10-25.
//?
#include <cpmsys2aps.h>

namespace CpmPala2_2{
   #include <cpmsys2apscpp.h>
}

namespace CpmPala2_3{
   #include <cpmsys2apscpp.h>
}
```

192 cpmsys2apscpp.h

```
//? cpmsys2apscpp.h
//? Status of work 2008-10-25.
//?
/*
   piece of code for use in different namespace-contexts
*/

using CpmGeo::Angle;
using CpmPhysics::gEarth_PHYS;
using CpmPhysics::gEarth;

using namespace CpmSystem;
using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmApplication;
using namespace CpmProperties;
using namespace CpmFunctions;
using namespace CpmGraphics;
using namespace CpmImaging;

namespace{

   Z fAux(R const& t, R const& tSpan, R2 const& vC, R2 const& aC,
      R2 const& etaC)
   {
      cpmassert(tSpan>0,"Sys2Appl, fAux");
      R tN=t/tSpan;
      R fac=1-R_Func::hill()(tN);
         // fac starts with 0 and goes to 1
      R vI=vC.x1;
      R vF=vC.x2;

      R aI=aC.x1;
      R aF=aC.x2;

      R etaI=etaC.x1;
      R etaF=etaC.x2;

      R d_v=vF-vI;
      R d_a=aF-aI;
      R d_eta=etaF-etaI;

      SysControl::vCrit=  vI+d_v*fac;
      SysControl::aCrit=  aI+d_a*fac;
      SysControl::eta=  etaI+d_eta*fac;
      return 1; // not needed
   }
```

```
   const R rimPaLa=0.;
   const Color cPaLa(Z(150),Z(150),Z(255));
}

void Sys2Appl::doTheWork()
{
   Z mL=1;
   Word loc("Sys2Appl::doTheWork()");
   CPM_MA
   sizeAndRank();
   Z i;
   Frame fr0;
   Graph g(fr0);
   g.setWithOrigin(0);
   g.paint(cPaLa);
   g.display();
   if (size!=1){ // thus action only for defined(CPM_USE_MPI)
      // here all the files prepared in the base class should
      // get 'ornamented'
      for (i=1;i<=nInFiles();++i){
         inFiles[i]=inFiles[i].appBefExt("_"&cpm(rank));
         cpmmessage("inFiles["&cpm(i)&"]="&inFiles[i]);
      }
      for (i=3;i<=nOutFiles();++i){ // first two will be
         // used by rank 1 only (movie-files)
         outFiles[i]=outFiles[i].appBefExt("_"&cpm(rank));
         cpmmessage("outFiles["&cpm(i)&"]="&outFiles[i]);
      }
   }

   Word sec="selection";
   rch.read(sec,"cpmdbg",CpmSystem::Message::debug);
   R runTimeSpan=1e6;
      // should never come to action
   Z continue_run=0;
   sec="syscontrol";
   SysControl(rch,sec);
      // Some of SysControl's data will be overwritten soon.
   Z nSteps1,filingPeriod1,docuPeriod1,viewingPeriod1,
     stoppingPeriod,fixedStep1;
   R dtStep1;
   V<R> vsCrit(2,SysControl::vCrit),asCrit(2,SysControl::aCrit),
        etas(2,SysControl::eta);
      // should get initialized since we should not expect to
      // read this from a ini-file in all cases
   bool readFromFile1,readFromFile2;
   bool ignore1,ignore2;
   F<R,Z> f1(Z(0)),f2(Z(0));
   sec="runcontrol1";
```

```
rch.read(sec,"ignore",ignore1);
if (!ignore1){
   rch.read(sec,"readFromFile",readFromFile1);
   rch.read(sec,"dtStep",dtStep1);
   rch.read(sec,"nSteps",nSteps1);
   rch.read(sec,"filingPeriod",filingPeriod1);
   rch.read(sec,"docuPeriod",docuPeriod1);
   rch.read(sec,"viewingPeriod",viewingPeriod1);
   rch.read(sec,"stoppingPeriod",stoppingPeriod);
   rch.read(sec,"fixedStep",fixedStep1);
   rch.read(sec,"vsCrit",vsCrit,1);
   rch.read(sec,"asCrit",asCrit,1);
   rch.read(sec,"etas",etas,1);

   Z dvs=vsCrit.dim();
   cpmassert(dvs>1,loc);
   Z das=asCrit.dim();
   cpmassert(das>1,loc);
   Z detas=etas.dim();
   cpmassert(detas>1,loc);

   SysControl::fixedStep=fixedStep1;
   R t1=dtStep1*nSteps1;

   f1=F4<R,R,R2,R2,R2,Z>(t1,R2(vsCrit[1],vsCrit[2]),
      R2(asCrit[1],asCrit[2]), R2(etas[1],etas[2]))(fAux);
   // this makes SysControl::vCrit, SysControl::aCrit, and
   // SysControl::eta following a smooth sigmoid transition between
   // the initial and final values inputted here
}

Z nSteps2,filingPeriod2,docuPeriod2,viewingPeriod2;

R dtStep2;
bool storeGraphAsImage=false;
Z nOrderSteps=0;
Z facOrder=2;
V<R> ivOrder("",0.75,1);

sec="runcontrol2";
rch.read(sec,"ignore",ignore2);
if (!ignore2){
   rch.read(sec,"readFromFile",readFromFile2);
   rch.read(sec,"dtStep",dtStep2);
   rch.read(sec,"nSteps",nSteps2);
   rch.read(sec,"filingPeriod",filingPeriod2);
   rch.read(sec,"docuPeriod",docuPeriod2);
   rch.read(sec,"viewingPeriod",viewingPeriod2);
   Z ped=0;
   rch.read(sec,"storeGraphAsImage",storeGraphAsImage,ped);
```

```
      rch.read(sec,"nOrderSteps",nOrderSteps,ped);
     // needs not to be there, since initialized already
   }

   Timer tm(runTimeSpan*3600);
      // runTimeSpan is in hours, and Timer() expects seconds

   using namespace CpmAlgorithms;

   sec="cameracontrol";
   Camera cam(rch,sec,fr0);
  // cpmdata<<cam;
   Sys2 sys;
   bool earlyStop=false;
   Record rc=rch.getRecord();

   if (continue_run==2){
      ignore1=true;
      ignore2=false;
      readFromFile2=true;
   }
   else if (continue_run==1){
      ignore1=false;
      readFromFile1=true;
      // inFiles[1] is the file that will be continued
      // and the name will be given by outFiles[1] and runName
   }
   else{;}

   if (!ignore1){
      if (readFromFile1){
         cpmmessage("branch (!ignore1) and (readFromFile1) taken");
         sys=Sys2(inFiles[1],rc);
      }
      else{
         cpmmessage("branch (!ignore1) and (!readFromFile1) taken");
         sys=Sys2(rch);
      }
      if (!(continue_run==1)){
         cpmmessage("branch !(continue_run==1) taken");
         sys.set_t(0);
      }

      OFileStream ofs1(outFiles[3]);
      earlyStop=sys.run(dtStep1,nSteps1,filingPeriod1,docuPeriod1,
         viewingPeriod1,stoppingPeriod,outFiles[1],cam,tm,f1,ofs1);
      if (docuPeriod1>0){
            // then docuStep was idle
         X2<Z,Word> data1=sys.docuStep(ofs1);
         R_Matrix rShow1(data1.c1());
```

```
         IFileStream ifs1(outFiles[3]);
         rShow1.scanFromFreeFor(ifs1());
         Frame fr1;
         fr1.paint(cPaLa);
         Graph gr1(fr1,rimPaLa);
         gr1.setGridColor(Color(FAINTGRAY));
         gr1.setColorActive(Color(WHITE));
         gr1.show(rShow1,rainBow,data1.c2(),0.25);
         cpmwait(tWaitWorkScreen_);
         fr1.paint(cPaLa);
         fr1.display();
      }
      if (earlyStop){
         ignore1=true;
         ignore2=true;
      }
   } // !ignore1

   if (!ignore2){
      if (readFromFile2){
         cpmmessage("branch (!ignore2) and (readFromFile2) taken");
         if (continue_run==2){
            Record rc0;
            sys= Sys2(inFiles[2],rc0);
               // partition data taken from movie file since
               // rc0.isVoid()==true
         }
         else sys= Sys2(inFiles[2],rc);
            // partition data taken from active ini-file via rc
      }
      else if (ignore1) {
         cpmmessage("branch (!ignore2) and (ignore1) taken");
         sys= Sys2(rch);
            // then no sys was created earlier
      }
      else{
         cpmmessage("branch (!ignore2) and (else) taken");
         ;  // use sys as created earlier
      }

// preparing the main run

      if (!(continue_run==2)){
         cpmmessage("branch !(continue_run==2) taken");
         sys.set_t(0);
         sys.setVel();
      }

      SysControl(rch,"syscontrol");
      // Notice that the second run is fully controlled by "syscontrol".
```

```
// This allows flexible testing of the relative influence of
// certain interactions. On the other hand, one has to make sure
// that experimental values have to be correctly re-set for
// 'production runs' of the program.
OFileStream ofs2(outFiles[4]);
if (nOrderSteps>1){
  // Word secl("runcontrol2"); // l for local
   sec="runcontrol2";
   Z facOrder=2,repLog=0;
   R repOrderD=0,repOrderE=0;
   V<R> ivOrder("",0,1);
   cpmrhf(facOrder);
   cpmrhf(ivOrder);
   cpmrhf(repOrderD);
   cpmrhf(repOrderE);
   cpmrhf(repLog);
   Iv ivOrd(ivOrder[1],ivOrder[2]);
   R_Vector orders=sys.anlOrd(dtStep2,nSteps2,nOrderSteps,facOrder,
      getRunName(),cam,ivOrd,repLog,repOrderD,repOrderE);
}
else{
   stoppingPeriod=0;
   F<R,Z> f2(Z(0));
   earlyStop=sys.run(dtStep2,nSteps2,filingPeriod2,docuPeriod2,
   viewingPeriod2,stoppingPeriod,outFiles[2],cam,tm,f2,ofs2);
   if (docuPeriod2>0){
      // then docuStep was not idle and we have to show some
      // screen graphics
      X2<Z,Word> data2=sys.docuStep(ofs2);
   // here this adds the data for the final state which is OK
      Z dr2=data2.c1();
      Word title2=data2.c2();
      R_Matrix rShow2(dr2);

      IFileStream ifs2(outFiles[4]);
      rShow2.scanFromFreeFor(ifs2());
      Frame fr2;
      fr2.paint(cPaLa);
      Graph gr2(fr2,rimPaLa);
      gr2.setGridColor(Color(FAINTGRAY));
      gr2.setColorActive(Color(WHITE));

      bool brown=sys.getRc().getB("done","Brown");

      if (brown){
         gr2.setXY(rShow2[1],rShow2[2],1);
         gr2.draw(rShow2[1],rShow2[2],Color(BLACK));
         gr2.setText(title2);
         gr2.display();
      }
```

```
            else{
               gr2.setAutoScale(1);
               gr2.show(rShow2,rainBow,title2,0.25);
            }

            if (storeGraphAsImage){
               Word in="img_"&getRunName();
               gr2.write(in);
            }
         }
         if (earlyStop){
            ignore1=true;
            ignore2=true;
         }
      }
   } // !ignore2
   cpmwait(tWaitWorkScreen_,2);
   CPM_MZ
}
```

193 cpmsys2cpp.h

```
//? cpmsys2cpp.h
//? Status of work 2008-10-25.
//?
// This file defines the constructors of CpmTop2::Sys2 and CpmTop2::Sys3
// and thus the main functionality also of Sys2Appl

#include <cpmunits.h>
#include <cpmroot.h>
#include <cpmperm.h>
#include <cpms.h>
#include <cpmfile.h>
#include <cpmmacros.h>
#include <cpminifilebasapp.h>
#include <cpmdistribution.h>

namespace CPM_TOP{

using CPM_RIGID_BODY::PointPar;
using CPM_RIGID_BODY::PointParC;
using CPM_RIGID_BODY::PointParV;

using CPM_RIGID_BODY::ComPar;
using CPM_RIGID_BODY::ComParC;
using CPM_RIGID_BODY::ComParV;

using CPM_RIGID_BODY::RigidObject;
using CPM_CAMERA::Camera;

using CPM_DIM::Spc;
using CPM_DIM::Vec;
using CPM_DIM::AxVec;
using CPM_DIM::Group;
using CPM_DIM::RefSys;
using CPM_DIM::SubsetOfSpace;
using CPM_DIM::dimVec;
using CPM_DIM::Rdim;

using CPM_PHYSICS::Force;
using CPM_PHYSICS::ForTor;
using CPM_PHYSICS::VectorField;
using CPM_PHYSICS::ExtSys;
using CPM_PHYSICS::VecE;
using CPM_PHYSICS::VecBH;
using CPM_PHYSICS::VecG;
using CPM_PHYSICS::FieldSpace;

using CPM_PHYSICS::VibData;
```

```
using CPM_PHYSICS::PotData;
using CPM_PHYSICS::LidData;
using CPM_PHYSICS::FenData;
using CPM_PHYSICS::SphTankData;
using CPM_PHYSICS::HomFilData;
using CPM_PHYSICS::DrumData;

using CPM_PHYSICS::Pot;
using CPM_PHYSICS::Lid;
using CPM_PHYSICS::Fen;
using CPM_PHYSICS::HomFil;
using CPM_PHYSICS::Drum;

using CpmDynSys::StepControl;
using CpmDynSys::ParDynSys;

using CpmProperties::NuggetData;

using CpmGeo::Angle;
using CpmGeo::DEG;

using CpmSystem::OFile;
using CpmArrays::S;
using CpmArrays::X2;
using CpmArrays::X4;
using CpmArrays::DisR;

using CpmMPI::Cpm_com;
using CpmMPI::Com;
using CpmFunctions::F;
using CpmFunctions::F1;
using CpmFunctions::F2;
using CpmFunctions::F3;
using CpmFunctions::F4;
using CpmFunctions::F5;
using CpmApplication::dimApp;

using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmLinAlg;
using namespace CpmGraphics;
using namespace CpmAlgorithms;
using namespace CpmImaging;
using namespace CpmProperties;
using namespace CpmPhysics;

//////////////////////////////// class Sys2 ////////////////////

namespace{
```

```
// remember: typedef ParOfSpc<Sys2Dat> Partition;
   // partition with a simple rectangular box geometry.
Partition makePartition(const Record rc)
{
   Z mL=3;
   Word loc("makePartition(Record)");
   cpmmessage(mL,loc&" started");

   Word sec="partition data";
   R xL,xU,yL,yU,zL,zU,overlap1,overlap2,overlap3;
   Z nL1,nL2,nL3,nS1,nS2,nS3;
   bool isTorus1,isTorus2,isTorus3;
   rc.get(sec,"xL",xL);
   rc.get(sec,"xU",xU);
   rc.get(sec,"yL",yL);
   rc.get(sec,"yU",yU);
   rc.get(sec,"zL",zL);
   rc.get(sec,"zU",zU);
   rc.get(sec,"overlap1",overlap1);
   rc.get(sec,"overlap2",overlap2);
   rc.get(sec,"overlap3",overlap3);
   rc.get(sec,"nL1",nL1);
   rc.get(sec,"nL2",nL2);
   rc.get(sec,"nL3",nL3);
   rc.get(sec,"nS1",nS1);
   rc.get(sec,"nS2",nS2);
   rc.get(sec,"nS3",nS3);
   rc.get(sec,"isTorus1",isTorus1);
   rc.get(sec,"isTorus2",isTorus2);
   rc.get(sec,"isTorus3",isTorus3);

   Sys2Dat stdat(xL,xU,yL,yU,zL,zU);
   Partition partition(stdat,nL1,nL2,nL3,overlap1,overlap2,overlap3,
      nS1,nS2,nS3);

   if (isTorus1){
      Vec shift;
      shift[1]=xU-xL;
      Group g1(shift);
      partition.setTorus(1,g1);
   }
   if (isTorus2){
      Vec shift;
      shift[2]=yU-yL;
      Group g2(shift);
      partition.setTorus(2,g2);

   }
   if (isTorus3){
      if (dimVec==3){
```

```
         Vec shift;
         shift[3]=zU-zL;
         Group g3(shift);
         partition.setTorus(3,g3);
      }
      else{
         partition.setTorus(3,Group());
      }
   }
   cpmmessage(mL,loc&" done");
   return partition;
}

} // namespace

Sys2::Sys2(V<ParT> const& par_, Vp<ExtSys> const& ext_,
   const Partition& part_, Record const& rc, bool careful):
   Sys2Base(par_,ext_,part_),rc_(rc)
{
   Z mL=1;
   Word loc("Sys2(V<ComPar>&,...,Partition&)");
   cpmmessage(mL,loc&" started");
   const R fracIgnore=1e-2;
   const R disTol=2e-6;
   setTimesFromExtSys();
   Z nTot=par.dim();
   Z i;
   if (careful){
      V<bool> vb(nTot);
      SubsetOfSpace sX=partition.getX();
      R d1,f1,f2;
      for (i=1;i<=nTot;i++){
         Spc xi=par[i].get_x();
         sX.shiftExternalPointToSurface4(xi,d1,f1,f2);
         if (d1>0) par[i].set_x(xi);
         vb[i]=(d1<disTol);
      }
      selectParticles(vb);
      Z diff=nTot-get_np();
      if (diff!=0){
         Z diffOK=Z(fracIgnore*nTot);
         cpmwarning(loc&": "&cpm(diff)&" particles lost");
         cpmassert(diff<=diffOK,loc);
      }
      nTot=get_np();
   }
   for (i=1;i<=nTot;i++){
      par[i].cancel_id();
      par[i].set_id(i);
      Z oi=partition.ownerSimple(par[i].get_x());
```

```
      par[i].setOwner(oi);
   }
   nTot_=Z1(nTot);
   cpmmessage(mL, loc&" done, nTot="&cpm(nTot));
}

Sys2::Sys2(const ExtrSys2& extr):
Sys2Base(extr.get_vp(),extr.get_ex(),extr.get_parti())
{
   Z mL=1;
   Word loc("Sys2(Extractor)");
   cpmmessage(mL,loc&" started");
   setTimesFromExtSys();
   cpmmessage(mL, loc&" done, nTot="&cpm(nTot_));
}

X4<V<ParT>,Vp<ExtSys>,Partition,Z>
Sys2::readFromMovie(const Word& movieFile, R frac)
{
   Z mL=1;
   Word loc("Sys2::readFromMovie("&movieFile&",frac)");
   CPM_MA
   if (frac<0 || frac>1) cpmerror(loc&" frac out of range");
   ExtrSys2 extr(movieFile);
   Partition parti=extr.get_parti();
   bool tolerant=true;
   if (frac<1){
      while(extr.next(tolerant)){;} // should not create warnings
         // for reading incomplete frames at the end
         // reads all files in order to see how much of them
         // are there.
      R countAimR=extr.getCount();
      countAimR*=frac;
      Z1 countAim(countAimR);
      ExtrSys2 extr2(movieFile);
      while(extr2.next() && extr2.getCount()<countAim){;}
      cpmmessage(mL,loc&" done");
      return X4<V<ParT>,Vp<ExtSys>,Partition,Z>
         (extr2.get_vp(),extr2.get_ex(),parti,extr2.getCount());
   }
   else{
      V<ParT> vpOK;
      Vp<ExtSys> exOK;
       Z countOK=0;
      while(extr.next(tolerant)){ // should not create warnings
         // for reading incomplete frames at the end
         vpOK=extr.get_vp();
         exOK=extr.get_ex();
         countOK=extr.getCount();
         cpmmessage(mL,"frame #"&cpm(countOK)&" read");
```

```
      }
      CPM_MZ
      return X4<V<ParT>,Vp<ExtSys>,Partition,Z>(vpOK,exOK,parti,countOK);
   }
}

Sys2::Sys2(const Word& fn, const Record& rc, R frac)
{
   Z mL=1;
   Word loc("Sys2::Sys2(Word,Record,R)");
   cpmmessage(mL,"Sys2("&fn&", Record,R) started");
   if (frac<0 || frac>1) cpmerror(loc&": frac out of range");
   Partition parti;
   bool careful=!rc.isVoid();
   Z count=0;
   sizeAndRank();
   if (rank==1){
      X4<V<ParT>,Vp<ExtSys>,Partition,Z> x4=
         readFromMovie(fn,frac);
      parti=x4.c3();
      count=x4.c4();
      if (careful) parti=makePartition(rc);
         // override parti by data from rc if these are not trivial
      *this=Sys2(x4.c1(),x4.c2(),parti,rc,careful);
      sizeAndRank();
         // needed again since the previous function call changed
         // size and rank to the values carried by
         // Sys2(x4.c1(),x4.c2(),parti,rc,careful);
      Z nParti=parti.card();
      if (size!=1 && nParti!=size){
         ostringstream ost;
         ost<<"Sys2(RecordHandler,Word): number of processes does"
            <<endl;
         ost<<" match the input data for nL1,nL2,nL3 from argument rc"
            <<endl;
         ost<<"size="<<size<<endl;
         ost<<"nParti="<<nParti<<endl;
         cpmerror(Word(ost.str()));
      }
      cpmmessage(mL,"object built successfully by process of rank 1");
   }
   distributeSys2(parti);
   cpmmessage(mL,"Sys2("&fn&", Record,R) done, "&cpm(count)&
      " frames found");
}

Sys2::Sys2(void):
Sys2Base(V<ParT>(0),Vp<ExtSys>(0)){}

Sys2::Sys2(V<Word> const& typeList):
```

```
Sys2Base(V<ParT>(0),Vp<ExtSys>(typeList.dim()))
{
   Z mL=1;
   static Word loc("Sys2(V<Word>)");
   CPM_MA
   for (Z i=1;i<=ne;++i){
      Word wi=typeList[i];
      if (wi=="Lid") ext[i]=Lid();
      else if (wi=="Pot") ext[i]=Pot();
      else if (wi=="Fen") ext[i]=Fen();
      else if (wi=="SphTank") ext[i]=SphTank();
      else if (wi=="HomFil") ext[i]=HomFil();
      else if (wi=="Drum") ext[i]=Drum();
      else cpmerror(loc&": unknown type "&wi&
         " of external system found");
   }
   CPM_MZ
}

namespace{ // auxiliar functions to be used in
   // Sys2::Sys2(const RecordHandler& rch)
Spc createFunc(Z const& i, Rdim const& xL, Rdim const& xU)
// random points in a rectangular box
{
   Z j=3*i;
   Z shift=1;
   R r1=Iv(xL[1],xU[1]).ranVal(j);
   j+=shift;
   R r2=Iv(xL[2],xU[2]).ranVal(j);
   R r3=0;
   if (dimVec==3){
      j+=shift;
      r3=Iv(xL[3],xU[3]).ranVal(j);
   }
   Rdim res(r1,r2,r3);
   return Spc(res);
}
// The following (obviously problematic) version gives a strict
// correlation between the rotation angle and the position. I did
// not expect that this would be possible. Since it is not understood
// I leave the code here

/*
ParT createFuncPar(Z const& i, Rdim const& xL, Rdim const& xU,
   V<ParT> const& vp, R const& alpha)
// random particles from list vp in a rectangular box
{
   Z j=i;
   Z shift=137;
   R r1=Iv(xL[1],xU[1]).ranVal(j);
```

```
   j+=shift;
   R r2=Iv(xL[2],xU[2]).ranVal(j);
   R r3=0;
   if (dimVec==3){
      j+=shift;
      r3=Iv(xL[3],xU[3]).ranVal(j);
   }
   Rdim rd(r1,r2,r3);
   Spc p(rd);
   ParT res=vp.cyc(i);
   res.ranRot(alpha,i);
   res.set_x(p);
   return res;
}
*/

ParT createFuncPar(Z const& i, Rdim const& xL, Rdim const& xU,
   V<ParT> const& vp, V<Spc> const& vs, R const& alpha)
// particles from list vp randomly placed in a rectangular box
// at random rotational orientation
{
   cpmassert(i>0,"createFuncPar(...)");
   Z vsd=vs.dim();
   Spc p;
   if (i<=vsd){ // use the positions from the list first
      p=vs[i];
   }
   else{
      Z j=(i-vsd); // j starts with 1
      j*=dimVec; // to use new j's for the next i
      R r1=Iv(xL[1],xU[1]).ranVal(j++);
      R r2=Iv(xL[2],xU[2]).ranVal(j++);
      R r3= dimVec==3 ? r3=Iv(xL[3],xU[3]).ranVal(j++) : 0;
      Rdim rd(r1,r2,r3);
      p=Spc(rd);
   }
// for selecting the particle and its orientation, the mechanism
// for creation of p is irrelevant
   ParT res=vp.cyc(i);
   res.ranRot(alpha,i);
   res.set_x(p);
   return res;
}

bool cond1FuncPar(ParT const& par, Drum const&  b1,
   Drum const& b2, R const& d)
{
   if (b1.isActive()){
      R g=b1.gap(par);
      if (g<=d) return false;
```

```
   }
   if (b2.isActive()){
      R g=b2.gap(par);
      if (g<=d) return false;
   }
   return true;
}

bool cond2FuncPar(X2<ParT,ParT> const& pp, R const& d)
{
   ParT p1=pp.c1();
   ParT p2=pp.c2();
   R g=p1.gap(p2);
   return g<=d ? false : true;
}

bool cond2Func(X2<Spc,Spc> const& p, R const& d)
// yields 'true' if the distance of the points is larger
// than d
{
   return (p.first()-p.second()).absVal()>d;
}

bool restrictFunc(Spc const& p, R const& r)
// returns true for an acceptable position
{
   Spc ori;
   return (p-ori).absVal()>r;
}

void mc(ParT& p, bool isOsm=true) // make colorful
// unifying the rules for particle display colors
{
   if (Spc::dimension()==2){
      R q=p.get_q();
      if (q>0) p.setCol(BLUE);
      else if (q<0) p.setCol(RED);
      else{
         if (!isOsm) p.setCol(YELLOW);
      }
   }
   else p.setCol(INVISIBLE);
}

}// namespace

//////// the most important function ////////////////////////////////////

Sys2::Sys2(const RecordHandler& rch)
// this is the function for getting data from pala2.ini
```

```
// Notice that this function will be called with rch being the defining
// data source for the whole application Sys2Appl. Not all of these
// data get used here.
{
   const Z mL=1;
   Word loc("Sys2(RecordHandler)");
   CPM_MA
   Partition parti;
   sizeAndRank();
   cpmmessage(mL,"size="&cpm(size)&" rank="&cpm(rank));
   Z i;
   if (rank==1){
      Record rca=rch.getRecord(); // rc from argument
      Record rc; // notice that the native Record-member is rc_
         // the effect of writing to rc_ here would be destroyed by
         // the '*this='-statement. So we write on rc for the present.
         // rc_ can here be filled with situation dependent data
         // that may be used via getSec() in any context. Notice also
         // that there is a prize to be payed for the flexibility of
         // this data container. The compiler will not warn if we
         // try to read some data from rc which we didn't write to
         // it earlier.
      Word sec="topic under consideration";
      Word topic,documode;
      rca.get(sec,"topic",topic);
      rc.setW(sec,"topic",topic);
         // conserve this as a 'kind of data element' of *this,
         // see the *this=... statement
      if (topic=="general"){
         sec="general data";
         rca.get(sec,"documode",documode);
         rc.setW(sec,"documode",documode);
      }
      else if (topic=="osmosis"){
         sec="osmosis data";
         rca.get(sec,"documode",documode);
         rc.setW(sec,"documode",documode);
      }
      else if (topic=="spherical_tank"){
         sec="spherical_tank data";
         rca.get(sec,"documode",documode);
         rc.setW(sec,"documode",documode);
      }
      else if (topic=="mill"){
         sec="mill data";
         rca.get(sec,"documode",documode);
         rc.setW(sec,"documode",documode);
      }
      else{
         cpmerror("no valid topic selected");
```

```
      }

      if (documode=="equilibrium"){
         R tDisplay;
         rca.get("equilibrium control","tDisplay",tDisplay);
         rc.setR("equilibrium control","tDisplay",tDisplay);
      }

      sec="runcontrol2";
      V<R> v0;
      rca.get(sec,"v0",v0);
      rc.setRs(sec,"v0",v0);
      R vAngleRange;
      rca.get(sec,"vAngleRange",vAngleRange);
      rc.setR(sec,"vAngleRange",vAngleRange);
      R dtStep;
      Z nSteps;
      rca.get(sec,"dtStep",dtStep);
      rca.get(sec,"nSteps",nSteps);
         // needs not to be there
      R tTot=nSteps*dtStep;
      parti=makePartition(rca);
         // assumes that there is a section 'partition data' to
         // read from
      Sys2Dat pDat=parti.getDat();
      Rdim psL=pDat.corL();
      Rdim psU=pDat.corU();
      R hPot=psU[2]-psL[2];
      Iv iv1=pDat.iv1();
      Iv iv2=pDat.iv2();
      Iv iv3=pDat.iv3();
// the fill-in space describes a volume where the particles get
// placed initially. It is assumed that this region is a box (brick)
// which is contained in the box of the partition.
      sec="fill-in space";
      R xL,xU,yL,yU,zL,zU,frac;
      Z nTot;
      rca.get(sec,"xL",xL);
      rca.get(sec,"xU",xU);
      rca.get(sec,"yL",yL);
      rca.get(sec,"yU",yU);
      rca.get(sec,"zL",zL);
      rca.get(sec,"zU",zU);
      rca.get(sec,"nTot",nTot);
      rca.get(sec,"frac",frac);
      cpmassert(xL>=iv1.inf(),loc);
      cpmassert(xU<=iv1.sup(),loc);
      cpmassert(yL>=iv2.inf(),loc);
      cpmassert(yU<=iv2.sup(),loc);
      cpmassert(zL>=iv3.inf(),loc);
```

```
      cpmassert(zU<=iv3.sup(),loc);

      Rdim fsL(xL,yL,zL);
      Rdim fsU(xU,yU,zU);
         // fs stands for fill-in space
         // the box should be a subset of the area for which the
         // partition is defined (let us refer to this rectangular box
         // as the 'master box'.
      Rdim svd=psU-psL;
      R rPot=0.5*svd.absVal();
         // radius of a bounding sphere of the 'master box'
      RigidObject ro(rPot);
      ExtSys es(ro);

      sec="homogeneous fields";
      V<R> vE,vB,vG;
      rca.get(sec,"E",vE);
      rca.get(sec,"B",vB);
      rca.get(sec,"G",vG);

      VecE fE(vE);
      VecBH fBH(vB,"B"); // understand Tesla units
      VecG fG(vG);

      HomFilData hfd(fE,fBH,fG); // homogeneous field data
      HomFil hf(es,hfd);

      sec="pot data";
      BodyProperties bpPot(rch,sec);

      Body pb(es,bpPot);
      PotData pd(svd);
      bool sideWalls=true;
      rca.get(sec,"sideWalls",sideWalls,0); // needs not to be there
      pd.setSideWalls(sideWalls);
// new vibration data
      VibData vd(rca,"pot vibration data");
      Pot pot(pb,pd,vd);
         // construction is based on svd and thus symmetrical to the
         // origin. This may not always be wanted; so lets shift it in a
         // position given by psU and psL and not only by their
         // difference.
      Spc cU(psU);
      Spc cL(psL);
      Spc cC=cL.cen(cU); // center = midpoint
      Spc ori;
      Vec shPot=cC-ori;
      pot*=Group(shPot); // how nicely geometry works !!
      V<ParT> vc;
      Vp<ExtSys> ext;
```

```
 // these two objects have still to be defined in order to make the
 // 'final' statement meaningful: *this=Sys2(vc,ext,parti,rc);
 // If topic = general, we need a Pot (which has vertical walls
 // and a bottom plane), and an upper Lid, and no other external
 // systems. Homogeneous fields are part of the Pot.

 sec="lid data";
 BodyProperties bpLid(rch,sec);
 Body lb(es,bpLid);
 LidData ldU(iv1,iv3,iv2.sup());
 VibData lvd(rca,"lid vibration data");
 Lid lidU(lb,ldU,lvd);

 sec="particle data";
    // data for a single particle type
 Z stochasticParticles=1;
 Z ped=1;
 R rotAng1=0;
 rca.get(sec,"rotAng1",rotAng1,ped);
 rca.get(sec,"stochasticParticles",stochasticParticles,ped);
 Angle ang1(rotAng1,DEG);
 AxVec ax1(ang1);
 Group g1(ax1);
 V<ParT> vpp(stochasticParticles);
 for (i=1;i<=stochasticParticles;++i){
    vpp[i]=ParT(rch,sec,i);
    vpp[i]*=g1;
       // for each new i, a new random selection
 }
 Z dimSpc=Spc::dimension();
 ParT pp=vpp[1];
 // the'master particle'
 R rP=pp.get_r();
 R mP=pp.get_m();
 cpmdata<<endl<<"The radius of the master particle is "<<rP<<endl;
 cpmdata<<endl<<"The mass of the master particle is "<<mP<<endl;
// if (dimSpc==3) pp.setCol(INVISIBLE);
// else pp.setCol(YELLOW);
 R dMin=2*rP*frac;
 if (topic=="spherical_tank"){
    Rdim fsd=fsU-fsL;
    R rTank=0.5*fsd.absVal();
    R facMag=0;
    ped=2;
    rca.get("spherical_tank data","facMag",facMag,ped);
    rTank*=facMag;
    cpmdata<<endl<<"The radius of the spherical tank is "
       <<rTank<<endl;
    SphTankData sphtankdata(rTank);
    SphTank sphtank(pb,sphtankdata);
```

```
      ext=Vp<ExtSys>(2);
      ext[1]=hf;
      ext[2]=sphtank;
      bool unipolarCharge=false;
      rca.get(sec,"unipolarCharge",unipolarCharge,ped);
      F<Spc,bool> c1(true);
      F<Z,Spc> cr=F2<Z,Rdim,Rdim,Spc>(fsL,fsU)(createFunc);
      F<X2<Spc,Spc>,bool> c2=F1<X2<Spc,Spc>,R,bool>(dMin)(cond2Func);
      Place<Spc> place(cr,c1,c2);
      V<Spc> xp=place(nTot);
      Z nAll=xp.dim();
      vc=V<ParT>(nAll);
      for (i=1;i<=nAll;i++){
         vc[i]=vpp.cyc(i);
         vc[i].set_x(xp[i]);
         if (!unipolarCharge){
            if (i%2){
               vc[i].invertCharge();
               vc[i].removeMagDipole();
            }
         }
         mc(vc[i]);
      }
   }
   else if (topic=="mill"){
      // most functional and mature part, needs not to deal with
      // mills only. We have two drums, a hollow one and a massive
      // one.
      sec="mill data";
      R minimumGap=SysControl::singRange;
      R rotAngleRange=0;
      rca.get(sec,"minimumGap",minimumGap,ped);
      rca.get(sec,"rotAngleRange",rotAngleRange,ped);

      sec="drum1 data";
      ped=1;
      DrumData drumData1(rch,sec,ped);
      VibData dvd1(rca,"drum1 vibration data");
      Drum drum1(pb,drumData1,dvd1);
         // notice that the material data come from pb and thus
         // from section "pot data"
      sec="drum1 shift";
      Vec shift1;
      rca.get(sec,"shift1",shift1[1]);
      rca.get(sec,"shift2",shift1[2]);
      Group g1(shift1);
      drum1*=g1;

      sec="drum2 data";
      DrumData drumData2(rch,sec,ped);
```

```
VibData dvd2(rca,"drum2 vibration data");
Drum drum2(pb,drumData2,dvd2);
sec="drum2 shift";
Vec shift2;
rca.get(sec,"shift1",shift2[1]);
rca.get(sec,"shift2",shift2[2]);
Group g2(shift2);
drum2*=g2;

ext=Vp<ExtSys>(5);
ext[1]=hf;
ext[2]=drum1;
ext[3]=drum2;
ext[4]=pot;
ext[5]=lidU;

sec="particle position list";
Z npl=0;
rca.get(sec,"n",npl,ped);
V<Spc> vs(npl);
Word wpl("p");
for (i=1;i<=npl;i++){
   V<R> vi;
   rca.get(sec,wpl&cpmwrite(i),vi,ped);
   Spc pi(vi);
   vs[i]=pi;
}
bool addMirrorImages=false;
rca.get(sec,"addMirrorImages",addMirrorImages,ped);

F<Z,ParT> cr=F5<Z,Rdim,Rdim,V<ParT>,V<Spc>,R,ParT>
   (fsL,fsU,vpp,vs,rotAngleRange)(createFuncPar);
F<ParT,bool> c1=F3<ParT,Drum,Drum,R,bool>
   (drum1,drum2,minimumGap)(cond1FuncPar);
F<X2<ParT,ParT>,bool> c2=F1<X2<ParT,ParT>,R,bool>
   (minimumGap)(cond2FuncPar);
Place<ParT> place(cr,c1,c2);

vc=place(nTot,100000);
Z nAll=vc.dim();
for (i=1;i<=nAll;i++) vc[i].setCol(BLUE);
if (addMirrorImages){
   V<ParT> vc2=vc;
   Spc p0;
   for (i=1;i<=nAll;i++){
      vc2[i].mir_(1);
      Vec xi=vc2[i].get_x()-p0;
      xi.mir_(1);
      vc2[i].set_x(p0+xi);
      vc2[i].setCol(YELLOW);
```

```
            }
            vc&=vc2;
            nAll=vc.dim();
         }
/************************* helps to see what we got in the first place
         Camera cam(2.2);
         drum1.setColor();
         drum2.setColor();
         drum1.mark(cam);
         drum2.mark(cam);
         for (i=1;i<=nAll;i++) vc[i].mark(cam);
         cam.display();
         cpmwait(15);
*/
      }
      else if (topic=="general"){
         sec="general data";
         ped=1;
            // ped='pedantry'
            // we read data that are not present in older ini-files
         R fracVol=1;
         rca.get(sec,"fracVol",fracVol,ped);
         R dh=hPot*(fracVol-1);
         R vh=dh/tTot;
       // setting the velocity of the lid
         Vec vLid(0,vh,0);
         lidU.boost(vLid);

         ext=Vp<ExtSys>(3);
         ext[1]=hf;
         ext[2]=pot; // pot, by design, has a bottom -
            // however not as a separable entity; i.e. forces
            // on the side walls and on the bottom can not be
            // registered separately.
         ext[3]=lidU;
            // closing upper lid

         bool unipolarCharge=false;
         rca.get(sec,"unipolarCharge",unipolarCharge,ped);
         R xBrown=1;
         rca.get(sec,"xBrown",xBrown,ped);
         rc.setR(sec,"xBrown",xBrown);
         F<Spc,bool> c1(true);
         if (xBrown>1){ // then overlap may result, and we take counter-
            // measures.
            const R facFree=2.0;
            R rRestrict=facFree*rP*xBrown;
            c1=F1<Spc,R,bool>(rRestrict)(restrictFunc);
         }
         F<Z,Spc> cr=F2<Z,Rdim,Rdim,Spc>(fsL,fsU)(createFunc);
```

```
      F<X2<Spc,Spc>,bool> c2=F1<X2<Spc,Spc>,R,bool>(dMin)(cond2Func);
      Place<Spc> place(cr,c1,c2);
      V<Spc> xp=place(nTot);
      Z nTot1=xp.dim();
      Z nAll=(xBrown>1 ? nTot1+1 : nTot1);
      vc=V<ParT>(nAll);
      for (i=1;i<=nTot1;i++){
         vc[i]=vpp.cyc(i);
         vc[i].set_x(xp[i]);
         if (!unipolarCharge){
            if (i%2){
               vc[i].invertCharge();
               vc[i].removeMagDipole();
            }
         }
         mc(vc[i],false);
      }
      if (xBrown>1){
         rc.setB("done","Brown",true);
         Spc ori;
         vc[nAll]=vpp[1];
         vc[nAll].set_x(ori);
         vc[nAll]*=xBrown;
         vc[nAll].setCol(LIGHTBLUE);
      }
   }
   else if (topic=="osmosis"){
      sec="osmosis data";
      R expPrsVel=1.5;
         // to let both parts of the osmosic cell with
         // equal pressure one has to give the snaller particles
         // a larger velocity. We assume also in 2D the mass to scale
         // with radius^3 (constant density, 3-dimensional particle
         // geometry). We assume a law pressure ~ v^expPrsVel.
         // For ideal gases one has expPrsVel=2, for the gases where
         // the particle volume fills a substantial part of the space
         // expPrsVel=1.5 is a better value.
      Z sof=1;
      rca.get(sec,"expPrsVel",expPrsVel,sof);
      sec="particle data";
      R facLarge;
      rca.get(sec,"facLarge",facLarge);
      ParT pp1=pp;
      pp1*=facLarge;
      pp1.invertCharge();
      if (dimSpc==2) pp1.setCol(LIGHTBLUE);
      // the larger particles is obtained by scaling
      if (dimSpc==2) pp.setCol(YELLOW);
```

```
Z nPar=2;
V<ParT> vcPrel(nPar);
vcPrel[1]=pp1;
vcPrel[2]=pp;

R_Vector vcr(nPar);
for (i=1;i<=nPar;++i) vcr[i]=vcPrel[i].get_r();
R3 r3=vcr.infMeanSup();
R rMin=r3[1];
R rMean=r3[2];
R rMax=r3[3];
R fac=rMin/rMax;
R exVel=3./expPrsVel;
R facVel=pow(fac,exVel);
sec="data";
rc.setR(sec,"rMin",rMin);
rc.setR(sec,"rMean",rMean);
rc.setR(sec,"rMax",rMax);
rc.setR(sec,"facVel",facVel);
dMin=2*rMax*frac;
// This is the distance between particles that is not allowed
// to surpass.
// setting frac<1 allows to create configurations where
// overlapping particles are allowed

// now we design the fill-in box so that it fitts the upper half
// of the whole space automatically. We hold the same distance
// to
// the semi-permeable membrane that the input data set for the
// distance to the upper lid.
R gapToLid=psU[2]-fsU[2];
R hph=(psU[2]-psL[2])*0.5;
fsL[2]=psU[2]-hph+gapToLid;
F<Z,Spc> cr=F2<Z,Rdim,Rdim,Spc>(fsL,fsU)(createFunc);
F<Spc,bool> c1(true);
F<X2<Spc,Spc>,bool> c2=F1<X2<Spc,Spc>,R,bool>(dMin)(cond2Func);
Place<Spc> place(cr,c1,c2);
V<Spc> xp=place(nTot);
Z nTot1=xp.dim();
// there is no guarantee that nTot positions can be found
// by this method
Z nTot2=2*nTot1;
vc=V<ParT>(nTot2);
Vec shift(0,-hph);
for (i=1;i<=nTot1;i++){
   vc[i]=vcPrel[1];
   vc[i].set_x(xp[i]);
   vc[i+nTot1]=vcPrel[2];
   vc[i+nTot1].set_x(xp[i]+shift);
}
```

```
      for (i=1;i<=nTot2;++i) mc(vc[i]);
      sec="osmosis data";
      V<bool> lidFence;
      rca.get(sec,"lidFence",lidFence);
      Z nMembranes=0;
      Z dlf=lidFence.dim();
      cpmassert(dlf==2,loc);
      for (i=1;i<=dlf;++i) if (lidFence[i]) nMembranes++;
      Z nExt=3+nMembranes;
      ext=Vp<ExtSys>(nExt);
      Z iAct=0;
      ext[++iAct]=hf;
      ext[++iAct]=pot;
      ext[++iAct]=lidU;
      if (lidFence[1]){
         Lid lidM=lidU;
         lidM.setPinhole(rMean);
         lidM.makeBottom();
         Vec shift;
         shift[2]=-iv2.absVal()*0.5;
         lidM*=Group(shift);
         ext[++iAct]=lidM;
      }
      if (lidFence[2]){
         sec="fence data";
         ContactProperties cpFen(rch,sec);
         BodyProperties bpFen(cpFen);
         Body fb(es,bpFen);
         R facPicket,relWidth;
         rca.get("fence data","facPicket",facPicket);
         rca.get("fence data","relWidth",relWidth);

         R rPicket=rMin*(1-facPicket)+rMax*facPicket;
         R gap=2*relWidth*rMin;
         FenData fd(iv1,iv3,rPicket,gap);
       // the smallest particle should pass the fence
         Fen fen(fb,fd);
         ext[++iAct]=fen;
      }
   }
   *this=Sys2(vc,ext,parti,rc);
   Z nParti=parti.card();
   sizeAndRank(); // called already but needed here
   if (size!=1 && nParti!=size){
      ostringstream ost;
      ost<<
         "Sys2(RecordHandler): number of processes does not"<<endl;
      ost<<
         " match the input data for nL1,nL2,nL3 in station.ini"<<endl;
      ost<<"size="<<size<<endl;
```

```
         ost<<"nParti="<<nParti<<endl;
         cpmerror(Word(ost.str()));
      }
   }
   distributeSys2(parti);
      // notice that by definition of setPartition(...), the
      // function argument gets used only for rank==1. For
      // syntactic reasons it has to be there for each rank.

   CPM_MZ
}
//////// end of the most important function ////////////////////////////
//////////////////////////////////////////////////////////////////////////

void Sys2::mark(const Camera& ca, bool display, const Word& text)const
// Shows the whole system even in parallel mode, when the particles
// get distributed over the processes.
// It seems to be important that all ranks have the same system to
// show since it seems to depend on casualities of program run which
// rank does the job of bringing the data to screen.
{
   const Z mL=3;
   Word loc("Sys2::mark(Camera,bool,Word)");
   CPM_MA
   sizeAndRank();
   if (size==1){
      ca.clear();
      Sys2Base::mark(ca,display,text);
      CPM_MZ
      return;
   }
   V<ParT> par_p=par;
   cpmmessage(mL,"nTot_="&cpm(nTot_)&" np="&cpm(np));
   cpmassert(np<=nTot_,loc);
   V< V<ParT> > parAll(size, V<ParT>(0));
   Cpm_com.gather(par_p,parAll);
   V<ParT> vp(nTot_);
   if (rank==1){
      Z i,j;
      for (i=1;i<=size;i++){
         Z n=parAll[i].dim();
         for (j=1;j<=n;j++){
            ParT val=parAll[i][j];
            Z k=val.get_id();
            vp[k]=val;
         }
      }
   }
   Cpm_com.bcast(vp);
   Sys2 sysTot(vp,getExt(),getPartition(),getRc());
```

```
   ca.clear();
   sysTot.Sys2Base::mark(ca,display,text);
   CPM_MZ
}

X2<R,R> Sys2::totChr()const
{
   R qMinus=0, qPlus=0;
   Z np=get_np();
   for (Z i=1;i<=np;i++){
      R q=getPar(i).get_q();
      if (q<0) qMinus+=q;
      if (q>0) qPlus+=q;
   }
   return X2<R,R>(qMinus,qPlus);
}

namespace{
   Word frId("// frame number =");
}

bool Sys2::file(const Word& fn, const Timer& tm)const
// Even in parallel mode the whole system gets written to file.
// The whole system is put together from the parts seen by the
// individual processes via message passing.
// A simplified form of this 'putting together' is being used
// in function mark(...)
{
   Z mL=1;
   Word loc("Sys2::file("&fn&"),Timer)");
   CPM_MA
   sizeAndRank();
   if (size==1){
      file1(fn);
      cpmmessage(mL,loc&": done for size==1");
      return tm.timeOut();
   }
   static Word fnActive("this_is_not_a_filename");
      // it is assumed that fn is a reasonable file name and thus
      // different from this strange initialization value for
      // fnActive
   static OFile ofi;
   static Z count;
   Z i,j;
   if (fn!=fnActive){
      // mechanism for writing some data only once
      fnActive=fn;
      count=1;
      if (rank==1){
         ofi=OFile(fn);
```

```
// partition
         partition.prnOn(ofi());
// exTypes
         V<Word> exTypes(ne);
         for (i=1;i<=ne;++i){
            Word wi=getExt()(i).nameOf();
            exTypes[i]=wi;
         }
         exTypes.prnOn(ofi());
      }
      // now we need to gather from all ranks
      V<ParCT> vcc(np);
      if (cpmverbose>=mL)
         cpmmessage("nTot_="&cpm(nTot_)&
         " np="&cpm(np));
      cpmassert(np<=nTot_,loc);
      for (i=1;i<=np;i++) vcc[i]=par[i].getC();
      V< V<ParCT> > vccAll(size, V<ParCT>(0));
      Cpm_com.gather(vcc,vccAll);
      if (rank==1){
         V<ParCT> vc(nTot_);
         for (i=1;i<=size;i++){
            Z n=vccAll[i].dim();
            for (j=1;j<=n;j++){
               ParCT val=vccAll[i][j];
               Z k=val.get_id();
               vc[k]=val;
            }
         }
         for (i=1;i<=nTot_;i++){
            if (!vc[i].get_idValid())
               cpmwarning("vc["&cpm(i)&"] not initialized");
         }
         vc.prnOn(ofi());
      }
   }
// this gets done for each frame
   typedef X3<B,Z1,ParVT> xcz;
   V<xcz> vcv(np);
   for (i=1;i<=np;i++){
      Z idi=par[i].get_id();
      Spc xi=par[i].get_x();
      B valid(par[i].getOwner()==rank);
      vcv[i].c1()=valid;
      if (!valid) continue;
      vcv[i].c2()=idi;
      ParVT cvi=par[i].getV();
      cpmassert(cvi.get_x()!=Spc(),loc);
      vcv[i].c3()=cvi;
   }
```

```
   V< V<xcz> > vcvAll(size,V<xcz>(0));
   Cpm_com.gather(vcv,vcvAll);
   B earlyStop; // will be set by rank 1 and
      // broadcasted to all others ranks
   if (rank==1){
      V<B> testArray(nTot_);
      V<ParVT> vv(nTot_);
      for (i=1;i<=size;i++){
         Z ni=vcvAll[i].dim();
         for (j=1;j<=ni;j++){
            B valid=vcvAll[i][j].c1();
            if (!valid) continue;
            Z k=vcvAll[i][j].c2();
            ParVT val=vcvAll[i][j].c3();
            vv[k]=val;
            testArray[k]=true;
            cpmassert(val.get_x()!=Spc(),loc);
         }
      }
      for (i=1;i<=nTot_;i++){
         bool bi=testArray[i];
         if (!bi){
            ostringstream ost;
            ost<<endl<<"particle "<<i<<" could not be found";
            cpmwarning(Word(ost.str()));
         }
      }
      Word frNum=frId&cpm(count);
      count++;
      frNum.prnOn(ofi());
      B vvOK=ParT::isVal(vv);
      vv.prnOn(ofi());
      for (i=1;i<=ne;++i) getExt()(i).prnOn(ofi());
      earlyStop=B(tm.timeOut()).vel(vvOK.non());
   }
   Cpm_com.bcast(earlyStop);
   cpmmessage(mL,"Sys2::file("&fn&") done");

   return earlyStop;
}

void Sys2::file1(const Word& fn)const
// creating a file and writing to it sich that
// the extractor class in cpmextractorx.h can read it
// Notice the new order partition, exTypes, vc (all once)
// vv, ex (one per frame)
// This is the simple method to be used for size==1
// in function file
{
   Z mL=2;
```

```
   Word loc("Sys2::file1("&fn&")");
   CPM_MA
   static Word fnActive("this_is_not_a_filename");
      // it is assumed that fn!="this_is_not_a_filename"
   static OFile ofi;
   static Z count;
   Z i,id,oi;
   if (fn!=fnActive){ // do once
      ofi=OFile(fn);
      fnActive=fn;
      count=1;
// partition
      partition.prnOn(ofi());
// exTypes
      V<Word> exTypes(ne);
      for (i=1;i<=ne;++i){
         Word wi=getExt()(i).nameOf();
         exTypes[i]=wi;
      }
      exTypes.prnOn(ofi());
// vc
      V<ParCT> vc(nTot_);
      for (i=1;i<=np;i++){
         id=par[i].get_id();
         oi=par[i].getOwner();
         if (oi>0) vc[id]=par[i].getC();
      }
      cpmmessage(mL,"nTot_="&cpm(nTot_)&" np="&cpm(np));
      vc.prnOn(ofi());
   }
// do for each frame
   V<ParVT> vv(nTot_);
   for (i=1;i<=np;i++){
      id=par[i].get_id();
      oi=par[i].getOwner();
      if (oi>0) vv[id]=par[i].getV();
   }
   Word frNum=frId&cpm(count);
      // frId starts with '//', thus irrelevant for the
      // Extractor
   count++;
   frNum.prnOn(ofi());
   B vvOK=ParT::isVal(vv);
// print vv
   vv.prnOn(ofi());
// print ex
   for (i=1;i<=ne;++i) getExt()(i).prnOn(ofi());
   CPM_MZ
   cpmassert(vvOK,loc);
}
```

```
void Sys2::distributeSys2(const Partition& parti)
{
   const Z mL=1;
   Word loc("Sys2::distributeSys2(...)");
   CPM_MA
   sizeAndRank();
   if (size==1){
      cpmmessage(mL,loc&" trivially done, size==1");
      CPM_MZ
      return;
   }
   cpmmessage(mL,"size="&cpm(size)&" rank="&cpm(rank));
   V<Word> typeList;
   Z i;
   if (rank==1){
      typeList=V<Word>(ne);
      for (Z i=1;i<=ne;++i){
         Word ti=ext(i).nameOf(); // having [i] instead
            // of (i) would give the name of the common base class
         typeList[i]=ti;
         cpmmessage(loc&": type is "&ti);
      }
   }
   Cpm_com.bcast(typeList);
   if (rank==1){
      Z j,paca=parti.card();
      if (paca!=size){
         Word mes("Sys2::distributeSys2(const Partition&): ");
         mes&="partition does not match the number of processes";
         cpmwarning(mes);
         cpmmessage("partition card() found as"&cpm(paca));
      }
      partition=parti; // now *this has the correct partition

      cpmassert(np==nTot_,loc);
         // in the present state of initialization par is the array of all
         //
         // particles of the whole system (all processes together)
      V< V<bool> > vvb(size,V<bool>(np,false));
      for (j=1;j<=np;j++){
         Z oj=par[j].getOwner();
         vvb[oj][j]=true;
      }
      for (i=2;i<=size;i++){
         cpmmessage("sending Sys2 to process "&cpm(i)&
            " in preparation");
         Z tagi=Com::tagFromToStatic(1,i,size);
         Sys2 sta(getPar(),getExt(),getPartition(),getRc());
            // both the statement
```

```
         //    Sys2 sta(*this);
         // and the experimental replacement
         //    Sys2 sta=*this;
         // make the following statement change *this if
         // #define CPM_USECOUNT is set and g++ is used for
         // compilation. The use-count mechanism seems to get
         // stressed too much if this function is called
         // within a constructor (as the present function will
         // be used exclusively) where *this is to be considered
         // incomplete. So to be defensive we call here only
         // functions that are members of the base class.
      sta.selectParticles(vvb[i]);
         // this does not mean that nTot_ is changed to
         // the number of selected particles

      Z ne=sta.getExt().dim();
      ostringstream ost;
      for (Z ii=1;ii<=ne;++ii){
         ost<<sta.getExt()(ii).nameOf()<<" ";
      }
      cpmmessage(Word(ost.str()));

      cpmmessage(mL,"preparation done, sending started");
      sta.send(i,tagi);
         // sending to i the station with particles that are right
         // for i, special sending implementation for extra large
         // data structures
      cpmmessage("sending Sys2 to process "&cpm(i)&" done; "&
         cpm(sta.get_np())&" particles sent");
   }
   selectParticles(vvb[1]);
      // adjusting the particles of rank 1
   cpmmessage("rank 1 has "&cpm(get_np())&" particles");
}
else{
   // receiving from rank 1 what was sent to rank
   Sys2 staRec(typeList);
   staRec.rec(1,Com::tagFromToStatic(1,rank,size));
   *this=staRec;
      // Notice that despite the internal complexity of
      // instances of Sys2 this initialization of an instance
      // (even within a constructor) from sent data follows
      // a very simple syntax
   cpmmessage("we got "&cpm(get_np())&" particles");
}
B masterReady;
if (rank==1) masterReady=true;
Cpm_com.bcast(masterReady);
while (!masterReady){;}
CPM_MZ
```

```
}

bool Sys2::prnOn(ostream& str)const
{
   Z mL=2;
   Word loc("Sys2::prnOn(ostream)");
   CPM_MA
   cpmwt("Sys2");
   if(!Sys2Base::prnOn(str)){
      cpmwarning(loc&": printing of base failed");
      CPM_MZ
      return false;
   }
   cpmp(nTot_);
   cpmp(rc_);
   CPM_MZ
   return true;
}

bool Sys2::scanFrom(istream& str)
{
   Z mL=2;
   Word loc("Sys2::scanFrom(istream)");
   CPM_MA
   if (!Sys2Base::scanFrom(str)){
      cpmwarning(loc&": scanning of base failed");
      CPM_MZ
      return false;
   }
   cpms(nTot_);
   cpms(rc_);
   CPM_MZ
   return true;
}

namespace{

   R logS(R x){ return x<1e-12 ? -12 : ::log10(x);}
}

X2<Z,Word> Sys2::docuStep(OFileStream& ofs)const
{
   Word loc("Sys2::docuStep(OFileStream& ofs)");
   Z mL=3;
  // static bool firstrun=true;
   static R eTotIni=0;
   static R eTotIniInv=0;
   CPM_MA
   Word title("idle");
   Z n=0, np=nTot_,ne=get_ne();
```

```
Word sec="topic under consideration";
Word topic,documode;
   // Although documode come from varius locations in the
   // ini-file, there is a single variable associeted
   // with it !
rc_.get(sec,"topic",topic);
if (topic=="osmosis"){
   sec="osmosis data";
   rc_.get(sec,"documode",documode);
}
else if (topic=="general"){
   sec="general data";
   rc_.get(sec,"documode",documode);
}
else if (topic=="spherical_tank"){
   sec="spherical_tank data";
   rc_.get(sec,"documode",documode);
}
else if (topic=="drum"){
   sec="drum data";
   rc_.get(sec,"documode",documode);
}
else if (topic=="mill"){
   sec="mill data";
   rc_.get(sec,"documode",documode);
}
else{
   cpmwarning(loc&" unvalid topic");
   goto Finis;
}
Z i;

if (topic=="osmosis" && documode=="normal"){
   R1 tx(get_t());
   Z sumsu=0;
   Z sumll=0;
   for (i=1;i<=np;++i){
      Z v1=parType(i);
      Z v2=parLoc(i);
      if (v1==1){
         if (v2==2) sumsu++;
      }
      else{ // v1=2
         if (v2==1) sumll++;
      }
   }
   tx.prnOn(ofs());
   R1(sumsu).prnOn(ofs());
   R1(sumll).prnOn(ofs());
```

```
   n=3;
   title=
      "x: time, y: # of small particles in upper box (black), # of";
   title&=" large part. in lower box (violet)";
}
else if (documode=="force"){
   R1 tx(get_t());
   Z yDir=2;
   R mf1=getExt()(2).getForce()[yDir];
      // pot which consists of a bottom plane and side walls
      // This is the force which the pot feels from the particles
   R mf2=getExt()(3).getForce()[yDir];
      // upper lid.
      // This is the force which the lid feels fromthe particles
   Spc ori;
   FieldSpace fExt=getExt()(1).getField(ori);
      // the field is homogeneous and may be evaluated at any point
   R accG=(fExt.c3())[yDir];
   R fg=get_m()*accG;
   // This is the force which the particles feel from the field
   tx.prnOn(ofs());
   R1(mf1).prnOn(ofs());
   R1(mf2).prnOn(ofs());
   R1(fg).prnOn(ofs());
   n=4;
   title="x: time, y: force on bottom (black), ceiling (violet)";
   title&=" weight of particles (blue)";
}
else if (documode=="mean_force"){
   R1 tx(get_t());
   Z yDir=2;
   R mf1=-getExt()(2).getMeanForce()[yDir];
   R mf2=getExt()(3).getMeanForce()[yDir];
      // here the intention is to easily see whether the pressure
      // on the lid and the bottom are the same
   tx.prnOn(ofs());
   R1(mf1).prnOn(ofs());
   R1(mf2).prnOn(ofs());

   n=3;
   title="x: time, y: mean force on bottom (black), ceiling (violet)";
}
else if (documode=="energy"){
   R tx=get_t();
   R r1=kinEnr();
   FieldSpace fs=getExt()(1).getField(Spc());
   R r2=potEnr()+potElasticExt()+potHomFil(fs);
   R r3=r1+r2;
   R1(tx).prnOn(ofs());
   R1(r1).prnOn(ofs());
```

```
   R1(r2).prnOn(ofs());
   R1(r3).prnOn(ofs());
   n=4;
   title="x: time, y: E_kin (black), E_pot (violet), E (blue)";
}
else if (documode=="total_energy"){
   R tx=get_t();
   FieldSpace fs=getExt()(1).getField(Spc());
   R r3=kinEnr()+potEnr()+potElasticExt()+potHomFil(fs);
   if (!running_){
      eTotIni=r3;
      eTotIniInv=cpminv(cpmabs(r3));
      running_=B(true);
   }
   else{
      R r4=(r3-eTotIni)*eTotIniInv;
      cpmvalue("dEtotRel",r4,4);
      R1(tx).prnOn(ofs());
      R1(r4).prnOn(ofs());
   }
   n=2;
   title="x: time, y: relative change of total energy";
}
else if (documode=="momentum"){
   R tx=get_t();
   R r1=0;
   Z com=2;
   for (Z i=1;i<=np;++i){
      ParT pi=par[i];
      r1+=pi.get_m()*pi.get_v()[com];
   }
   R r2=getExt()(1).get_momentum()[com];
   R r3=getExt()(2).get_momentum()[com];
   R r4=r1+r2+r3;
   R1(tx).prnOn(ofs());
   R1(r1).prnOn(ofs());
   R1(r2).prnOn(ofs());
   R1(r3).prnOn(ofs());
   R1(r4).prnOn(ofs());
   n=5;
   title="x: time, y: vertical component of momentum of";
   title&=" particles(black), pot(violet), lid(blue), all(cyan)";
}
else if (documode=="angular_momentum"){
   R tx=get_t();
   Z com=3;
   R r1=getExt()(2).get_angularMomentum()[com];
   R1(tx).prnOn(ofs());
   R1(r1).prnOn(ofs());
   n=2;
```

```
   title="x: time, y: 3-component of angular momentum(violet)";
}
else if (documode=="torque"){
   R tx=get_t();
   Z com=3;
   R r2=getExt()(2).getTorque()[com];
   R r3=getExt()(3).getTorque()[com];
   R1(tx).prnOn(ofs());
   R1(r2).prnOn(ofs());
   R1(r3).prnOn(ofs());
   n=3;
   title=
      "x: time, y: 3-component of torque 2(black), torque 3(violet) ";
}
else if (documode=="equilibrium"){
   V<R> ek(np);
   for (i=1;i<=np;++i) ek[i]=par[i].kinEnr();
   DisR dek(ek);
   R tDisplay=rc_.getR("equilibrium control","tDisplay");
   Word title1=
      "x: kinetic energy, y: binned particle numbers(black),";
   title1&=" Boltzmann fit(violet)";
   R2 val=dek.tBoltzmann(tDisplay,title1);
   R tx=get_t();
   R r1=logS(val[1]);
   R r2=logS(val[2]);
   R1(tx).prnOn(ofs());
   R1(r1).prnOn(ofs());
   R1(r2).prnOn(ofs());
   n=3;
   title="x: time, y: log T(black), log of misfit(violet)";
}
else if (documode=="trajectory"){
   Spc ori;
   Z np=nTot_;
   Spc p=par[np].get_x();
   Vec x=p-ori;
   R r1=x[1];
   R r2=x[2];
   R1(r1).prnOn(ofs());
   R1(r2).prnOn(ofs());
   n=2;
   title="Path of the large particle (equal scale in x- and ";
   title&="y-direction). Start at center of screen.";
}
else if (documode=="idle"){
   cpmmessage(loc&": idle documode selected");
}
else{
   cpmwarning(loc&": unvalid documode");
```

```
   }
Finis:
   CPM_MZ;
   return X2<Z,Word>(n,title);
}

Z Sys2::parType(Z i)const
   // returns a discrete type number of par[i]. For the osmosis
   // application 1 for the small particles and 2 for the large ones.
{
   Word topic=rc_.getW("topic under consideration","topic");
   R rMean=rc_.getR("data","rMean");
   if (topic=="osmosis"){
      R ri=par[i].get_r();
      return ri<=rMean ? 1 : 2;
   }
   else return 0;
}

Z Sys2::parLoc(Z i)const
  // returns a discrete type number of par[i]. For the osmosis
  // application
  // 1 for the small particles and 2 for the large ones
{
   Word topic=rc_.getW("topic under consideration","topic");
   if (topic=="osmosis"){
      Spc xi=par[i].get_x();
      R h=xi[2];
      return h<=0 ? 1 : 2;
   }
   else return 0;
}

void Sys2::boost(const Vec& v, R angleRange)
{
   Word topic=rc_.getW("topic under consideration","topic");

   if (topic=="osmosis"){ // let the smaller particles
      // get the same kinetic energy than the large ones
      R facVel=rc_.getR("data","facVel");
      Angle a(angleRange,DEG);
      Z j=0;
      for (Z i=1;i<=nTot_;++i){
         Vec vi=v;
         Z ti=parType(i);
         if (ti!=1) vi*=facVel;
         vi=vi.randomize(0,a,++j);
         par[i].boost(vi);
      }
   }
```

```
   else{
      Sys2Base::ranBoost(v,0,angleRange);
   }
}

void Sys2::setVel()
{
   Word sec("runcontrol2");
   V<R> v0=rc_.getRs(sec,"v0");
   R vAngleRange=rc_.getR(sec,"vAngleRange");
   Vec vIni(v0);
   boost(vIni,vAngleRange);
      // now each particle has its regular velocity
      // Finally we have to stop the Brownian large
      // particles if we are dealing with this case.
   sec="topic under consideration";
   Word topic=rc_.getW(sec,"topic");
   if (topic=="general"){
      sec="general data";
      R xBrown=rc_.getR(sec,"xBrown");
      if (xBrown>1){
         Z np=nTot_;
         par[np].stop();
      }
   }
}

} // namespace
```

194 cpmsys2dat.h

```
//? cpmsys2dat.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_SYS2DAT_H_
#define CPM_SYS2DAT_H_
/*
   Description: declares class of data related to class Sys2.
*/

#include <cpmparofspc.h>
// defines CPM_DIMDEF2 or CPM_DIMDEF3

#if defined(CPM_DIMDEF2)

#define CPM_DIM CpmDim2
#define CPM_TOP CpmPala2_2

namespace CPM_TOP{
   #include <cpmsys2datx.h>
}

#undef CPM_DIM
#undef CPM_TOP

#endif

//////////////////////////////////////////////////////////////////

#if defined(CPM_DIMDEF3)

#define CPM_DIM CpmDim3
#define CPM_TOP CpmPala2_3

namespace CPM_TOP{
   #include <cpmsys2datx.h>
}

#undef CPM_DIM
#undef CPM_TOP

#endif

#endif
```

195 cpmsys2dat.cpp

```
//? cpmsys2dat.cpp
//? Status of work 2008-10-25.
//?
#include <cpmsys2dat.h>

#if defined(CPM_DIMDEF2)

#define CPM_DIM CpmDim2
#define CPM_TOP CpmPala2_2
#include <cpmsys2datcpp.h>
#undef CPM_DIM
#undef CPM_TOP

#endif

#if defined(CPM_DIMDEF3)

#define CPM_DIM CpmDim3
#define CPM_TOP CpmPala2_3
#include <cpmsys2datcpp.h>
#undef CPM_DIM
#undef CPM_TOP

#endif
```

196 cpmsys2datcpp.h

```
//? cpmsys2datcpp.h
//? Status of work 2008-10-25.
//?
#include <cpmunits.h>
#include <cpmroot.h>
#include <cpmperm.h>
#include <cpms.h>

namespace CPM_TOP{

using CPM_DIM::Spc;
using CPM_DIM::Vec;
using CPM_DIM::AxVec;
using CPM_DIM::Group;
using CPM_DIM::RefSys;
using CPM_DIM::dimVec;
using CPM_DIM::Rdim;
using CpmGeo::Angle;
using CpmGeo::RAD;

using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmLinAlg;

R Sys2Dat::getArea()const
{
   return get_xLength()*get_yLength();
}

V<Rdim> Sys2Dat::testPoints(Z n)const
{
   Iv iv(xL,xU);
   V<R> vx=iv.centers(n);
   V<Rdim> res(n);
   for (Z i=1;i<=n;i++) res[i]=Rdim(vx[i],getH());
   return res;
}

bool Sys2Dat::scanFrom(std::istream& str)
{
   cpms(dim_);
   cpms(xL);
   cpms(xU);
   cpms(yL);
   cpms(yU);
   cpms(zL);
   cpms(zU);
```

```
   adjust(); // here the assertion in in adjust()
      // may point out inconsitencies if SysViewerAppl
      // processes a inputfile which was created within
      // a dimension context inconsistent with the value dim_
   return true;
}

bool Sys2Dat::prnOn(std::ostream& str)const
{
   CpmRoot::writeTitle("Sys2Dat",str);
   cpmp(dim_);
   cpmp(xL);
   cpmp(xU);
   cpmp(yL);
   cpmp(yU);
   cpmp(zL);
   cpmp(zU);
   return true;
}

void Sys2Dat::adjust()
{
   Z mL=3;
   Word loc("Sys2Dat::adjust():");
   CPM_MA
   if (dim_!=dimVec){
      ostringstream ost;
      ost<<loc<<"We are reading a "<<dim()<<"D movie with the "
      <<dimVec<<"D version of the viewer program."<<endl
      <<"One needs to set mode="<<dim()<<
      " in section selection of the ini-file";
      cpmwarning(ost);
   }
   cpmassert(dim_==dimVec,loc);
   di1=DiIv(xL,xU);
   di2=DiIv(yL,yU);
   if (dimVec>2) di3=DiIv(zL,zU);
   CPM_MZ
}

Sys2Dat::Sys2Dat(R xL_,R xU_, R yL_,R yU_, R zL_, R zU_):
   xL(xL_),xU(xU_),yL(yL_),yU(yU_),zL(zL_),zU(zU_)
{
   if (xL>xU) cpmerror("Sys2Dat(...): xL>xU");
   if (yL>yU) cpmerror("Sys2Dat(...): yL>yU");
   if (zL>zU) cpmerror("Sys2Dat(...): zL>zU");
   dim_=dimVec;
   Sys2Dat::adjust(); // in this case the assertion in
      // adjust holds true
}
```

```
Rdim Sys2Dat::f(const Rdim& w)const
{
   Rdim res;
   res[1]=di1.p(w[1]);
   res[2]=di2.p(w[2]);
   if (dimVec>2) res[3]=di3.p(w[3]);
   return res;
}

Rdim Sys2Dat::fInv(const Rdim& w)const
{
   Rdim res;
   res[1]=di1.x(w[1]);
   res[2]=di2.x(w[2]);
   if (dimVec>2) res[3]=di3.x(w[3]);
   return res;
}

} // namespace
```

197 cpmsys2datx.h

```
//? cpmsys2datx.h
//? Status of work 2008-10-25.
//?
   using CPM_DIM::Rdim;
   using CPM_DIM::ParOfSpc;
      // partition of space, see cpm2/cpmpartition.h

   using CpmRoot::Word;
   using CpmRoot::R;
   using CpmRoot::Z;

   using CpmRootX::Z1;
   using CpmRootX::R1;
   using CpmRootX::B;

   using CpmGeo::Angle;
   using CpmGeo::DiIv;
   using CpmGeo::RAD;
   using CpmGeo::Iv;

   using CpmArrays::V;

//doc namespace CpmTOP{

//////////////////////////// class Sys2Dat ////////////////////////////////
// data that determine a rectangular box and data relevant to
// partitioning particles into groups to be handled by different processes
// in parallel computing. Here we have a rectangular box as the living
// room for the and not the complicated geometry as in PaDyAn

class Sys2Dat { // sys2 data

   Z1 dim_;
      // 2 or 3; info on dimension which does
      // not depend on the reading context

   R1 xL;
      // boundary in x-direction, smallest x-value
   R1 xU;
      // boundary in x-direction, largest x-value
   R1 yL;
      // boundary in y-direction, smallest y-value
   R1 yU;
      // boundary in y-direction, largest y-value
   R1 zL;
      // boundary in z-direction, smallest z-value
   R1 zU;
```

```
      // boundary in z-direction, largest z-value

// dependent data for efficiency

   DiIv di1,di2,di3;

   void adjust();
      // setting the values of the dependent quantities
public:
   typedef Sys2Dat Type;
   Sys2Dat(){}
   Sys2Dat(R xL, R xU, R yL, R yU, R zL=0, R zU=0);
      // constructor from independent data
   CPM_IO
   Rdim f(const Rdim&)const;
      // maps the region of interest into the unit cube
   Rdim fInv(const Rdim&)const;
      // is an inverse function only on the unit cube
   Z dim()const{ return dim_;}

   R getArea()const;
      // area in the (1,2)-plane

   R get_xL()const{ return xL;}
   R get_xU()const{ return xU;}
   R get_yL()const{ return yL;}
   R get_yU()const{ return yU;}
   R get_zL()const{ return zL;}
   R get_zU()const{ return zU;}
   R get_xLength()const{ return xU-xL;}
   R get_yLength()const{ return yU-yL;}
   R getH()const{ return yU-yL;}
   R get_zLength()const{ return zU-zL;}
   V<Rdim> testPoints(Z n)const;
   Rdim corL()const{ return Rdim(xL,yL,zL);}
      // lower corner
   Rdim corU()const{ return Rdim(xU,yU,zU);}
      // upper corner
   Iv iv1()const{ return Iv(xL,xU);}
      // interval on 1-axis
   Iv iv2()const{ return Iv(yL,yU);}
      // interval on 2-axis
   Iv iv3()const{ return Iv(zL,zU);}
      // interval on 3-axis
};

typedef ParOfSpc<Sys2Dat> Partition;
   // partition with a simple rectangular box geometry
```

198 cpmsys2x.h

```
//? cpmsys2x.h
//? Status of work 2008-10-25.
//?
/*
  Describing a particle system enclosed by structures
  of type Pot, Lid, Fen suitable for simulating osmosis.
  Without using these confining structures also free
  particle systems such as star clusters can be studied.

  Implementation of fast and parallel dynamics as defined in
  cpmpardynsys.h
*/
namespace CPM_TOP{
   // Topics namespace

   using CPM_DIM::Vec;
   using CPM_DIM::AxVec;
   using CPM_DIM::Spc;
   using CPM_DIM::Group;
   using CPM_DIM::RefSys;
   using CPM_DIM::Plane;
   using CPM_DIM::InTens;
   using CPM_DIM::Rdim;
   using CPM_DIM::ParOfSpc;
      // partition of space, see cpmpartition.h

   using CPM_CAMERA::Camera;

   using CPM_RIGID_BODY::RigidObject;

   using CPM_RIGID_BODY::PointPar;
   using CPM_RIGID_BODY::PointParC;
   using CPM_RIGID_BODY::PointParV;

   using CPM_RIGID_BODY::ComPar;
   using CPM_RIGID_BODY::ComParC;
   using CPM_RIGID_BODY::ComParV;

   using CPM_PHYSICS::ExtSys;
   using CPM_PHYSICS::Body;
   using CPM_PHYSICS::Pot;
   using CPM_PHYSICS::Lid;
   using CPM_PHYSICS::Fen;
   using CPM_PHYSICS::SphTank;

   using CPM_PHYSICS::Force;
   using CPM_PHYSICS::ForTor;
```

```
   using CPM_PHYSICS::FieldSpace;
   using CPM_PHYSICS::ScalarField;
   using CPM_PHYSICS::VectorField;
   using CPM_PHYSICS::PointCharge;
   using CPM_PHYSICS::SourceOfVectorField;
   using CPM_PHYSICS::MagDipole;

   using CpmDynSys::ParDynSys;

   using CpmGraphics::Graph;
   using CpmGraphics::Frame;
   using CpmGraphics::Frames;

   using CpmImaging::Color;
   using CpmImaging::Palette;
   using CpmImaging::INVISIBLE;

   using CpmProperties::BodyProperties;
   using CpmProperties::ParticleProperties;

   using CpmAlgorithms::Switch;

   using CpmRoot::Word;
   using CpmRoot::R;
   using CpmRoot::Z;

   using CpmRootX::Z1;
   using CpmRootX::R1;
   using CpmRootX::B;
   using CpmRootX::RecordHandler;
   using CpmRootX::Record;

   using CpmArrays::R_Vector;
   using CpmArrays::I_Vector;
   using CpmArrays::R_Matrix;
   using CpmArrays::V;
   using CpmArrays::Vp;
   using CpmArrays::Vo;
   using CpmArrays::X2;
   using CpmArrays::X4;

   using CpmFunctions::F;

   using CpmSystem::Error;
   using CpmSystem::Timer;
   using CpmSystem::OFileStream;

//////////////////////////////// class Sys2 ///////////////////////
// Parallelism now extended to all public functions
```

```
#if defined(CPM_POINT_PARTICLES)

typedef PointPar ParT;
typedef Force ForT;
typedef PointParC ParCT;
typedef PointParV ParVT;

#else

typedef ComPar ParT;
typedef ForTor ForT;
typedef ComParC ParCT;
typedef ComParV ParVT;

#endif

typedef ParDynSys<ParT,ExtSys,Vec,Spc,Group,Partition,ForT,
   FieldSpace,ScalarField,VectorField,PointCharge,
   SourceOfVectorField,MagDipole,PointPar,Camera> Sys2Base;
   // specialization of the 'universal' template class in
   // cpmpardynsys.h

typedef CPM_TOP::ParExt<ParCT,ParVT,ParT,ExtSys,Sys2Base> ExtrSys2;
   // file oriented type needed in
   // explicit Sys2(const ExtrSys2& extr);

class Sys2:public Sys2Base{
// Sys2, the final physical system for confined granular matter dynamics

   mutable B running_;
      // introduced 2005-06-14
      // initialized as false, which is OK
      // A device for allowing remembering physical properties
      // of the initial state in function docuStep(...).
      // Presently only used for documode=="total_energy".

   Z1 nTot_;
      // total number of particles (in all processes)

   Record rc_;
      // flexible container for data, instead of naming them all
      // Not named rec_ since this would suggest also to use rec and this
      // would clash with function rec for parallel send and receive

   Sys2(V<ParT> const& vc, Vp<ExtSys> const& ext,
      Partition const& pr, Record const& rec, bool careful=true);
      // auxiliary constructor which lets rank==1 build the
      // whole system and leaves distribution to other
      // processes as a task for additional code.
      // If the last argument has value true, only those
```

```
      // particles from vc will be used that are reasonably
      // within the process volume SubsetOfSpace pr.getX().

   explicit Sys2(const ExtrSys2& extr);
      // construction from a file extractor; not for
      // parallel computing. Needed for reading movie files.

   explicit Sys2(V<Word> const& typeList);
      // Setting the exact types for the external system
      // needed in function distributeSys2 as a prerequisite
      // to properly receiving the data describing the
      // external systems. In a sense this is a trick to make
      // my polymorphic array Vp<> work also under the harder
      // conditions of parallelism (2006-02-08).

   void distributeSys2(const Partition&);
      // assumes that rank 1 has all particles available
      // The second argument was introduced in 02-04-23 to
      // avoid long-standing malfunction when reading states from file.
      // The distribution of particles with additional overlap
      // deactivated seems to be appropriate only for states as coming
      // out directly from a clean constructor which did not yet
      // perform dynamical time steps.
      // Does not only set the particles for the other processes but
      // makes the whole station known to ranks different from 1.

   void file1(const Word& fn)const;
      // filing for size==1

   typedef Sys2 Type;

public:

   static X4<V<ParT>,Vp<ExtSys>,Partition,Z>
      readFromMovie(const Word& movieFile, R frac=1);
      // returns the quantities specified in the return value
      // from the ith frame of a moviefile, where i is characterized
      // by a fraction frac of the whole movie.
      // i is returned as the 6th component of the return value.
      // Thus frac=1 corresponds to the last frame of the movie.

   explicit Sys2(const RecordHandler&);
      // parallel version!
      // construction from design data

   explicit Sys2(const Word& movieFile, const Record& rc,
      R frac=1);
      // parallel version!
      // construction from the ith frame of a moviefile
      // where i is characterized by a fraction frac of the whole movie
```

```
      // Thus frac=1 corresponds to the last frame of the movie.
      // The argument rc contributes only through its section
      // 'partition_data':
      // It adds the data that define the partition attribute of the
      // base class. This over-writes the value written to file. This
      // has the advantage, that one can use a state from
      // an movie file as initial state of a run for which the number of
      // parallel processors differs from the number that was active when
      // creating the movie file.
      // The profile wave numbers k1Sh, k1PC, k3Sh, k3PC get adjusted to
      // the new partition in order not to disturb the particles by
      // moving them from the end of the partition domain to the beginning
      // (was not the case till 02-07-16).
      // If the new partition corresponds to a smaller region of space
      // than the original one, the number of particles get reduced
      // accordingly.
      // If, however; rc is trivial (i.e. rc.isVoid()==true) the
      // partition attribute will be taken from movieFile and it is
      // assumed that the data of particles and external systems
      // in movieFile are mutually consistent so that the adjustments
      // discribed earlier will not be made. This is the desired behavior
      // for continuation of moviefiles, i.e for taking the final frame
      // of a movie file as the initial state of a new run.

   Sys2(void);
      // Works in parallel programs since no structures get created
      // which depend on the partition of space (in particular,
      // no particles!). A correct idiom for creating and using a
      // Sys2 in a parallel program is (see padyanapscpp.h)
/*
       Sys2 ton;
       if (...){
            ton=Sys2(aRecordHandler);
               // makes ton the right object in each process
       }
       else{
            ton=Sys2(aWord,aRecord);
               // makes ton the right object in each process
       }
       ton.run(...);
*/
      // notice that the default constructor together with '='
      // allows more flexible program control than non-trivial
      // constructors alone.

// functions that are virtual in Sys2Base

   CPM_IO
      // input and output in C+- style
```

```
CPM_NAM(Sys2)
   // 'modern' form of providing the nameOf function

virtual void mark(const Camera& ca, bool display,
   const Word& text=Word())const;
   // display system on screen;
   // The parts of the systems for which the various processes are
   // responsible get messaged to process 1 and graphically
   // represented by process 1 in succession.

virtual X2<Z,Word> docuStep(OFileStream& ofs)const;
   // if active==false, nothing is written to ofs and only the
   // return value is given. The return value puts together the
   // number of data arrays and a diagram title.

virtual void boost(const Vec& v, R vAngleRange);
   // boost the light particles with  velocity v and the heavy ones
   // with a lower velocity that gives them
   // the same kinetical energy (simulating equal temperatures)

virtual void setVel();
   // virtual and trivial in Sys2Base, see explanation in
   // cpmpardynsys.h
   // Here we implement the setting of suitable velocity distributions
   // controlled by the state of rc_. The velocities are understood as
   // addition to the existing velocities (which a state may have if
   // it was created as the final state of a pre-run).

Z parType(Z i)const;
   // returns a discrete type number of par[i]. For the osmosis
   // application 1 for the small particles and 2 for the large ones.
   // In general, the value has to be determined by the state of par[i]
   // and by the state of pc_ (which is highly flexible).

Z parLoc(Z i)const;
   // returns a discrete type number of par[i]. For the osmosis
   // application 1 for the small particles and 2 for the large ones.

virtual bool file(const Word& fn, const Timer& tm)const;
   // virtual in base
   // writes the state into a suitable file, which should
   // be made visible by a suitable viewer program on a
   // single computer. Since the number of particles to be
   // considered is huge, it is rather evident that one should
   // file not too much redundant data. This suggests to divide
   // the data of an object in constant ones and those that change
   // with time. This is implemented in the present function.

virtual Z get_nTot()const{ return nTot_;}
   // overwrites virtual function of Base
```

```
// access functions concerning rec_
   Record getRc()const{ return rc_;}

   void setRc(Record const& rc){ rc_=rc;}

// constant functions returning properties of the system

   Spc carCenOfMas()const;
      // center-of-mass of the set of particles

   X2<R,R> totChr()const;
      // Name means 'total charges'.
      // Return.first is the sum of all negative charges of particles.
      // Return.second is the sum of all positive charges of particles.
      // Notice that increasing values from left to right are
      // considered natural order; so negative charges have to come
      // first.

   R getHeight()const;
      // evident
};
} // namespace
```

199 cpmsystem.h

```
//? cpmsystem.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_SYSTEM_H
#define CPM_SYSTEM_H
/*
   Purpose: Declaring basic facilities for a C+- based program to
   communicate with the user.
*/

#include <cpmword.h>

namespace CpmSystem{ // C+- system basic facilities

   using namespace CpmStd;

   using CpmRoot::Word;
   using CpmRoot::R;
   using CpmRoot::Z;

////////////////////// function glue ////////////////////////////////

Word glue(Word const& path, Word const& file);
   //: glue
   // Connects a (partial)directory name and a file name
   // to give a full file name. 'glues' path and file together.
   // For extended==false, the result is simply
   // path&Word('/')&file. This is the correct behavior if file is a
   // normal filename which does not contain directory separators.
   // if extended==true we take into account that file may start
   // with ./ or ../. Then these dots will be eliminated.
   // Actually all all dots will be eliminated except the one
   // preceeding the file extension and

//////////////////////////// class Message ////////////////////////////////

class Message{ // provides basic output (unidirectional communication)
      // channels for a program. Two output files (cpmcerr and cpmdata
      // according to their define alias, see comments to data member
      // 'out' for these important entities) and a status bar featuring
      // 4 addressable segments are made available.
      // If we are running a console application, messages also go to the
      // console.
      // There is also limited communication in the other direction:
      // A file cpmconfig.ini, if it can be opened, will be used to
      // modify some static data members of the present class. This
      // excludes the data used to create the files cpmcerr and cpmdata.
```

```
   //
   // This class only has static data and methods, thus all instances
   // are identical. The information for initializing the static data
   // comes from the file cpmsystemdependencies.h.
   // A statement
   //    Cpmsystem::updFromIniFile();
   // allows to change most static data by replacing them by values
   // read from the ini-file cpmconfig.ini. Since updFromIniFile is a
   // friend of Message, it can do this replacement of private data of
   // class Message.

friend void updFromIniFile();
   // this allows to manipulate the private data of Message, by a
   // function which can be implemented only later when class
   // RecordHandler is available. Will be implemented in
   // cpmapplication.cpp

static bool streamInitialized;

static Z vpw_, vph_, fsw_, fsh_;
   // viewport width, viewport height,
   // framestore width, framestore height,
static Word sttmes1;
static Word sttmes2;
static Word sttmes3;
static Word sttmes4;

static Z sttBarWid1;
   // status bar width 1
   // the status bar has 4 'panes' and the width of each is
   // characterized by a number; sttBarWidi is this number for the
   // i-th pane
static Z sttBarWid2;
static Z sttBarWid3;
static Z sttBarWid4;

static Word relPos1;
static Word relPos2;
static Word relPos3;
static Word relPos4;
   //: relative position
   // relative directory positions, presently used only to locate
   // basic input file  cpmconfig.ini startng from the location
   // of the executable when starting from the command line) or
   // from the working directory as seen by an IDE one is working
   // with.

static Word inpDir;
   //: input directory
   // directory root read from cpmconfig.ini that is available for
```

```
      // defining interaction of a program with the file system of the
      // machine

    static Word fontDir;
      //: font directory
      // font location  read from cpmconfig.ini that is available for
      // defining interaction of a program with the file system of the
      // machine. So a C+- user is completely free where he places
      // the C+- font file on his computer.

    static Word runId;
      //: run ID
      // created prior to the creation of the log files

    static bool cerrExt;
      // If this is true, the filename cpmcerr.txt will be replaced
      // by "cpmcerr_"&runID&".txt". The value of cerrExt is 'false'
      // unless we have in cpmdependencies.h a line
      // #define CPM_CERR_EXTEND

    static bool appRunId;
      //: append run id
      // If this is true, the run ID will be appended to the names of
      // auto-created image files.

   static bool initialized(void){ return streamInitialized;}
      //

protected: // for usage by derived class Error

   static Z maxNumMes;
      // number of 'unenforced' messages to be written on out (helps to
      // avoid creating unintentionally huge text files and to slow
      // program run down

   static bool silent;
      // if this is true, the basic function communication runs idle
      // If it is set to true initially no log files will be created
      // and no possibility exists to swich it to false.
      // Thus setting CPM_NOLOG reliably says that there will no log
      // files be created during program run.

   static void communication(Word const&, Word const&, Word const&,
      bool, Z barSeg=1);

public:
// data
   static Z verbose;
      // controls the ammount of messages to be written to file in a way
      // that every message that writes under some value of
```

```
      // Message::verbose, will also be written under all higher values.

   static Z debug;
      // controls the behaviour of the cpmassert macro.
      // For debug==1 Message::warning will be executed
      // For debug==2 Message::error instead
      // In all other cases the asserted condition will not be tested

// functions
   static void ini(void);
      // one has not to call this function, but it can do no harm either,
      // helps experimentation in non-standard situations
   static Z pane1(){ return sttBarWid1;}
      //: pane 1
   static Z pane2(){ return sttBarWid2;}
      //: pane 2
   static Z pane3(){ return sttBarWid3;}
      //: pane 3
   static Z pane4(){ return sttBarWid4;}
      //: pane 4

   static Z w(){ return vpw_;}
      // tendative frame width in pixel
   static Z h(){ return vph_;}
      // tendative frame height in pixel
   static Z ws(){ return fsw_;}
      // tendative frame store width in pixel
   static Z hs(){ return fsh_;}
      // tendative frame store height in pixel

   static ofstream out;
      // writing via '>>' to Message::out is the primarily provided
      // (unidirectional) communication facility of the system.
      // After program run, one will find the text file cpmcerr.txt in
      // the execution directory that contains all what was written to
      // stream Message::out.

      // If CPM_USE_MPI is defined and the number np of processes is > 1
      // the program is aware of its rank and this will be appended
      // to the filenames cpmcerr and cpmdata in a way that
      // alphabetic order is the natural order. E.g. for np=64
      // we have cpmcerr_01.txt,...cpmcerr_09.txt, cpmcerr_10.txt, ...

      // The next program run will overwrite cpmcerr(_rank).txt. In most
      // cases this is OK in order not to pollute the working directory.
      // Sometimes it is better to retain the cpmcerr, cpmdata files
      // e.g. for comparing them for two related runs. Of course,
      // we may rename the files after creation in an informative way
      // to enable this. There is, however, also a automatic facility
      // of appending a 'time-stamp' to the file name. To enable this
```

```
   // mechanism one has to define in cpmsystemdependencies.h a macro
   // in the form
   // #define CPM_CERR_EXTEND
   // (no value associated with it). A small number p may be inputted
   // e.g. as p=4 in
   // #define CPM_RUN_ID_LENGTH 4
   // Then the time stamp consists of the last 4 digits of the date
   // string yyyymmddhhmmss. This thus always allows to diffeentiate
   // between runs for which the time of start differs by a second.
   // The larger the number p, the more complete the time-stamp reveals
   // the complete date.
    // For programs that make use of the application framework
   // given by class IniFileBasedApplication (see cpminifilebasapp.h)
   // there are more convenient documentation facilities controled
   // via IniFileBasedApplication::(sel, mode, runName).
   // Notice that CPM_CERR_EXTEND and CPM_RUN_ID_LENGTH  can not be
   // overwritten by cpmconfig.ini. This is natural since the concept
   // of a run ID asks for uniquness within a run and needs to be ready
   // before files cpmcerr and cpmdata are created. Since the class
   // RecordHandler makes heaviy use of cpmcerr for diagnostics
   // messages, ini-files cannot be opened before cpmcerr is made.
   // For a few months, I had a simple text file cpmconfig.txt that
   // allowed to input data from file prior to creation of cpmcerr.
   // This required rather complex programming logic and would break
   // the C+- uniform style of data input. Moreover, former files
   // cpmwindow.ini, cpmconfig.txt are now (2008-07-06) united into a
   // single cpmconfig.ini.

static ofstream outData;
   // writing via '>>' to Message::outData is a second
   // (unidirectional) communication facility of the system.
   // After program run, one will find the text file cpmdata.txt in
   // the execution directory that contains all what was written to
   // stream Message::outData

static void onStatusBar(Word const& w, Z barSegment=1);
   // stores w in the Word memory for 'pane'=barSegment
   // If possible for the system, it displays the Words
   // sttmes1,...sttmes4 on the status bar of the application
   // window. The status bar has several segments, numbered
   // 1,2,... and the second argument determines the field
   // on which the message shall be placed. Presently
   // there are 4 such fields. In CONSOLE applications, only
   // w is displayed on the console.
   // For barSegment<=0 no action is done

static void setMaxMes(Z mes=1000){maxNumMes=mes;}
   // setting the maximum number of (non-enforced) messages to be
   // written on out.
```

```
   // writing to the status bar

   static void progress(Word const& taskName, R frac, Z barSegment=1,
      Z field=3); // field =3 is usual
      // Displays task progress on the status bar.
      // for 0<=frac<=1 we get 000, 001, 002, ... 999, 1000
      // as a sufficient information on the degree of completion, only if
      // a digit changes there is change in the display. This is the
      // situation for field==3; for field=2 we go from
      // 00 to 100; etc. Only field=1,2,3,4,5 work this way; for any other
      // value one gets the same output as for field==4.

// writing to out

   static void report(Word const& text, Z val);
      // writes the number val on out if verbose>0

   static void report(Word const& text, R val);
      // writes the number val on out if verbose>0

// writing to both the status bar and out

   static void value(Word const& text, R val, Z barSegment=1);
      // writes the number val on the status bar and out if verbose>0

   static void values2(Word const& text, R val1, R val2, Z barSegment=1);
      // writes the numbers val1 and val2 on the status bar and out if
      // verbose>0

   static void values3(Word const& text, R val1, R val2, R val3,
      Z barSegment=1);

   static void values4(Word const& text, R val1, R val2, R val3,
      R val4, Z barSegment=1);

   static void value(Word const& text, Z val, Z barSegment=1);
         // writes the number val on the status bar and out if verbose>0

   Message(void);
      // initializes Message::out (if not done earlier)

   explicit Message(Word const& w);
      // initializes out (if not done earlier) and writes w to it

// Message and warning only differ in the wording of the output.
// Two Word arguments allow to specify the class in the scope of which
// the message is created. For barSeg<=0 no action on the status bar.
// Reason: Sometimes it is important to write some data
// to cpmcerr which would overload and disturb the status display
// on the status bar. If barSeg<0, the message goes to the
```

```
// console if we have one, even if CPM_NOGRAPHICS is not defined.

   static void warning(Word const& text, bool fileAlways=false);
      // A simple warning message function writing on out.
      // If fileAlways is false, a limitation of messages to
      // be written to file will take place to avoid slowing down program
      // execution too much

   static void warning(std::ostringstream const& ost,
      bool fileAlways=false)
   { Message::warning(Word(ost),fileAlways);}

   static void message(Word const& text, Z barSeg=1,
      bool fileAlways=false);

   static void message(std::ostringstream const& ost, Z barSeg=1,
      bool fileAlways=false)
   { Message::message(Word(ost),barSeg,fileAlways);}

   static void message(Word const& className, Word const& text,
      bool fileAlways=false);

   static void warning(Word const& className, Word const& text,
      bool fileAlways=false);

   static void message(Z mL, Word const& text, Z barSeg=1,
      bool fileAlways=false);
      // messaging takes place only if cpmverbose>=mL

   static void urgentMessage(Word const& text)
      // a message that will always be given
   {
      Z verbMem=verbose;
      verbose=1; message(text, 0, false); verbose=verbMem;
   }

   static void setSil(bool sil){ silent=sil;}
      // set silent

   static bool getSil(){ return silent;}

   static Word getRunId(){ return runId;}
      //: get run ID

   static bool getAppRunId(){ return appRunId;}
      //: get append run ID

   static void appToRunId(Word const& app){ runId&=app;}
      //: append to run ID
```

```
   static Word getInpDir(){ return inpDir;}
      //: get input directory

   static Word getFontDir(){ return fontDir;}
      //: get font directory

   static Word ext(Word const& w){ return glue(inpDir,w);}
      //: extend
      // Returns a file name which can be used to open a file.
      // For instance,
      //    IFileStream ifs(Message::ext("basics.txt"));
      //    Z sel; ifs>>sel;
      // reads an integer sel from file "basics.txt" which will be
      // searched in the directory determined by Message's conception
      // of a input directory. This allows different programs (with
      // different execution directories) access the same input data
      // for reliable performance comparisons. The different
      // programs need to mention in their respective files cpmconfig.ini
      // the same directory location as input directory.
      // It also allows to test a single program with various input
      // data sets (e.g. an experimental one and one already delivered
      // to a customer) by changing the content of the programs
      // cpmconfig.ini. This file may e.g. contain the lines
/*
      d:/cpm/pala
         // stable delivery data
      d:/cpm/palaX
         // experimental data
*/
   // Such entries between which one can swith by commentarization,
   // help to keep track of what one is doing.

};

void error(Word const& w);
void error(std::ostringstream const& ost);
void error(Word const& w1, Word const& w2);

//////////////////////////// class Error ////////////////////////////

class Error: public Message{ // used if C+- classes throw errors
    // No new data to Message, thus also a pure action class.
    // Since the action is determined by the constructor, the
    // functionality is new.
    // Inheritence allows to use the data and methods of Message to use
    // for implementation.

public:

explicit Error(Word const& messageText=Word("unspecified error"));
```

```
       // reporting a text in case of error, throws terminating exception

   Error(Word const& className, Word const& messageText);
       // reporting a class name and a text in case of error,
       // throws terminating exception
   friend void error(Word const& w);
   friend void error(std::ostringstream const& ost);
   friend void error(Word const& w1, Word const& w2);
};

inline void error(std::ostringstream const& ost)
   { CpmSystem::error(Word(ost));}

struct Exception{ // simple debugging tool
// instead of saying cpmmessage(" reached point 3"); one simply says
// Exception(3);
   Z i_;
   Exception(Z i):i_(i){ Message::message("Exception "&Word::write(i_));}
};

//////////////// class Timer ////////////////////////////////////////////

void wait(R t, Z pane=0);
      // delays program run for t seconds (especially to have time to
      // look to some screen information in absense of getchar() interupt
      // in Win32s programs. When the thread of control reaches a wait
      // statement it is not guaranteed that all graphical effects
      // created earlier on video memory can actually be seen on screen!
      // If the last argument is equal to one of the valid pane indexes
      // 1,2,3,4, the remaining time in seconds will be shown there.

R time(void);
      // returns the time at evaluation instant in seconds with respect
      // to a system dependent starting point. In applications only
      // differences have an un-abiguous meaning.
      // The starting point seems to be related to program start, and not
      // to date. Relies on system function clock()

class Timer{ // implements e.g. getSecondsLeft()
   R tBegin,tEnd;
   bool act;
public:
   Timer():tBegin(0),tEnd(0),act(false){}
   explicit Timer(R sp): tBegin(CpmSystem::time()),tEnd(tBegin+sp),
      act(true){}
      // sets tBegin to the time of function call
      // and tEnd to tBegin+sp. All times in seconds
   R getSecondsLeft()const;
      // returns the remaining span to tEnd expressed in seconds
      // For the default counter this function always returns 137*3600
```

```
   R getHoursLeft()const;
      // returns getSecondsLeft()/3600
   bool timeOut()const{ return getSecondsLeft()<=0.;}

};

//////////////////////// class IFileStream ////////////////////////////
//
// typical usage of file stream classes
// Word fIn("c:/d/cpm/myfile.txt");
// Word fOut("log.txt");
// IFileStream infi(fIn);
// OFileStream outfi(fOut);
// V<R> vr;
// vr.scanFrom(infi());
// vr.prnOn(outfi()); // notice '()' in using the streams
// In cpmfile.h there will be defined classes OFile and IFile which
// define assignment and copy.
// See also classes IFile and OFile in cpmfile.h for files which
// satisfy the strict value interface.

const bool binDef=true;
   // true is proven, false for experimentation
   // default value for setting the argument 'binary'
   // for opening streams. Introduced 2004-04-29

class IFileStream{ // input file stream

   ifstream* fs;
   typedef IFileStream Type;
   CPM_INVAR(IFileStream)

public:

   explicit IFileStream(Word const& fileName,
      bool safe=true, bool binary=binDef);
      // If the second argument is true one gets an error if the
      // file could not be opened, otherwise only a warning will
      // be issued.

   IFileStream():fs(0){}

   virtual ~IFileStream(){ fs->close(); delete fs;}

   char readChar(){ char res('x'); *fs>>res; return res;}
      // reading a char from stream

   string readWord(){ string res; *fs>>res; return res;}
      // reads the next contiguous group of non-whitespace characters
```

```
   string readLine(){ string res; std::getline(*fs,res); return res;}
      // reads the next line with all charcters also whitespace

   bool isVal()const{ return (*fs)!=0;}
      // returns stream status

   ifstream& operator()(void){return *fs;}
      // returns the object for direct access

   bool skipComments();
      // over-reads lines starting with '/', ';', '*', or '#'
};

//////////////////////// class OFileStream ////////////////////////////

class OFileStream{ // output file stream

   ofstream* fs;
   typedef OFileStream Type;
   CPM_INVAR(OFileStream)
public:

   explicit OFileStream(Word const& fileName,
      bool safe=true, bool binary=binDef);

   OFileStream():fs(0){}

   virtual ~OFileStream(){ fs->close();delete fs;}

   bool isVal()const{ return (*fs)!=0;}
      // returns stream status

   ofstream& operator()(void){return *fs;}
      // returns the object for direct access,
      // this is all one needs to write to the object
};

} // namespace

// An assert macro which is fully under the control of the C+- class
// system. The first argument should be a piece of C++ code which
// evaluates to bool. (Thus no C++-function can do the job since there
// are no variables that hold pieces of code)
// Example of usage:
//    cpmassert(x>0,"norm in function eval should be positive");
// notice that the final ';' is not needed

#define cpmassert(X,Y)\
if (CpmSystem::Message::debug==1){\
if (!(X)) CpmSystem::Message::warning(CpmRoot::Word(#X)&\
```

```
   " violated: "&Y);}\
else if (CpmSystem::Message::debug==2)\
{ if (!(X)) CpmSystem::error(CpmRoot::Word(#X)&" violated: "&Y);}

//////////////////// short names ////////////////////////////////////////
#define cpmcerr         CpmSystem::Message::out
#define cpmdata         CpmSystem::Message::outData
#define cpmverbose      CpmSystem::Message::verbose
#define cpmsilent       CpmSystem::Message::getSil
#define cpmdbg          CpmSystem::Message::debug
   // the name cpmdebug, which would be natural here, was defined
   // earlier, see cpmtypes.h
#define cpmmessage      CpmSystem::Message::message
#define cpmurgent       CpmSystem::Message::urgentMessage
#define cpmwarning      CpmSystem::Message::warning
#define cpmprogress     CpmSystem::Message::progress
#define cpmreport       CpmSystem::Message::report
#define cpmstatus       CpmSystem::Message::onStatusBar
#define cpmerror        CpmSystem::error
#define cpmwait         CpmSystem::wait
#define cpmvalue        CpmSystem::Message::value
#define cpmvalues2      CpmSystem::Message::values2
#define cpmvalues3      CpmSystem::Message::values3
#define cpmvalues4      CpmSystem::Message::values4
#define cpmtime         CpmSystem::time
#define cpmexc          CpmSystem::Exception

#endif // guard
```

200 cpmsystem.cpp

```
//? cpmsystem.cpp
//? Status of work 2008-10-25.
//?
#include <time.h>
#include <iomanip>

#include <cpmgreg.h>
#include <cpmsystem.h>
#include <cpmsystemdependencies.h>

#if !defined(CPM_NOGRAPHICS)
   #include <cpmviewport.h>
#endif

//////////////////// class Message ////////////////////////////////////
using namespace CpmStd;
using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::Word;
using CpmArrays::V;

namespace{
   const char sep='/';
   const char dot='.';

   bool startDot(Word const& file)
   {
      if (file.dim()==0) return false;
      return file[1]==dot;
   }

   bool startSep(Word const& file)
   {
      if (file.dim()==0) return false;
      return file[1]==sep;
   }

}

Word CpmSystem::glue(Word const& path, Word const& file)
{
// Do not insert diagnostic messages, like cpmdebug(temp); into
// this function block, since the function is used in Message::ini()
// before the messaging file cpmcerr exists.
   if (file.dim()==0) return path;
   if (path.dim()==0) return file;
   Word res= startSep(file) ? path&file : path&Word(sep)&file;
```

```
   if (startSep(res)) res=Word(dot)&res;
   return res;
}

//////////////////////// class Message ////////////////////////////////////
// initialization by code
Z CpmSystem::Message::verbose=2;
Z CpmSystem::Message::debug=2;
Z CpmSystem::Message::maxNumMes=100000;
Word CpmSystem::Message::sttmes1="*";
Word CpmSystem::Message::sttmes2="**";
Word CpmSystem::Message::sttmes3="***";
Word CpmSystem::Message::sttmes4="****";
Word CpmSystem::Message::runId="";
bool CpmSystem::Message::streamInitialized=false;
ofstream CpmSystem::Message::out;
ofstream CpmSystem::Message::outData;

// initialization from cpmsystemsependencies.h
Word CpmSystem::Message::relPos1=CPM_REL_POS1;
Word CpmSystem::Message::relPos2=CPM_REL_POS2;
Word CpmSystem::Message::relPos3=CPM_REL_POS3;
Word CpmSystem::Message::relPos4=CPM_REL_POS4;
Word CpmSystem::Message::inpDir=CPM_INP_DIR;
Word CpmSystem::Message::fontDir=CPM_FONT_DIR;

#if defined(CPM_NOLOG)
   bool CpmSystem::Message::silent=true;
#else
   bool CpmSystem::Message::silent=false;
#endif
bool CpmSystem::Message::appRunId=false;
#if defined(CPM_CERR_EXTEND)
   bool CpmSystem::Message::cerrExt=true;
#else
   bool CpmSystem::Message::cerrExt=false;
#endif

Z CpmSystem::Message::sttBarWid1=CPM_PANE1;
Z CpmSystem::Message::sttBarWid2=CPM_PANE2;
Z CpmSystem::Message::sttBarWid3=CPM_PANE3;
Z CpmSystem::Message::sttBarWid4=CPM_PANE4;

Z CpmSystem::Message::vph_=CPM_HEIGHT;
Z CpmSystem::Message::vpw_=CPM_WIDTH;
Z CpmSystem::Message::fsh_=CPM_HEIGHT;
Z CpmSystem::Message::fsw_=CPM_WIDTH;

// functions
```

```
void CpmSystem::Message::ini(void)
{
   if (initialized()) return;
   Z size=CpmMPI::Cpm_com.getSize();
   Z rank=CpmMPI::Cpm_com.getRank();
   Word loc("CpmSystem::Message::ini(void)");
   Word logFile("cpmcerr");
   Word datFile("cpmdata");
   CpmTime::Greg date;
   date.now();
#if defined(CPM_RUN_ID_LENGTH)
   Z runIdLength=CPM_RUN_ID_LENGTH;
#else
   Z runIdLength=6;
#endif
   runId=date.getCode().tail(runIdLength);
   if (size>1){
      // log files for various processes should be numbered
      Z w;
      if (size<=9) w=1;
      else if (size<=99) w=2;
      else if (size<=999) w=3;
      else w=0;
      Word wrd=cpmwrite(rank,w);
      logFile&=wrd;
      datFile&=wrd;
   }
   if (cerrExt){
      Word appendix="_";
      appendix&=runId;
      logFile&=appendix;
      datFile&=appendix;
   }
   logFile&=".txt";
   datFile&=".txt";
   out.open(-logFile, std::ios_base::out);
   outData.open(-datFile, std::ios_base::out);
   bool streamInitialized1=true;
   bool streamInitialized2=true;
   if (!out) streamInitialized1=false;
   if (!outData) streamInitialized2=false;
   if (streamInitialized1&&streamInitialized2){
      streamInitialized=true;
      date.writeFormatted(out);
      Word mes=loc&
         ": files "&logFile&" and "&datFile&" successfully created,\n";
      mes&="  rank="&cpm(rank)&", size="&cpm(size);
      cpmmessage(mes); // works
   }
   else{ // then we had trouble in creating the message streams
```

```
      if (streamInitialized1){ // since we really need only logFile, this
      // might be a situation not asking for stopping the program
         Word mes1=loc&": not able to create "&datFile;
         cpmwarning(mes1);
      }
      else{
         Word mes2=loc&": not able to create "&logFile&
            ", stopping program";
#if defined(_CONSOLE)
         if (rank==1) std::cout<<-mes2<<endl;
#endif
         throw;
            // something went fundamentally wrong, no message can be sent
      }
   }
}

CpmSystem::Message::Message(void)
{
   ini();
}

CpmSystem::Message::Message(Word const& w)
{
    ini();
    if (verbose>0) out<<endl<<w<<endl;
    if (!out) cpmerror("CpmSystem::Message::Message(...): out==0");
}

#if !defined(CPM_NOGRAPHICS)
void CpmSystem::Message::onStatusBar(Word const& w, Z i)
{
   static Z firstrun_=1;
   if (i<=0) return; // was i<0 till 2006-06-30
   //in this case we can’t write on the status bar before a graphical
   // window is available
   bool vii=CpmGraphics::Viewport::isInitialized();
   if (firstrun_ && vii){
      firstrun_=0;
      for (Z j=1; j<=4;j++) {
         CpmGraphics::Viewport::onStatusBar("pane "&cpm(j)&" active", j);
      }
      cpmwait(0.5);
   }
   if (vii) CpmGraphics::Viewport::onStatusBar(w, i);
}
#else
   void CpmSystem::Message::onStatusBar(Word const& w, Z i){}
#endif
```

```
void CpmSystem::Message::communication(Word const& classText,
   Word const& commType, Word const& commText, bool fileAlways,
   Z barSeg)
// common building block for messages and warnings, not for errors
// Dependence on preprocessor directives:
// CPM_USE_MPI : messaging done only in process 1 (0 in original style)
// _CONSOLE: everything written on cpmcerr.txt will also be written on
// cout
{
   if (silent) return; // this allows us to use C+- functions which
      // contain cpmmessage or cpmwarning statements in situations
      // in which creation of the log streams did not yet happen.
   static Z messageCounter=0;
   ini(); // esuring that 'out' can be used, no work load if already
   // initialized. For silent==true we never come here by message
   // calling and thus we are sure never to create the log files
   // as long as silent==true. If silent gets switched to false
   // (by code or whatever, next call to the present function will
   // call ini() and thus create the log files.
   Z eowMem=Word::getEndOfWord();
   Word::setEndOfWord(0);
      // then writing Words to a stream has the same effect as writing
      // string literals
   Z rank=CpmMPI::Cpm_com.getRank();
   rank; // to avoid warnings on 'not use' in all macro configurations
   if (verbose){ // only then, something has to be done
      messageCounter++;  // it is useful to count the messages
         // even if they are not restricted in number
      ostringstream ostStatus;
      ostStatus<<classText<<" "<<commType<<" "<<commText<<
      " # "<<messageCounter;
         // status bar should be written also if number of filed messages
         // is limited
      if (barSeg>0){
         onStatusBar(Word(ostStatus.str()),barSeg);
      }
      ostringstream ostFile;
      if (fileAlways || messageCounter<maxNumMes){
         ostFile<<endl<<"counter = "<<messageCounter<<endl;
         ostFile<<classText<<" "<<commType<<" "<<commText<<endl;
         R t=time(); ostFile<<"  "<<t<<" s after program start";
         out<<ostFile.str()<<endl<<std::flush;
#if defined(_CONSOLE)
         if (rank==1 && barSeg<0) std::cout<<ostFile.str()<<endl;
#endif
#if defined(CPM_NOGRAPHICS)&&defined(_CONSOLE)
         if (rank==1 && barSeg>0) std::cout<<ostFile.str()<<endl;
#endif
      }
      if (messageCounter==maxNumMes){
```

```
         ostFile<<endl<<
         "No more normal messages or warnings will be filed";
         out<<ostFile.str()<<endl;
#if defined(CPM_NOGRAPHICS)&&defined(_CONSOLE)
         if (rank==1) std::cout<<ostFile.str()<<endl;
#elif defined(_CONSOLE)
         if (rank==1 && barSeg<=0) std::cout<<ostFile.str()<<endl;
#endif
      }
   }
   Word::setEndOfWord(eowMem);
      // restoring the writing behavior of Word
}

void CpmSystem::Message::warning(Word const& errorText, bool fileAlways)
{
   static R waitTime=0.4;
   waitTime*=0.9;
      // if there are many warnings the run should not get slowed down
      // considerably
   communication("","Warning:",errorText, fileAlways);
   if (cpmverbose>=2) cpmwait(waitTime);
}

void CpmSystem::Message::message(Word const& text, Z barSeg,
   bool fileAlways)
{
   communication("","Message:",text,fileAlways,barSeg);
}

void CpmSystem::Message::message(Z mL, Word const& errorText,
   Z barSeg, bool fileAlways)
{
   if (cpmverbose>=mL)
   communication("","Message:",errorText, fileAlways,barSeg);
}

void CpmSystem::Message::warning(Word const& className,
   Word const& errorText, bool fileAlways)
{
   communication(className,"Warning:",errorText, fileAlways);
}

void CpmSystem::Message::message(Word const& className,
   Word const& errorText, bool fileAlways)
{
   communication(className,"Message:",errorText, fileAlways);
}

void CpmSystem::Message::progress(Word const& taskName, R frac, Z seg,
```

```
   Z field)
// new implementation (2005-04-17) asks for graphical action only if
// needed.
// This assumes that a specific segment on which progress display is
// active will not have to insert messages of different character.
// Actually the static memory elements should be associated with the
// segments directly and not only within a specific function such as
// 'progress'.
{
   if (silent) return;
   static Word mem1;
   static Word mem2;
   static Word mem3;
   static Word mem4;
   Word act1,act2,act3,act4;
   R fac2=1000;
   if (field==3) ;
   else if (field==1) fac2=10;
   else if (field==2) fac2=100;
   else if (field==4) fac2=10000;
   else if (field==5) fac2=100000;
   else field=3;
   bool write=false;
   Word act=cpm((Z)(frac*fac2),field);
   if (seg==1){
      if (act!=mem1){
         mem1=act;
         write=true;
      }
   }
   else if (seg==2){
      if (act!=mem2){
         mem2=act;
         write=true;
      }
   }
   else if (seg==3){
      if (act!=mem3){
         mem3=act;
         write=true;
      }
   }
   else if (seg==4){
      if (act!=mem4){
         mem4=act;
         write=true;
      }
   }
   if (write){ // graphical action only if needed
      write=false;
```

```
      Word mes=(verbose && seg==1) ? "Task: "&taskName&": Progress " :
         taskName&": ";
      mes&=act;
      if (verbose>2){
         message(mes,seg); // messages go always to the status bar
      }
      else{
         onStatusBar(mes,seg);
      }
   }
}

// shared code

#define CPM_SC\
   static const Z prec=4;\
   if (verbose){\
      ostringstream ost;\
      ost.precision(prec);\
      ost<<text<<" = ";\
      ost.width(8);\
      ost<<val;\
      if (verbose>2){\
         message(Word(ost.str()),seg);\
      }\
      else{\
         onStatusBar(Word(ost.str()),seg);\
      }\
   }

void CpmSystem::Message::value(Word const& text, R val, Z seg)
{
   CPM_SC
}

void CpmSystem::Message::values2(Word const& text, R val1, R val2, Z seg)
{
   static const Z prec=4;
   if (verbose){
      ostringstream ost;
      ost.precision(prec);
      ost<<text<<": ";
      ost.width(8);
      ost<<val1<<", "<<val2;
      if (verbose>2){
         message(Word(ost.str()),seg);
      }
      else{
         onStatusBar(Word(ost.str()),seg);
      }
```

```
   }
}

void CpmSystem::Message::values3(Word const& text, R val1, R val2,
   R val3, Z seg)
{
   static const Z prec=3;
   if (verbose){
      ostringstream ost;
      ost.precision(prec);
      ost<<text<<": ";
      ost.width(7);
      ost<<val1<<", "<<val2<<", "<<val3;
      if (verbose>2){
         message(Word(ost.str()),seg);
      }
      else{
         onStatusBar(Word(ost.str()),seg);
      }
   }
}

void CpmSystem::Message::values4(Word const& text, R val1, R val2,
   R val3, R val4, Z seg)
{
   static const Z prec=3;
   if (verbose){
      ostringstream ost;
      ost.precision(prec);
      ost.width(7);
      ost<<text<<": "<<val1<<", "<<val2<<", "<<val3<<", "<<val4;
      if (verbose>2){
         message(Word(ost.str()),seg);
      }
      else{
         onStatusBar(Word(ost.str()),seg);
      }
   }
}

void CpmSystem::Message::value(Word const& text, Z val, Z seg)
{
   CPM_SC
}

#undef CPM_SC

#define CPM_SC\
   if (verbose){\
      ostringstream ost;\
```

```
      ost<<text<<"="<<val;\
      message(Word(ost.str()));\
   }

void CpmSystem::Message::report(Word const& text, Z val)
{
   CPM_SC
}

void CpmSystem::Message::report(Word const& text, R val)
{
   CPM_SC
}
#undef CPM_SC

//////////////////////////// class Error ////////////////////////////////

// notice: base class constructor Message() called anyway

CpmSystem::Error::Error(Word const& messageText)
{
#if !defined(CPM_NOLOG)
   silent=false; // errors have to be reported
   verbose=1;
   communication("C+-",messageText,"CpmError",true);
#elif defined(_CONSOLE)
   cout<<endl<<"C+- "<<messageText<<" CpmError"<<endl;
#endif
   throw;  // errors should terminate
}

CpmSystem::Error::Error(Word const& className, Word const& messageText)
{
#if !defined(CPM_NOLOG)
   silent=false;
   verbose=1;
   communication(className,messageText,"CpmError",true);
#elif defined(_CONSOLE)
   cout<<endl<<"C+- "<<className<<" "<<messageText<<" CpmError"<<endl;
#endif
   throw;  // errors should terminate
}

///////////////// class Timer ////////////////////////////////////////

R CpmSystem::Timer::getSecondsLeft()const
   {
      static const R future=137*3600;
      if (!act) return future;
      R tAct=CpmSystem::time();
```

```
      return tEnd-tAct;
   }

R CpmSystem::Timer::getHoursLeft()const
{
   const R inv_h=1./3600.;
   return inv_h*getSecondsLeft();
}

//////////////////////// class-less functions ////////////////////////

void CpmSystem::error(Word const& w)
{
#if !defined(CPM_NOLOG)
   Message::ini();  // unclear whether needed
   Message::onStatusBar("Error",1);
   Message::onStatusBar(w,2);
#endif
   throw Error(w);
}

void CpmSystem::error(Word const& w1, Word const& w2)
{
#if !defined(CPM_NOLOG)
   Message::ini();
   Message::onStatusBar(w1,1);
   Message::onStatusBar(w2,2);
#endif
   throw Error(w1,w2);
}

void CpmSystem::wait(R t, Z pane)
{
   static bool firstrun=true;
   static R clInv=0;
   Z mL=3;
   if (firstrun){
      cpmmessage(mL,"CLOCKS_PER_SEC="&cpm(Z(CLOCKS_PER_SEC)));
      clInv=1./CLOCKS_PER_SEC;
      firstrun=false;
   }
   R tStart=clock()*clInv;
   R tEnd=tStart+t;
   R tLeft=tEnd-tStart;
   Z sLeft=cpmround(tLeft);
   while (tLeft>0)
   {
      tLeft=tEnd-clock()*clInv;
      if (pane>=1 && pane<=4){
         Z s0Left=cpmround(tLeft);
```

```
         if (s0Left<sLeft){
            sLeft=s0Left;
            cpmvalue(" tLeft",s0Left,pane);
         }
      }
   }
   return;
}

R CpmSystem::time(void)
{
   static R cpsInv=1./CLOCKS_PER_SEC;
   return cpsInv*clock();
}

////////////////// class IFileStream //////////////////////////
CpmSystem::IFileStream::IFileStream(Word const& fileName,
   bool safe, bool binary)
{
   fs=(binary? new ifstream(-fileName,std::ios_base::binary) :
      new ifstream(-fileName));
   if (!*fs){
      Word mes="IFileStream(Word,bool,bool): unable to open "&fileName;
      if (safe) cpmerror(mes);
      else cpmwarning(mes);
   }
   else{
      cpmmessage("IFileStream(Word,bool,bool) did open "&fileName);
   }
}

bool CpmSystem::IFileStream::skipComments()
{
   return CpmRoot::eatComments(*fs);
}

////////////////// class OFileStream //////////////////////////
CpmSystem::OFileStream::OFileStream(Word const& fileName,
   bool safe, bool binary)
{
   fs=(binary? new ofstream(-fileName,std::ios_base::binary) :
      new ofstream(-fileName));
   if (!*fs){
      Word mes="OFileStream(Word,bool,bool): unable to open "&fileName;
      if (safe) cpmerror(mes);
      else cpmwarning(mes);
   }
   else{
      cpmmessage("OFileStream(Word,bool,bool) did open "&fileName);
   }
```

```
}
```

201 cpmsystemdependencies.h

```
//? cpmsystemdependencies.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_SYSTEMDEPENDENCIES_H_
#define CPM_SYSTEMDEPENDENCIES_H_
/**********************************************************************
   cpmsystemdependencies.h
   Description: By this file one can place defines in implementation
    files in cases that the proper implementation is dependent on the
    computer system, e.g. an the access to a display or a file system.
    These defines may also modify the way who we use such facilities;
    e.g. using or not using the graphical display. Or, creating or
    not creating log files. Especially such changes of usage should
    not entail too much recompilation. So the present file should
    never be included in header files. Presently it is only included
    in six files: cpmnumbers.cpp, cpmsystem.cpp, cpmapplication.cpp
    cpmgraph.cpp, cpmimg24.cpp, cpminifilebasapp.cpp.

    Notice that changes in file cpmdefinitions.h trigger recompilation
    of virtually all C+- translation units.

    See also cpmconfig.ini, where most data can be overwritten.

**********************************************************************/

#define CPM_REL_POS1 "."
#define CPM_REL_POS2 "./control"
#define CPM_REL_POS3 ".."
#define CPM_REL_POS4 "../control"
   // four positions relative to the executables or the IDE's
   // idea of working directory. This is needed only for
   // finding cpmconfig.ini, on which the locations for further
   // input and for font files can be given

#define CPM_INP_DIR "."
   // default location of the directory for input
   // Typically not needed since the input directory can
   // comfortibly set in config.ini

#define CPM_FONT_DIR "c:/e/cpm/fonts"
   // location of the fonts file

#define CPM_HEIGHT 832
#define CPM_WIDTH  1432
   // screen data in pixels

#define CPM_PANE1 350
```

```
#define CPM_PANE2 250
#define CPM_PANE3 250
#define CPM_PANE4 100
   //relative size (arbitrary units) of the status bar subfields ('panes')

#define CPM_RUN_ID_LENGTH 4
   // number of digits of the run ID

#define CPM_WRITE_PRECISION 8
   // Numerical precision in writing floating point data to file

#define CPM_WRITE_TITLE
   // writing class instances to file will contain the
   // class name as 'title'

//#define CPM_CERR_EXTEND
   // if this is defined, the run ID will be appended
   // to the name of the log files

//#define CPM_NOLOG
   // if defined, no cpmcerr and cpmdata will be created

//#define CPM_NOGRAPHICS
   // if defined no graphical content will be shown on screen
   // and no graphics libraries (openGL and GLUT need to be linked

#endif
```

202 cpmsysviewer.h

```
//? cpmsysviewer.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_SYSVIEWER_H_
#define CPM_SYSVIEWER_H_

#include <cpminifilebasapp.h>
#include <cpmimagingtools.h>
#include <cpmcamera.h>

namespace CpmPalaAux{

//////////////////////// ViewerAppl ////////////////////////////

   using CpmApplication::IniFileBasedApplication;
   using namespace CpmRoot;
   using namespace CpmRootX;
   using namespace CpmImaging;
   using namespace CpmArrays;

class SysViewerAppl:public IniFileBasedApplication{// viewer application

   V<CpmCamera2::Camera> ca2_;
   V<CpmCamera3::Camera> ca3_;
   ShowControl sc_;

public:
   SysViewerAppl(Word const& w);
   V<Z> showMovie2();
   V<Z> showMovie3();
   void doTheWork();
      // calling showMovie in a suitable context
};
} // CpmPalaAux
#endif
```

203 cpmsysviewer.cpp

```
//? cpmsysviewer.cpp
//? Status of work 2008-10-25.
//?
#include <cpmsysviewer.h>
#include <cpmalgorithms.h>
#include <cpmsys2.h>
#include <cpmgreg.h>
#include <cpmframes.h>
#include <iomanip>

namespace CpmPalaAux{

using CpmArrays::V;
using CpmSystem::IFileStream;
using CpmTime::Greg;

using namespace CpmSystem;
using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmApplication;
using namespace CpmAlgorithms;
using namespace CpmProperties;
using namespace CpmGraphics;
using namespace CpmImaging;

V<Z> SysViewerAppl::showMovie2()
// all cameras get set to newColor=false so since otherwise the display
// is too slow and not all images get refreshed on equal footing.
{
#if defined(CPM_DIMDEF2)
   using namespace CpmPhysics2;
   using namespace CpmRigidBody2;
   using namespace CpmDim2;
   using namespace CpmPala2_2;
   Z mL=1;
   Word loc("showMovie2(...)");

   V<CpmCamera2::Camera> ca=ca2_;
   #include <cpmsysviewercpp.h>
#endif
   return V<Z>();
}

V<Z> SysViewerAppl::showMovie3()

{
#if defined(CPM_DIMDEF3)
```

```
   using namespace CpmPhysics3;
   using namespace CpmRigidBody3;
   using namespace CpmDim3;
   using namespace CpmPala2_3;
   Z mL=1;
   Word loc("showMovie3(...)");
   V<CpmCamera3::Camera> ca=ca3_;
   #include <cpmsysviewercpp.h>
#endif
   return V<Z>();
}

//////////////////// SysViewerAppl ////////////////////////////////

SysViewerAppl::SysViewerAppl(const Word& w):
   IniFileBasedApplication(w),sc_()
{
   const  Z mL=1;
   Word loc("SysViewerAppl("&w&")");
   cpmmessage(mL,loc&" started");
   Z i, ped=1, n=inFiles.dim();

   Word sec="syscontrol";
   SysControl(rch,sec,ped);

   sec="runcontrol";

   Z clearPeriod, imagingPeriod, viewingPeriod;
   rch.read(sec,"clearPeriod",clearPeriod);
   rch.read(sec,"imagingPeriod",imagingPeriod);
   rch.read(sec,"viewingPeriod",viewingPeriod);
   rch.read(sec,"firstFrame",sc_.offSet);
   rch.read(sec,"boxSel",sc_.boxSel);

   V<R> xRange,yRange,zRange;
   rch.read(sec,"xRange",xRange);
   rch.read(sec,"yRange",yRange);
   rch.read(sec,"zRange",zRange);
   sc_.ivx=Iv(xRange[1],xRange[2]);
   sc_.ivy=Iv(yRange[1],yRange[2]);
   sc_.ivz=Iv(zRange[1],zRange[2]);

   sc_.cyalClear=CyclicAlarm(clearPeriod);
   sc_.cyalImg=CyclicAlarm(imagingPeriod);
   sc_.cyalView=CyclicAlarm(viewingPeriod);

   rch.read(sec,"tWait",sc_.tWait);
   rch.read(sec,"modality",sc_.modality);

   V<Z> frameNumbers;
```

```
   rch.read(sec,"frameNumbers",frameNumbers);
   sc_.nocyalImg=NonCyclicAlarm(frameNumbers);

   rch.read(sec,"maxNumberImages",sc_.maxNumberImages);

// reading two Zs-lists in a way that they get ignored if they contain
// a single 0. This is OK since valid values have to be >0.
   sc_.invisibleParticles=V<Z>(0);
   V<Z> invisibleParticles(0);
   bool succ=rch.read(sec,"invisibleParticles",invisibleParticles,ped);
   if (succ){
      if (!(invisibleParticles.dim()==1&&invisibleParticles[1]==0))
         sc_.invisibleParticles=invisibleParticles;
   }

   sc_.listColorize=V<Z>(0);
   V<Z> listColorize(0);
   succ=rch.read(sec,"listColorize",listColorize,ped);
   if (succ){
      if (!(listColorize.dim()==1&&listColorize[1]==0))
         sc_.listColorize=listColorize;
   }

// now the cameras get initialized from the ini-file
   sec="cameracontrol";
   rch.read(sec,"autoFrameFactor",sc_.autoFrameFactor);
   Frame f0;
   R fill=(n==1 ? 1.0 : 0.98);
   Frames frs(f0,n,fill);
  // setModeFromInFiles(); experiment
   if (mode_==2){
#if defined(CPM_DIMDEF2)
   ca2_=V<CpmCamera2::Camera>(n);
   for (i=1;i<=n;i++) ca2_[i]=CpmCamera2::Camera(rch,sec,frs(i));
#endif
   }
   else if (mode_==3){
#if defined(CPM_DIMDEF3)
   ca3_=V<CpmCamera3::Camera>(n);
   for (i=1;i<=n;i++) ca3_[i]=CpmCamera3::Camera(rch,sec,frs(i));
#endif
   }
   else{
      cpmerror("CpmViewer::SysViewerAppl(...): unvalid mode_");
   }
   cpmmessage(mL,loc&" done");
}

void SysViewerAppl::doTheWork()
{
```

```
   const  Z mL=1;
   cpmmessage(mL,"SysViewerAppl::doTheWork() started");
   V<Z> count(0);

   if (mode_==2){
#if defined(CPM_DIMDEF2)
      count=showMovie2();
#endif
   }
   else if (mode_==3) {
#if defined(CPM_DIMDEF3)
      count=showMovie3();
#endif
   }
   else{
      cpmerror("CpmViewer::SysViewerAppl::doTheWork(): unvalid mode_");
   }
   cpmdata<<"movie frames found: "<<endl;
   count.prnOn(cpmdata);
   cpmmessage(mL,"SysViewerAppl::doTheWork() done");
}

} // CpmPalaAux
```

204 cpmsysviewercpp.h

```
//? cpmsysviewercpp.h
//? Status of work 2008-10-25.
//?
/*
   cpmsysviewercpp.h
   code for inclusion into different namespace contexts.
   Common definition of the functions SysViewerAppl::showMovie2
   and SysViewerAppl::showMovie3
*/

   CPM_MA
   Frame frv;
   frv.paint(LIGHTGREEN);
   frv.display();
  // Record rc=rch.getRecord();
   Z nca=ca.dim();
   Z nfn=inFiles.dim();
   if (nca!=nfn)
      cpmerror("showMovie(V<Camera>,V<Word>,Z ): dimension mismatch");

   Z i;
   cpmmessage(mL,"Number of movies shown simultaneously: "&cpm(nfn));
   V<ExtrSys2> extr(nfn);
   // Notice that cpmsys2x.h made the definition
   // typedef ParExt<ParCT,ParVT,ParT,ExtSys,Sys2Base> ExtrSys2;
   // and also gives definitions of ParCT, ParVT, and ParT.
   cpmmessage(loc&" initializing Extr started");
   for (i=1;i<=nfn;i++){
      Word ifi=Message::ext(inFiles[i]);
  //    Word ofi=Message::ext(outFiles[i]);
      cpmdebug(ifi);
    //  cpmdebug(ofi);
      extr[i]=ExtrSys2(ifi);
   }
   cpmmessage(mL,loc&" initializing Extr done");

   V<ShowControl> shco(nfn,sc_);
   for (i=1;i<=nfn;i++) shco[i].imgFile=inFiles[i];
   V<Z> count(nfn);
   bool b=true;
   while(b){
      for (i=1;i<=nfn;i++){
         b=extr[i].next();
         if (b==false) break;
         count[i]=extr[i].getCount();
      }
      if (b==false) break;
```

```
   for (i=1;i<=nfn;i++){
      extr[i].show(ca[i],shco[i]);
         // this is the essential graphical activity
         // see cpmextractorx.h for definition
      bool bi=extr[i].isVal();
      if (bi==false) b=false;
   }
}
CPM_MZ;
return count;
```

205 cpmtestr.h

```
//? cpmtestr.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_TEST_R_H_
#define CPM_TEST_R_H_

/*
Purpose: New uniform test facility for all
classes which implement the rich interface and also for
other interfaces and classes

*/

#include <cpmtypes.h>
#include <cpmc.h>
#include <cpmvr.h>
#include <cpmsr.h>
#include <cpmfr.h>
#include <cpmp.h> // for Vp
#include <iomanip> // for std::setprecision

namespace CpmTests{

   using namespace CpmStd;

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::Word;
   using CpmArrays::V;
   using CpmArrays::Vo;
   using CpmRootX::B;
   using CpmSystem::IFileStream;
   using CpmSystem::OFileStream;
   using std::setprecision;

template <class T>
class Sym{ // testing symmetry of equality
public:
   Sym(){}
   bool operator()(T const& t1, T const& t2)
      // T == T is symmetric if this function returns true for all
      // input arguments (then we say: 'sym==true')
   {
      if (t1==t2) return (t2==t1); else return true;
   }
};
// Typical usage;
```

```
// T t1...;
// T t2...;
// Sym<T> s;
// bool b=s(t1,t2); // b ==true says that symmetry has been confurmed
// for the particular objects t1 and t2.


template <class T>
class Trans{ // testing transitivity of equality
public:
   Trans(){}
   bool operator()(T const& t1, T const& t2, T const& t3)
// T == T is an equivalence relation if sys==true and trans==true
   {
      if (t1==t2 && t2==t3) return (t1==t3); else return true;
   }
};

template <class T>
class DefaultConstructor{ // testing consistency of default constructor
public:
   DefaultConstructor(){}
   bool operator()(T const& t)
   // value of  t not used, carries type information
   {
      t; // to avoid warning concerning not making use of the argument
      T t1; T t2; return (t1==t2);
   }
};

template <class T>
class CopyConstructor{ // testing consistency of copy constructor
public:
   CopyConstructor(){}
   bool operator()(T const& t)
   {
      T t1(t); return t1==t;
   }
};

template <class T>
class Assignment{ // testing consistency of assignement
public:
   Assignment(){}
   bool operator()(T const& t1, T const& t2)
   {
      T tStart(t1);
      tStart=t2;
      return tStart==t2;
   }
```

```
};

template <class T>
class StrictAssignment{ // testing strict consistency of assignement
public:
   StrictAssignment(){}
   bool operator()(T const& t1, T const& t2, T const& t3)
   {
      T tStart(t1);
      T t2c(t2);
      tStart=t2c;
      T tMem(tStart);
      t2c=t3; // source for tStart's value re-defined
      return tStart==tMem; // tests change of tStart
   }
};

// Let T be a type for which the statement
// strictAssignment<T> sa; compiles then we say that the
// T implements the strict assignment on compilation
// If we find sa(t1,t2,t3)==true for all T typed argument triplets
// based on the logic of the code we say that T implements the strict
// assignment.
// If we veryfy sa(t1,t2,t3)==true by running a program with sufficiently
// many arguments we may say that the strict assignment was tested with
// positive result.

// if for a fixed triplet of types T1, T2, C this
// function is true for all arguments, we says that C is a
// polymorphic single-slot-container for T1 and T2

template <class T1, class T2, class C>
class PolymorphicSingle{ // testing polymorphic containers
public:
   PolymorphicSingle(){}
   bool operator()(const T1& t1, const T2& t2, const C& c )
   {
      C cc(c);
      cc()=t1;
      T1 tc1(cc());
      bool b1=(t1==tc1);
      cc()=t2;
      T2 tc2(cc());
      bool b2=(t2==tc2);
      return b1 && b2;
   }
};

template <class T1, class T2, class C>
class PolymorphicMulti{ // testing polymorphic containers
```

```
public:
   PolymorphicMulti(){}
   bool operator()(const T1& t1, const T2& t2, const C& c, Z i1, Z i2 )
   {
      if ( i1>=1 && i1<=c.dim() && i2>=1 && i2<=c.dim() && i1!=i2 ){
         C cc(c);
         cc[i1]=t1;
         cc[i2]=t2;
         T1 tc1(cc[i1]);
         T2 tc2(cc[i2]);
         return t1==tc1 && t2==tc2;
      }
      else return true;
   }
};

// if for a fixed triplet of types T1, T2, C this
// function is true for arguments, we says that c is a
// polymorphic multi-slot-container for T1 and T2

// We will see that Vp<T> is a polymorphic multi-slot-container for all
// types T1, T2 equal to or derived from T.


template <class T>
class TestBase{ // base class for classes which test that class T
// implements a certain interface
protected:
   R errorSum;

   static void wrt_(ostringstream& ost){
      Word wd=Word(ost.str());
      wd.prnOn(cpmcerr);
      wd.prnOn(cout);
   }

   static void wrt_(Word const& wd){
      wd.prnOn(cpmcerr);
      wd.prnOn(cout);
   }

   static R comp(Word top, T const& lhs, T const& rhs);
      // means for comparing results
public:
   TestBase():errorSum(0.){}
   R val()const{return errorSum;}
};

template <class T>
R TestBase<T>::comp(Word top, T const& lhs, T const& rhs)
```

```
{
   using namespace CpmSystem;
   const R tiny=1e-4;

   R al=CpmRoot::absVal<T>(lhs);
   R ar=CpmRoot::absVal<T>(rhs);
   Word w1="Warning: trivial argument in ";
   Word w2=top;
   ostringstream ost;
   if (al==0. || ar==0.){
      ost<<endl<<(w1+w2);
   }
   R newError=CpmRoot::disVal<T>(lhs,rhs);
   ost<<endl<<"Test of "
      <<top<<endl<<" error = "<<newError<<endl
      <<"cpmabs(lhs)= "<<al<<endl
      <<"cpmabs(rhs)= "<<ar<<endl
     ;
   if (newError>tiny){
     ost<<endl<<" lhs="<<endl;
     CpmRoot::prnOn<T>(lhs,ost);
     ost<<endl<<" rhs="<<endl;
     CpmRoot::prnOn<T>(rhs,ost);
   }
   wrt_(ost);
   return newError;
}

////////////// testing the value interface ////////////////////////////

template <class T>
class Test_v: public TestBase<T>{ // testing the value interface

   typedef TestBase<T> Base;

public:
   Test_v(Z complexity=1, Z rep=4);
};

template <class T>
Test_v<T>::Test_v(Z tvs, Z rep):Base()
      // Tests all relations concerning the value interface and returns
      // an error sum
{
   using namespace CpmRoot;
   using namespace CpmRootX;

   const R dSmall=1e-4;
   const R dTiny=1e-12;
```

```
   Word top;
   Word loc("Test_v<T>::Test_v(Z tvs, Z rep)");
   R res=0;
   Z counter=0;
   T xd,x;
   T xt=xd.testVal(tvs);
   while (counter++<rep){
      x=xt.ranVal();
      cpmassert(xt.disVal(x)>dSmall,loc); // the random generator should
      // generate
      // new values, if it does so only after many iterations it is
      // probably not reliable, so we expect this to be valid for
      // sufficiently small
      // dSmall
      T y=xt.ranVal();
      cpmassert(y.disVal(x)>dSmall,loc);
      T z=xt.ranVal();
      cpmassert(z.disVal(x)>dSmall,loc);
      cpmassert(z.disVal(y)>dSmall,loc);
      R r1=y.disVal(z);
      R r2=z.disVal(y);
      cpmassert(cpmabs(r1-r2)<dTiny,loc);

// now x,y,z are supposed to be sufficiently random to make test
// meaningful
      top="is copy equal to original ?";
      T xl(z);
      res+=comp(top,xl,z);
      top="is assigned value equal to original ?";
      y=x;
      res+=comp(top,y,x);
// This was all to be shown
      xt=z; // to start the new loop were the old one ended
   }
   ostringstream ost;
   ost<<endl<<"  Test of Value Interface; errorsum= "<<res<<endl;
   Base::wrt_(ost);
   Base::errorSum=res;
}

///////////// testing the strict value interface ///////////////////////

template <class T>
class Test_sv: public TestBase<T>{ // testing the strict value interface

   typedef TestBase<T> Base;

public:
   Test_sv(Z complexity=1, Z rep=4);
};
```

```
template <class T>
Test_sv<T>::Test_sv(Z tvs, Z rep):Base()
      // Tests all relations concerning the strict value interface and
      // returns an error sum
{
   using namespace CpmRoot;
   using namespace CpmRootX;

   const R dSmall=1e-4;
   const R dTiny=1e-12;

   Word top;
   Word loc("Test_sv<T>::Test_sv(Z tvs, Z rep)");
   R res=0;
   Z counter=0;
   T xd,x;
   T xt=xd.testVal(tvs);
   while (counter++<rep){
      x=xt.ranVal();
      cpmassert(xt.disVal(x)>dSmall,loc); // the random generator should
      // generate new values, if it does so only after many iterations it
      // is probably not reliable, so we expect this to be valid for
      // sufficiently small dSmall
      T y=xt.ranVal();
      cpmassert(y.disVal(x)>dSmall,loc);
      T z=xt.ranVal();
      cpmassert(z.disVal(x)>dSmall,loc);
      cpmassert(z.disVal(y)>dSmall,loc);
      R r1=y.disVal(z);
      R r2=z.disVal(y);
      cpmassert(cpmabs(r1-r2)<dTiny,loc);
      T zr=z;

// now x,y,z are supposed to be sufficiently random to make test
// meaningful
      top="is copy equal to original ?";
      T xl(zr);
      z=z.ranVal();
      z=z.ranVal(); // although zr was made from z, this change in z
         // should not affect zr, thus zr and xl have still to be equal
      res+=comp(top,xl,zr);

      top="is assigned value equal to original ?";
      y=x;
      T x1=x;
      T x2=y;
// messing up the originals; x1 and x2 should remain unchanged
      x=x.ranVal();
      x=x.ranVal();
```

```
      y=y.ranVal();
      y=y.ranVal();
      res+=comp(top,x1,x2);
// This was all to be shown
      xt=z; // to start the new loop were the old one ended
   }
   ostringstream ost;
   ost<<endl<<" Test of Strict Value Interface; errorsum="<<res<<endl;
   Base::wrt_(ost);
   Base::errorSum=res;
}

/////////// testing the r-interface ////////////////////////////////////

template <class T, class S>
class Test_r: public TestBase<T>{ // testing the r-interface
   // class for testing classes that
   // implement the r-interface
   // Testing e.g. Vr<R> is done e.g. by the following code
   // Test_r<Vr<R>,R> a(10);

   typedef TestBase<T> Base;

public:
   Test_r(Z complexity=1 , Z output=0, Z write=0);
};

template <class T, class S>
Test_r<T,S>::Test_r(Z tvs, Z output, Z write):Base()
      // Tests all relevant mathematical relations and returns
      // an error sum
      // output 0 : only end of test and error sum displayed
      // output 1 : output of every test. The potentially
      //      time consuming output of the test object is suppressed
      // output 2: only the first component of the test object is
      // displayed
      // output 3: the test object is displayed in full detail
      // This form only works for classes (not for the types R and Z. As
      // a compensation it shows the favorable property of not having to
      // know the namespace of T, since all access is intrinsic.
{

   using namespace CpmSystem;
   using namespace CpmFunctions;
   using CpmRoot::nameOf;
   static const Z multOrder=4;
      // order test is made on an array of
      // multOrder*tvs elements

   Z nOrd=multOrder*tvs;
```

```
T t, v, x, y, z, xl, xr;
Word top;

R res=0.;
ostringstream ost;
ost<<endl<<" Test of class "<<t.nameOf()<<endl;
Base::wrt_(ost);

if (output){
   cpmcerr<<endl<<
   " The distinguished objects associated with the class are: ";
}

T xd;

T xzero=xd.net(0);

if (output){
   cpmcerr<<endl<<"zero:";
   xzero.toWord().prnOn(cpmcerr);
}

T xunit=xd.net(1);
if (output){
   cpmcerr<<endl<<"unit:";
   xunit.toWord().prnOn(cpmcerr);
}

T xt=xd.testVal(tvs);

v=xt;
x=xt.ranVal(1);
y=xt.ranVal(2);
z=xt.ranVal(3);
T xWrite=z;

if (output){
   cpmcerr<<endl<<"testVal:";         xt.toWord().prnOn(cpmcerr);
   cpmcerr<<endl<<"first ranVal:";  x.toWord().prnOn(cpmcerr);
   cpmcerr<<endl<<"second ranVal:"; y.toWord().prnOn(cpmcerr);
   cpmcerr<<endl<<"third ranVal:";  z.toWord().prnOn(cpmcerr);
}
T v4=x;
T v5;
T v6=y;
v6=v5=x;
xl=v6;
xr=v4;
top="copy constructor, assignement, and equality";
```

```
     res+=comp(top,xl,xr);
   xl=x+y; xr=y+x; top="+ commutativity";
   res+=comp(top,xl,xr);
   xl=(x+y)+z; xr=x+(y+z); top="+ associativity";
   res+=comp(top,xl,xr);
   xl=(x*y)*z; xr=x*(y*z); top="* associativity";
   res+=comp(top,xl,xr);

   T xZero=x.net(0);

   xl=-x; xr=xZero-x; top="sign change";
   res+=comp(top,xl,xr);
   xl=x*(y+z); xr=x*y+x*z; top="left-distributivity";
   res+=comp(top,xl,xr);
   xl=(x+y)*z; xr=x*z+y*z; top="right-distributivity";
   res+=comp(top,xl,xr);
   // not clear why con(x*y) is not understood for T=Fr<Z,R>
   T xTemp=x*y;
   xl=xTemp.con(); xr=y.con()*x.con();
   top="conjugation commutativity";
   res+=comp(top,xl,xr);
   x=z;
   T xu=x;

   xu=xu.net(1);
   if (output){
      cpmcerr<<endl<<"unit=";
      xu.toWord().prnOn(cpmcerr);
   }

   if (output){
      cpmcerr<<endl<<"x=";
      x.toWord().prnOn(cpmcerr);
   }

   T xi=x.inv();

   if (output){
      cpmcerr<<endl<<"xInv=";
      xi.toWord().prnOn(cpmcerr);
   }

   xl=x*xi; xr=xu; top="right inverse";
   res+=comp(top,xl,xr);

   xl=xi*x; xr=xu; top="left inverse";
   res+=comp(top,xl,xr);
// properties with scalars
   S s;
   s=CpmRoot::testVal<S>(s,tvs);
```

```
S sZero=CpmRoot::net<S>(s,0);
S sUnit=CpmRoot::net<S>(s,1);
S sInit=sUnit+sUnit+sUnit;
sInit+=sInit;
Z count=0;
S s1=CpmRoot::ranVal<S>(sInit,count);
const Z countMax=100;
while (s1==sZero || s1==sUnit){
   s1=CpmRoot::ranVal<S>(sInit,++count);
   if (count>countMax){
      cpmwarning(
      "Test_r<T,S>: too many iterations (bad random generator of S)"
      );
      break;
   }
}
if (output){
   Word mes="iterations needed to get s1: "&cpmwrite(count);
   cpmcerr<<endl<<mes;
}
count=0;
S s2=CpmRoot::ranVal<S>(s1,++count);
while (s2==sZero || s2==sUnit || s2==s1){
   s2=CpmRoot::ranVal<S>(s1,++count);
   if (count>countMax){
      cpmwarning(
      "Test_r<T,S>: too many iterations (bad random generator of S)"
      );
      break;
   }
}
if (output){
   Word mes="iterations needed to get s2: "&cpmwrite(count);
   cpmcerr<<endl<<mes;
}
xl=s1*(x+y); xr=s1*x+s1*y; top="*s left-distributivity";
res+=comp(top,xl,xr);
xl=(x+y)*s2; xr=x*s2+y*s2; top="*s right-distributivity";
res+=comp(top,xl,xr);
S s12=s1*s2; // workaround:
   // not clear why xl=(s1*s2)*x; gets not compiled in some cases
xl=s12*x; xr=s1*(s2*x); top="s*s left-distributivity";
res+=comp(top,xl,xr);
xl=x*(s1*s2); xr=(x*s1)*s2; top="s*s right distributivity";
res+=comp(top,xl,xr);

xl=x; xl*=sUnit; xl=xl/sUnit; xr=(x*s1); xr/=s1;
top="/s and *s compatibility";
res+=comp(top,xl,xr);
```

```
// order properties
// rather simple test of ordering consistency

   T xMax=CpmRootX::sup(x,y,z);
   T xMin=CpmRootX::inf(x,y,z);

   if ( xMin.priorTo(xMax) || xMin.equalTo(xMax)  ){
      cpmcerr<<endl<<"comparison test OK"<<endl;
      cout<<endl<<"comparison test OK"<<endl;
   }
   else{
      cpmcerr<<endl<<"comparison test failed"<<endl;
      cout<<endl<<"comparison test failed"<<endl;
      res+=1;
   }

// let us order a list of random values
   CpmArrays::Vo<T> list(nOrd);
   Z i;
   for (i=1;i<=nOrd;i++){
      list[i]=xt.ranVal(i);
   }
   list.order();
   for (i=2;i<=nOrd;i++){
      if ( list[i].priorTo(list[i-1]) ){
         cpmmessage("comparison failure");
         cout<<endl<<"comparison failure";
         res+=1;
      }
   }

   if (write){
      cpmmessage("Writing path entered");
      Word filename(nameOf(xWrite));
      filename=filename.makeFileName();
      Word filename1="w-"+filename+".txt";
      OFileStream out1(filename1);
      if (!out1()) cpmerror(nameOf(xWrite),
         "test: output file cannot be opened");
      const int precWrite=12;
      out1()<<setprecision(precWrite);
      out1()<<endl<<"// I/O Test:"<<endl;
      out1()<<"// Object to be written: "<<endl;
      xWrite.prnOn(out1());
      cpmmessage("First writing done");
      if (!out1()) cpmerror(nameOf(xWrite),
         "out1 not OK before closing");
      IFileStream in(filename1);
      if (!in()) cpmerror(nameOf(xWrite),
         "test: input file cannot be opened");
```

```
      xr.scanFrom(in());
      cpmmessage("First reading done");
      if (!in()) cpmerror(nameOf(xWrite),
         "in not OK after reading and before closing");
      Word filename2="r-"+filename+".txt";
      OFileStream out2(filename2);
      if (!out2()) cpmerror(nameOf(xWrite), "test: out2 file cannot be\
       opened");
      out2()<<"// I/O Test. This is what we got back:"<<endl;
      xr.prnOn(out2());
      cpmmessage("Second writing done");
      if (!out2()) cpmerror(nameOf(xWrite),
         "out2 not OK before closing");
       xl=xWrite; top="file I/O";
      res+=comp(top,xl,xr);
      cpmmessage("Leaving writing path");
   }

   ostringstream ost2;
   ost2<<endl<<"  Test of "<<nameOf(xWrite)<<" errorsum= "<<res<<endl;
   Base::wrt_(ost2);
   Base::errorSum=res;
}

template <class T>
class Test_c: public TestBase<T>{ // testing complex classes
   typedef TestBase<T> Base;
public:
   Test_c(Z complexity=1 , Z output=0, Z write=0);
};


template <class T>
Test_c<T>::Test_c(Z tvs, Z output, Z write):Base()
{
 //  write; // to avoid warning
   static const Z multOrder=4;
      // order test is made on an array of
      // multOrder*tvs elements

   Z nOrd=multOrder*tvs;

   T  x, y, z, xl, xr, xt;

   ostringstream ost;
   ost<<endl<<" Test of class "<<x.nameOf()<<endl;
   Base::wrt_(ost);

   Word top;
   xt=x.testVal(tvs);
```

```
x=xt.ranVal(1);
y=xt.ranVal(2);
z=xt.ranVal(3);
T xWrite=z;

if (output){
   cpmcerr<<endl<<"testVal:"<<xt.toWord();
   cpmcerr<<endl<<"first ranVal:"<<x.toWord();
   cpmcerr<<endl<<"second ranVal:"<<y.toWord();
   cpmcerr<<endl<<"third ranVal:"<<z.toWord();
}

T v4=x;
T v5;
T v6=y;
v6=v5=x;
xl=v6;
xr=v4;

R res=0;
top="copy constructor, assignement, and equality";
  res+=comp(top,xl,xr);
xl=x+y; xr=y+x; top="+ commutativity";
res+=comp(top,xl,xr);
xl=x*y; xr=y*x; top="* commutativity";
res+=comp(top,xl,xr);
xl=(x+y)+z; xr=x+(y+z); top="+ associativity";
res+=comp(top,xl,xr);
xl=(x*y)*z; xr=x*(y*z); top="* associativity";
res+=comp(top,xl,xr);

T xZero=net(x,0);

xl=-x; xr=xZero-x; top="sign change";
res+=comp(top,xl,xr);
xl=x*(y+z); xr=x*y+x*z; top="left-distributivity";
res+=comp(top,xl,xr);
xl=(x+y)*z; xr=x*z+y*z; top="right-distributivity";
res+=comp(top,xl,xr);
// not clear why con(x*y) is not understood for T=Fr<Z,R>
T xTemp=x*y;
xl=con(xTemp); xr=con(y)*con(x); top="conjugation commutativity";
res+=comp(top,xl,xr);

x=z;
T xu=x;

xu=net(xu,1);
T xi=inv(x);
```

```
   xl=x*xi; xr=xu; top="right inverse";
   res+=comp(top,xl,xr);

   xl=xi*x; xr=xu; top="left inverse";
   res+=comp(top,xl,xr);

// order properties
// rather simple test of ordering consistency

   T xMax=CpmRootX::sup(x,y,z);
   T xMin=CpmRootX::inf(x,y,z);

   if (xMin<=xMax){
      cpmcerr<<endl<<"comparison test OK"<<endl;
   }
   else{
      cpmcerr<<endl<<"comparison test failed"<<endl;
      res+=1;
   }

// let us order a list of random values
   Vo<T> list(nOrd);
   Z i;
   for (i=1;i<=nOrd;i++){
      list[i]=ranVal(xt,i);
   }
   list.order();
   for (i=2;i<=nOrd;i++){
      if (list[i-1]>list[i]){
         cpmmessage("comparison failure");
         res+=1;
      }
   }
   xl=z;
   R r,phi;
   xl.toPolar(r,phi);
   xr.polar(r,phi);
   top="polar back and forth";
   res+=comp(top,xl,xr);

   xl=x; xr=x.sqrt()*x.sqrt();
   top="sqrt and square";
   res+=comp(top,xl,xr);

   xl=exp(x+y); xr=exp(x)*exp(y);
   top="exp addition law";
   res+=comp(top,xl,xr);

   xl=x; xr=exp(ln(x));
   top="exp(ln(z))=z";
```

```
res+=comp(top,xl,xr);

xl=sin(x+y); xr=sin(x)*cos(y)+cos(x)*sin(y);
top="sin addition law";
res+=comp(top,xl,xr);

xl=y; xr=sin(arcsin(y));
top="sin(arcsin(z)=z";
res+=comp(top,xl,xr);

xl=z; xr=cot(arccot(z));
top="cot(arccot(z)=z";
res+=comp(top,xl,xr);

Z n=3;
xl=pow(cosh(x)+sinh(x),n);
xr=cosh(n*x)+sinh(n*x);
top="Moivre for cosh and sinh";
res+=comp(top,xl,xr);
if (write){
   cpmmessage("Writing path entered");
   Word filename(nameOf(xWrite));
   Word filename1="w-"+filename+".txt";
   OFileStream out1(filename1);
   if (!out1()) cpmerror(nameOf(xWrite),
      "test: output file cannot be opened");
   const int precWrite=12;
   out1()<<setprecision(precWrite);
   out1()<<endl<<"// I/O Test:"<<endl;
   out1()<<"// Object to be written: "<<endl;
   out1()<<xWrite;
   cpmmessage("First writing done");
   if (!out1()) cpmerror(nameOf(xWrite),
      "out1 not OK before closing");
   IFileStream in(filename1);
   if (!in()) cpmerror(nameOf(xWrite),
      "test: input file cannot be opened");
   xr.scanFrom(in());
   cpmmessage("First reading done");
   if (!in()) cpmerror(nameOf(xWrite),
      "in not OK after reading and before closing");
   Word filename2="r-"+filename+".txt";
   OFileStream out2(filename2);
   if (!out2()) cpmerror(nameOf(xWrite),
      "test: out2 file cannot be opened");
   out2()<<"// I/O Test. This is what we got back:"<<endl;
   out2()<<xr;
   cpmmessage("Second writing done");
   if (!out2()) cpmerror(CpmRoot::nameOf(xWrite),
      "out2 not OK before closing");
```

```
       xl=xWrite; top="file I/O";
      res+=comp(top,xl,xr);
      cpmmessage("Leaving writing path");
   }
   Base::errorSum=res;
   ostringstream ost2;
   ost2<<endl<<"  Test of "<<nameOf(xWrite)<<" errorsum= "<<res<<endl;
   Base::wrt_(ost2);
}

/*
template <class T>
class Test_lin: public TestBase<T>{ // class for testing classes that
// implement the r-interface
 //  R errorSum;
//   static R comp(Word top, T const& lhs, T const& rhs);
      // means for comparing results
   typedef TestBase<T> Base;
public:
   Test_lin(Z complexity=1 , Z output=1, Z write=0);
 //  R val()const{return errorSum;}
};

template <class T>
Test_lin<T>::Test_lin(Z tvs, Z output, Z write):Base()
      // Tests all relevant mathematical relations and returns
      // an error sum
      // output 0 : only end of test and error sum displayed
      // output 1 : output of every test. The potentially
      //      time consuming output of the test object is suppressed
      // output 2: only the first component of the test object is
      // displayed
      // output 3: the test object is displayed in full detail
{
   static const Z multOrder=32;
      // order test is made on an array of
      // multOrder*tvs elements

   Z nOrd=multOrder*tvs;

   T t, v, x, y, z, xl, xr;
   Word top;

   R res=0.;

   ostringstream ost;
   ost<<endl<<" Test of class "<<nameOf(t)<<endl;
   Base::wrt_(ost);

   if (output) cpmcerr<<endl<<
```

```
   " The distinguished objects associated with the class are: ";

   T xd;
   T xt=testVal(xd,tvs);

   v=xt;
   x=ranVal(xt,1);
   y=ranVal(xt,2);
   z=ranVal(xt,3);
   T xWrite=z;

   if (output){
      cpmcerr<<endl<<"testVal:"<<xt;
      cpmcerr<<endl<<"first ranVal:"<<x;
      cpmcerr<<endl<<"second ranVal:"<<y;
      cpmcerr<<endl<<"third ranVal:"<<z;
   }
   T v4=x;
   T v5;
   T v6=y;
   v6=v5=x;
   xl=v6;
   xr=v4;
   top="copy constructor, assignement, and equality";
     res+=comp(top,xl,xr);
   xl=x+y; xr=y+x; top="+ commutativity";
   res+=comp(top,xl,xr);
   xl=(x+y)+z; xr=x+(y+z); top="+ associativity";
   res+=comp(top,xl,xr);

// order properties
// rather simple test of ordering consistency

   T xMax=CpmRootX::sup(x,y,z);
   T xMin=CpmRootX::inf(x,y,z);

   if (xMin<=xMax){
      cpmcerr<<endl<<"comparison test OK"<<endl;
   }
   else{
      cpmcerr<<endl<<"comparison test failed"<<endl;
      res+=1;
   }

// let us order a list of random values
   Vo<T> list(nOrd);
   Z i;
   for (i=1;i<=nOrd;i++){
      list[i]=ranVal(xt,i);
   }
```

```
   list.order();
   for (i=2;i<=nOrd;i++){
      if (list[i-1]>list[i]){
         cpmmessage("comparison failure");
         res+=1;
      }
   }

   if (write){
      cpmmessage("Writing path entered");
      Word filename(nameOf(xWrite));
      Word filename1="w-"+filename+".txt";
      OFileStream out1(filename1);
      if (!out1()) cpmerror(nameOf(xWrite),
         "test: output file cannot be opened");
      const int precWrite=12;
      out1()<<setprecision(precWrite);
      out1()<<endl<<"// I/O Test:"<<endl;
      out1()<<"// Object to be written: "<<endl;
      out1()<<xWrite;
      cpmmessage("First writing done");
      if (!out1()) cpmerror(nameOf(xWrite), "out1 not OK before closing")\
      ;
      IFileStream in(filename1);
      if (!in()) cpmerror(nameOf(xWrite),
         "test: input file cannot be opened");
      read(xr,in());
      cpmmessage("First reading done");
      if (!in()) cpmerror(nameOf(xWrite), "in not OK after reading and\
       before closing");
      Word filename2="r-"+filename+".txt";
      OFileStream out2(filename2);
      if (!out2()) cpmerror(nameOf(xWrite), "test: out2 file cannot be\
       opened");
      out2()<<"// I/O Test. This is what we got back:"<<endl;
      out2()<<xr;
      cpmmessage("Second writing done");
      if (!out2()) cpmerror(nameOf(xWrite), "out2 not OK before closing")\
      ;
      xl=xWrite; top="file I/O";
      res+=comp(top,xl,xr);
      cpmmessage("Leaving writing path");
   }
   Base::errorSum=res;
   ostringstream ost2;
   ost2<<endl<<"  Test of "<<xWrite.nameOf()<<" errorsum= "<<res<<endl;
   Base::wrt_(ost);
}
*/
```

```
// class for testing consistent polymorphic behavior
template <class T, class Td, class Container> // assumes that Td is
// derived from T
//
class TestOfPolymorphism{ // testing consistent polymorphic behavior
   public:
   TestOfPolymorphism(T const& t, const Td& td)
   {
       using namespace CpmSystem;
       Word loc("TestOfPolymorphism(T const& t, const Td& td)");
       Z tv=16;
       Container ct;
       T tMem=t;
       Td tdMem=td;
       ct.set(t);
       T t1=ct();
       cpmassert(t1==tMem,loc);
       ct.set(td);
       Td td1=ct();
       R eps=td1.disVal(tdMem);
       cpmmessage("TestOfPolymorphism() successfully done");
       cout<<endl<<"TestOfPolymorphism() done with eps="<<eps<<endl;
    }
};

////////////////////////////// Test_Vp ////////////////////////////////
// testing polymorphicity and value interface of Vp<>

template <class T0, class T1, class T2, class T3>
class Test_Vp: public TestBase<T0>{// testing behavior of Vp<>

   typedef TestBase<T0> Base;

public:
   Test_Vp(Z complexity=1 , Z output=1);
};

template <class T0, class T1, class T2, class T3 >
Test_Vp<T0,T1,T2,T3>::Test_Vp(Z complexity , Z output):Base()
      // Tests all relations concerning polymorphic array Vp<> and
      // returns an error sum
{
   using namespace CpmRoot;
   using namespace CpmRootX;
   using namespace CpmArrays;

   Vp<T0> v0;

   ostringstream ost;
   ost<<endl<<" Test of class "<<v0.nameOf()<<endl;
```

```
Base::wrt_(ost);

T3 t3;
t3=t3.testVal(complexity);
T3 t3_1=t3.ranVal(1);
T3 t3_2=t3.ranVal(2);
T3 t3_3=t3.ranVal(3);
T3 t3_4=t3.ranVal(4);
T3 t3_5=t3.ranVal(5);
T3 t3_6=t3.ranVal(6);

T0 t0_1=t3_1.toClnBase();
T0 t0_2=t3_2.toClnBase();
T0 t0_3=t3_3.toClnBase();
T0 t0_4=t3_4.toClnBase();
T0 t0_5=t3_5.toClnBase();
T0 t0_6=t3_6.toClnBase();

T1 t1_1=T1(t0_1);
T1 t1_2=T1(t0_2);
T1 t1_3=T1(t0_3);
T1 t1_4=T1(t0_4);
T1 t1_5=T1(t0_5);
T1 t1_6=T1(t0_6);

T2 t2_1=T2(t0_1);
T2 t2_2=T2(t0_2);
T2 t2_3=T2(t0_3);
T2 t2_4=T2(t0_4);
T2 t2_5=T2(t0_5);
T2 t2_6=T2(t0_6);

Z nvp=12;
Vp<T0> vp(nvp);
V<Word> wp(nvp);
vp[1]=t0_1;
vp[2]=t1_2;
vp[3]=t2_3;
vp[4]=t3_4;
vp[5]=t0_5;
vp[6]=t1_6;
vp[7]=t2_1;
vp[8]=t3_2;
vp[9]=t0_3;
vp[10]=t1_4;
vp[11]=t2_5;
vp[12]=t3_6;

wp[1]=t0_1.nameOf();
wp[2]=t1_2.nameOf();
```

```
   wp[3]=t2_3.nameOf();
   wp[4]=t3_4.nameOf();
   wp[5]=t0_5.nameOf();
   wp[6]=t1_6.nameOf();
   wp[7]=t2_1.nameOf();
   wp[8]=t3_2.nameOf();
   wp[9]=t0_3.nameOf();
   wp[10]=t1_4.nameOf();
   wp[11]=t2_5.nameOf();
   wp[12]=t3_6.nameOf();

// complicated process of creating an array Vp<T0> zp holding the same
// values
// as vp
   Vp<T0> xp; // default element
   xp=vp; // assignment to default element
   Vp<T0> yp(xp); // copy construction
   Vp<T0> zp(5001);
   zp.set(t0_6); // zp something different
   zp=yp; // assignment to something non-trivial

   R errSum=0;
   Z i;
   for (i=xp.b();i<=xp.e();++i){ // testing whether finally zp holds the
   // right objects
      // here only the name gets tested
      Word wpi=zp(i).nameOf();
      if (output) cout<<endl<<wpi;
      errSum+=wp[i].disVal(wpi);
   }

   Base::errorSum=errSum;
   ostringstream ost2;
   ost2<<endl<<" Test of class "<<v0.nameOf()<<" done."
   <<endl<<" Errorsum= "<<errSum;
   Base::wrt_(ost2);
}

//////////////////////////////// Test_F ////////////////////////////////////
// Testing the chaining operation & of F<>

template <class T1, class T2, class T3, class T4>
class Test_F: public TestBase<T1>{ // Testing the chaining operation of F
   typedef TestBase<T1> Base;
public:
   Test_F(Z complexity=1 );
};

template <class T1, class T2, class T3, class T4 >
Test_F<T1,T2,T3,T4>::Test_F(Z complexity):Base()
```

```
{
   using namespace CpmRoot;
   using namespace CpmRootX;
   using namespace CpmFunctions;

   Fr<T1,T2> fa;
   Fr<T2,T3> fb;
   Fr<T3,T4> fc;

   Word mes(" Test of classes ");
   mes&=(fa.nameOf()&", "&fb.nameOf()&", and "&fc.nameOf());
   Base::wrt_(mes);

   fa=fa.testVal(complexity);
   fb=fb.testVal(complexity);
   fc=fc.testVal(complexity);

   F<T1,T2> fa_1=fa.ranVal(1);
   F<T1,T2> fa_2=fa.ranVal(2);

   F<T2,T3> fb_1=fb.ranVal(3);
   F<T2,T3> fb_2=fb.ranVal(4);

   F<T3,T4> fc_1=fc.ranVal(5);
   F<T3,T4> fc_2=fc.ranVal(6);

   F<T1,T3> fab=fa_1&fb_2;
   F<T2,T4> fbc=fb_2&fc_1;

   Fr<T1,T4> fab_c=fab&fc_1;
   Fr<T1,T4> fa_bc=fa_1&fbc;

   Fr<T1,T4> fake=fa_2&fbc;

   R errSum=0;
   errSum+=fab_c.disVal(fa_bc);
   R dis=fab_c.disVal(fake); // should not be small
      // this detects programming bugs that let disVal always return 0.
   R tiny=1e-6;
   if (dis<tiny) errSum+=1;

   Base::errorSum=errSum;
   mes&=(" done.\n Errorsum="&cpm(Base::errorSum));
   Base::wrt_(mes);
}

//////////////////////////////// Test_set ////////////////////////////////

template <class S, class T> // S is the class of sets of type T;
// typically S=Sr<T>
```

```
class Test_set: public TestBase<S>{ // testing behavior of sets

   typedef TestBase<S> Base;

public:
   Test_set(Z complexity=1 , Z output=1, Z write=0);
};

template <class S, class T>
Test_set<S,T>::Test_set(Z complexity , Z output, Z write):Base()
      // Tests all relations concerning the value interface and returns
      // an error sum
{
   using namespace CpmRoot;
   using namespace CpmRootX;
   using namespace CpmArrays;

   const R dSmall=1e-4;
   const R dTiny=1e-12;

   T t;
   S s,s1,s2,s3,s4;
   S lhs, rhs;

   ostringstream ost;
   ost<<endl<<" Test of class "<<s.nameOf()<<endl;
   Base::wrt_(ost);

   s=s.testVal(complexity);
   t=testVal<T>(t,complexity);
   s1=s.ranVal(1);
   s2=s.ranVal(2);
   s3=s.ranVal(3);
   s4=s.ranVal(4);
   s1=s1|s4;
   s2=s2|s4;
   s3=s3|s4;
      // this achieves that not all sections become trivial
   S xWrite=s3;
   if (output){
      cpmcerr<<endl<<"testVal:";          s.prnOn(cpmcerr);
      cpmcerr<<endl<<"first ranVal:";  s1.prnOn(cpmcerr);
      cpmcerr<<endl<<"second ranVal:"; s2.prnOn(cpmcerr);
      cpmcerr<<endl<<"third ranVal:";  s3.prnOn(cpmcerr);
   }

   R res=0.;
   Word top;

   idem1(lhs,rhs,s1);
```

```
top="idem1";
res+=comp(top,lhs,rhs);

idem2(lhs,rhs,s2);
top="idem2";
res+=comp(top,lhs,rhs);

comm1(lhs,rhs,s1,s2);
top="comm1";
res+=comp(top,lhs,rhs);

comm2(lhs,rhs,s1,s2);
top="comm2";
res+=comp(top,lhs,rhs);

assoc1(lhs,rhs,s1,s2,s3);
top="assoc1";
res+=comp(top,lhs,rhs);

assoc2(lhs,rhs,s1,s2,s3);
top="assoc2";
res+=comp(top,lhs,rhs);

distr1(lhs,rhs,s1,s2,s3);
top="distr1";
res+=comp(top,lhs,rhs);

distr2(lhs,rhs,s1,s2,s3);
top="distr2";
res+=comp(top,lhs,rhs);

diffdistr1(lhs,rhs,s1,s2,s3);
top="diffdistr1";
res+=comp(top,lhs,rhs);

diffdistr2(lhs,rhs,s1,s2,s3);
top="diffdistr2";
res+=comp(top,lhs,rhs);

S xl,xr;

if (write){
   cpmmessage("Writing path entered");
   Word filename(nameOf(xWrite));
   filename=filename.makeFileName();
   Word filename1="w-"+filename+".txt";
   OFileStream out1(filename1);
   if (!out1()) cpmerror(nameOf(xWrite),
      "test: output file cannot be opened");
   const int precWrite=12;
```

```
      out1()<<std::setprecision(precWrite);
      out1()<<endl<<"// I/O Test:"<<endl;
      out1()<<"// Object to be written: "<<endl;
      xWrite.prnOn(out1());
      cpmmessage("First writing done");
      if (!out1()) cpmerror(nameOf(xWrite),
         "out1 not OK before closing");
      IFileStream in(filename1);
      if (!in()) cpmerror(nameOf(xWrite),
         "test: input file cannot be opened");
      xr.scanFrom(in());
      cpmmessage("First reading done");
      if (!in()) cpmerror(nameOf(xWrite),
         "in not OK after reading and before closing");
      Word filename2="r-"+filename+".txt";
      OFileStream out2(filename2);
      if (!out2()) cpmerror(nameOf(xWrite),
         "test: out2 file cannot be opened");
      out2()<<"// I/O Test. This is what we got back:"<<endl;
      xr.prnOn(out2());
      cpmmessage("Second writing done");
      if (!out2()) cpmerror(nameOf(xWrite),
         "out2 not OK before closing");
       xl=xWrite; top="file I/O";
      res+=comp(top,xl,xr);
      S xc=static_cast<S>(xl|xr).condense();
      cout<<endl<<"condensed union";
      xc.prnOn(cout);


      cpmmessage("Leaving writing path");
   }
   Base::errorSum=res;
   ostringstream ost2;
   ost2<<endl<<" Test of class "<<s.nameOf()<<" done."
   <<endl<<" Errorsum= "<<res;
   Base::wrt_(ost2);

}

//////////////////// class Test_logic<> ////////////////////////////////
// Test of classes modeling propositional logic
// So far implemented: cpmlogic1.h, classes PropLog<>, PropLogic

template <class T>
class Test_logic: public TestBase<T>{//test models of propositional logic

   typedef TestBase<T> Base;

public:
```

```
   Test_logic(Z complexity=1 , Z output=1, Z write=0);

};

template <class T>
Test_logic<T>::Test_logic(Z n, Z output, Z write):Base()
      // Tests all relevant mathematical relations and returns
      // an error sum
      // output 0 : only end of test and error sum displayed
      // output 1 : output of every test. The potentially
      //      time consuming output of the test object is suppressed
      // output 2: only the first component of the test object is
      // displayed
      // output 3: the test object is displayed in full detail
{
   Z i;
   if (n<3) n=3; // we need as many variables to make indicative examples

   V<T> p=T::list(n);
   T p1=p[1];
   for (i=2;i<=n;i++) p1=p1.et(p[i]);
   p1=p1.non();
   T p2=p[1].non();
   for (i=2;i<=n;i++) p2=p2.vel(p[i].non());
   bool b=p1.iff(p2).taut();
   cpmmessage("b="&B(b).toWord()&"; true is correct");

// example to the commutativity of the diagram shown in
// cpmlogic1.h for class PropLog<X>

   T q1=p[1].non().et(p[2]).vel(p[3]);
   T q2=p[1].vel(p[2]);
   B b1(q1.eval());
   B b2(q2.eval());

   T q12a=q1.et(q2);
   B b12a=b1.et(b2);
   B b12a_(q12a.eval());
   B resa(b12a==b12a_);
   cpmmessage("b12a==b12a_ ?"&resa.toWord()&"; true is correct");

   T q12o=q1.vel(q2);
   B b12o=b1.vel(b2);
   B b12o_(q12o.eval());
   B reso(b12o==b12o_);
   cpmmessage("b12o==b12o_ ?"&reso.toWord()&"; true is correct");

   T q1n=q1.non();
   B b1n=b1.non();
   B b1n_(q1n.eval());
```

```
   B resn(b1n==b1n_);
   cpmmessage("b1n==b1n_ ?"&resn.toWord()&"; true is correct");

/************** takes too long so far ***********
   Z nc=3;
   V<T> q=T::list(nc);
   T xorq=T::xorV(q);
   V<T> known(2);
   known[1]=xorq;
   known[2]=q[nc];
   V<T> aux=T::list(nc-1);
   for (i=1;i<nc;i++) aux[i]=q[i].non();
   T hyp=T::andV(aux);
   B resx(hyp.implied(known));
   cpmmessage("resx ? "&resx.toWord()&"; true is correct");
*/
}

} // namespace

#endif
```

206 cpmtrj.h

```
//? cpmtrj.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_TRJ_H_
#define CPM_TRJ_H_
/*
   Description: tools for a unified description of evolution
   resulting from an initial condition
*/

#include <cpmobservable.h>

namespace CpmQuantumMechanics{

   using CpmRootX::R1;
   using CpmRootX::B;
   using CpmRootX::Record;
   using CpmAlgorithms::PeriodicAlarm;
   using CpmGraphics::Graph;
   using CpmLinAlg::Z4;

//////////////////////// struct TrjCon ////////////////////////////

struct TrjCon{ // trajectory control
// Here I use the uninitialized native C types (instead of B,R1,Z1)
// since these presently cooperate better with class Record

   bool autoAdj_;
      // auto-adjust (graphics to curve size)
      // If autoAdj (auto-adjust) is true, the y-range gets reset
      // if needed to let the state not leave the range of the graphics
      // window. If secFac_ is only a bit larger than 1, this adjustment
      // will take place often, which could be annoying; so something
      // like
      // secFac_=1.25 should be chosen.
   bool autoStep_;
   // if autoStep_==false, dt_ is the time step that will be applied
   // to the state of *this. For autoStep_==true the time step to be
   // done will be computed (under control of QMBase::phiStep),
   // and not taken as the present dt_
   R dt_;
      // the prescribed time step of the trajectory that
      // is considered valid only if autoStep_==false
   R tTot_;
      // total time of the run, after the elapse of which it
      // considered finished. Dependent quantity
   bool enfNor_;
```

```
      // enforce normalization (of state to 1) after each
      // integration step
   bool ignPhs_;
      // ignore phases, i.e. only the absolute value of the wave
      // function (state) will be represented graphically
   Z nSteps_;
      // number of steps that make up the trajectory
   Z disMode_;
      // display mode
      // selects the graphical representation mode
   R secFac_;
      // security factor e.g. 1.25 controls with which
      // limits the graphics frame adapts to the curve.
      // if 1 adustment is done with any step that
      // changes the curve range by whatsoever small an
      // ammount
   R tWait_;
      // wait time for showing frames in function TrjB::run(Graph)
      // So better rather short such as 0.02
   Z vwnPer_;
      // viewing period. Only each vwnPer_'th computed step will
      // be shown on screen
   Z filPer_;
      // filing period. Only each filPer_'th computed step will
      // be written to file
   Z imgPer_;
      // filing period. Only each imgPer_'th computed step will
      // be written to image (bitmap)
   R dph1_,dph2_;
      // parameters controling an optional dephasing mechanism
   Z dphMethod_;
      // method parameter controling an optional dephasing mechanism
      // If this is 0, no dephasing is active

// parameters which define coarse graining of states after a dynamical
// step
   Z corGrnMet_;
      //: coarse graining method
      // Presently only one method which is switched off if this
      // parameter is zero
   Z nPhi_;
      // number of phi-values
      // all phases of state components will be changed to the
      // nearest of nPhi_ equi-spaced and equi-distributed
      // discrete values
      // see State::corGrn(Z,Z), used as first argument there
   Z nBins_;
      // number of bins
      // see State::corGrn(Z,Z), used as second argument there
   R gam_;
```

```
      //: gamma
      // display gamma
   Z intMet_;
      //: interpolation method
      // 0 for pixelization
      // 1 for bilinear interpolation

   typedef TrjCon Type;
   CPM_IO
   CPM_NAM(TrjCon)
   TrjCon():dphMethod_(0),corGrnMet_(0),dph1_(0),dph2_(0),
      gam_(1),intMet_(0){}
   TrjCon(Record const& rc);
      // constructor from objects that can conveniently be initialized
      // from text
};

//////////////////////// TrjBase ////////////////////////////////////////
// Basis for trajectory classes i.e. classes dealing with the
//'initial value problem'.
// In order to be able to hold various derivatives of this class
// as components of polymorphic arrays of type Vp<TrjBase>
// (see class Trj2) it is important to define the virtual function clone.

class TrjBase{ // basis for trajectory classes
   friend class Trj2;
protected:
    TrjCon tc_;
      // trajectory control
    Z1 d_;
      // dimension of state
    Z1 stepCon_;
      // step count, initialized as 0
    R1 t_,dt_;
      // time and time step. Notice t_ initialized as 0.
    R1 n_1_;
      // norm of the wavefunction (state) - 1
    C enr_;
      // energy (may due to integrator particularities have a small
      // imaginary component

    PeriodicAlarm vwn_;
    PeriodicAlarm fil_;
    PeriodicAlarm img_;

    CyclicAlarm vwns_;
    CyclicAlarm fils_;
    CyclicAlarm imgs_;

    void adjustBase_()
```

```
    {
       if (tc_.dt_< 0) tc_.autoStep_=true;
         // this should not apply to tc_.dt_==0 since in the presence of
         // dephasing methods also dt_==0 may be an interesting situation
       vwn_=PeriodicAlarm(tc_.vwnPer_*tc_.dt_);
       fil_=PeriodicAlarm(tc_.filPer_*tc_.dt_);
       img_=PeriodicAlarm(tc_.imgPer_*tc_.dt_);
       vwns_=CyclicAlarm(tc_.vwnPer_);
       fils_=CyclicAlarm(tc_.filPer_);
       imgs_=CyclicAlarm(tc_.imgPer_);
       tc_.tTot_=tc_.nSteps_*tc_.dt_;
       dt_=tc_.dt_; // experiment
    }
    virtual void adjust_(){adjustBase_();}
public:
   typedef TrjBase Type;
   virtual TrjBase* clone()const{ return new TrjBase(*this);}
   CPM_IO_V
   CPM_NAM(TrjBase)
   TrjBase(){}
   TrjBase(TrjCon const& tc):tc_(tc){
       cpmmessage("calling constructor of TrjBase");
       adjustBase_();
       }
      // not calling ajust_() here since the call of final adjust_
      // will be made prior to first serious usage of any instance
      // of this class or a derived class

   virtual void step_(R h){}

   Z dim()const{ return d_;}

   virtual State getRep()const{ return State(); }
      // the return type is State which here is used as the
      // adequate data type supporting visualization

   void showAsLine(Graph& g, Z4 const& z4)const
      { getRep().show(g,z4[1],z4[2],z4[3],z4[4]); g.dis();}

   virtual void showAsArea(Graph& g)const{cpmmessage("showAsArea");}

   virtual Word text()const{ return Word("TrjBase");}

   R t()const{ return t_;}
   R1& t(){ return t_;}
   // Cygwin g++ 3.4.4 does not allow to use dt_ directly in class TrjDMI
   // which is derived from a class which in turn is derived from the
   // present class.
   // (although all derivations are public). Functions  c a n  be used,
   // only with explicit scope resolution.
```

```
   // I don't agree with this rape of the C++ spirit.

   R dt()const{ return dt_;}
   R1& dt(){ return dt_;}

   R sup()const{ return getRep().sup();}

   R n_1()const{ return n_1_;}
      // should suggest norm - 1
      // The object the norm of which has to be taken is not
      // yet available in this class

   C enr()const{ return enr_;}
   C& enr(){ return enr_;}

   TrjCon trjCon()const{ return tc_;}

   bool vwnAlarm_()
   { if (tc_.dt_>0) return vwn_(t_); else return vwns_();}
      // the logic is: if tc_.dt_ has a useful value it will
      // be used (together with nSteps_) to determine a time span,
      // that controls function run,  otherwise all control is
      // based on nSteps_ alone.

   bool filAlarm_()
   { if (tc_.dt_>0) return fil_(t_); else return fils_();}

   bool imgAlarm_()
   { if (tc_.dt_>0) return img_(t_); else return imgs_();}

   bool keepGoing()const
      // Notice that we ideally plan the points in time
      // the passing of which should trigger the various actions
      // for which the counters take care. In this form the actual
      // triggering depends on numerical chance events. The shift
      // by tol avoids this.
   {
      const R tolFac=0.25;
      R tol=tolFac*dt_;
      if (tc_.dt_>0) return t_+tol <= tc_.tTot_;
      else return stepCon_<tc_.nSteps_;
   }

   Z stepCon()const{ return stepCon_;}
   Z1& stepCon(){ return stepCon_;}

// when trajectories get written to file and then retreived from there
// one should be able to create easily the diagrams concerning
// norm-1 and eTot that get greated at the end of the run-functions.
// The most systematic method would be to store system state sufficiently
```

```
// completely so that the values of n1_() and eTot() can be computed
// from them. Unfortunately these data are quite different in the various
// derived classes: in TrjSR they include even the huge (pre-computed)
// constant object so of type SpectralObject. So, in a sense we fake in
// this place and write and read these dependent quantities to and from
// which gives the read-function an ad hoc argument structure.

   virtual bool fileState_(ostream& str){ return false;}

   virtual bool readState_(istream& str){ return false;}

   bool autoStep()const{ return tc_.autoStep_;}
      // says whether the auto stepping option is valid

   void setAutoStep_(bool b){ tc_.autoStep_=b;}

   void runAsLine(Graph& g, ostream& str);
      // Running n steps of length n and representing this as a color
      // image.
      // The run begins with x0, i.e. t=0 and displays in succession
      // the states at times 0,dt,2*dt,......n*dt. These are (n+1) states
      // which correspond to n time steps

   virtual void runAsArea(Graph& g, ostream& str);
      // represents the wave function as a colored d_ times d_ image
      // which is natural only for two-particle states of FieldSpace.
      // Here it is the most detailed visualization of the state that
      // allows one to get a feeling for the difference of fermionic
      // and bosonic wave function evolution.

   void runAsCur(Graph& g, ostream& str);
      // running tc_.nSteps steps of length n and representing this as a
      // series of curves. Time range as in previous function.
      // Each frame is at least shown for a time of waitTime seconds.
      // The y-range is set so that if the initial state would be
      // multiplied by secFac (security factor) it would fill the frame.
      // So setFac>1 is needed to see the whole state *this.
      // Implemented in this class, see cpmtrj.cpp
      // Shows finally norm and energy, very instructive!

   void run(Graph& g, ostream& str)
   {
      adjust_();
      if (tc_.disMode_==1) runAsCur(g,str);
      else if (tc_.disMode_==2) runAsLine(g,str);
      else runAsArea(g,str);
   }

   void playAsLine(Graph& g, istream& str);
   void playAsArea(Graph& g, istream& str);
```

```
   void playAsCur(Graph& g, istream& str);

   void play(Graph& g,istream& str)
   {
      if (tc_.disMode_==1) playAsCur(g,str);
      else if (tc_.disMode_==2) playAsLine(g,str);
      else playAsArea(g,str);
   }
};

//////////////////////// TrjT<> ////////////////////////////////////////
// Trajectory class which adds to TrjBase the essential state
// descriptor x_ and thus allows a final implementation of
// some of the virtual functions in TrjBase

template <class StateT>
class TrjT: public TrjBase{//adds to TrjBase the state descriptor x_
protected:
    StateT x_;
      // state
    virtual void adjust_()
    {
       TrjBase::adjust_();
       d_=x_.dim();
       R n1=x_.norm();
       n_1_=n1-1;
    }
public:
   typedef TrjT<StateT> Type;
   virtual TrjBase* clone()const{ return new TrjT(*this);}
   CPM_IO_V
   TrjT():TrjBase(){}
   TrjT(StateT const& x, TrjCon const& tc):TrjBase(tc),x_(x){cpmmessage\
   ("calling ctor of TrjT");}

   void setState_(StateT const& x){  x_=x; adjust_();}
      // setting a new x_ and adjusting the rest. since adjust_
      // is virtual, this definition is OK for all derived classes

   State getRep()const{ return x_.toState(tc_.ignPhs_); }
      // the return type is State and not the template argument StateT
      // Here State is a data type supporting visualization
      // also in cases where StateT is not of fixed dimension

   StateT& x(){return x_;}

   StateT const& x()const{return x_;}

    void norIfEnf_()
       // normalize if enforced, update n_1_ always
```

```
   {
      if (tc_.enfNor_) x_.nor_();
      n_1_=x_.norm()-1.;
   }

   virtual Word text()const{ return Word("TrjT<>");}

   bool fileState_(ostream& str){
      cpmp(d_);
      cpmp(t_);
      cpmp(n_1_);
      cpmp(enr_);
      cpmp(x_);
      return true;
   }

   bool readState_(istream& str)
   {
      Z mL=3;
      Word loc="Trj<>::readState_(istream)";
      CPM_MA
      cpms(d_);
      cpms(t_);
      cpms(n_1_);
      cpms(enr_);
      cpms(x_);
      stepCon_+=1;
         // we dont write the step counter to file
      CPM_MZ
      return true;
   }
};

template <class StateT>
bool TrjT<StateT>::prnOn(ostream& str)const
{
   cpmwt("TrjT<>");
   if (! TrjBase::prnOn(str)) return false;
   cpmp(x_);
   return true;
}

template <class StateT>
bool TrjT<StateT>::scanFrom(istream& str)
{
   if (! TrjBase::scanFrom(str)) return false;
   cpms(x_);
   adjust_();
   return true;
}
```

```
//////////////////////// TrjDMI<> ////////////////////////////////////////
// Trajectory class which adds to TrjT<> the essential state
// change descriptor v_ and thus allows a final implementation of
// all virtual functions in TrjT<>

template <class StateT>
class TrjDMI: public TrjT<StateT>{ // adding velocity to TrjT

   B initialized;
   StateT v_; //velocity
   virtual StateT op(StateT const& x, R t)const{ t; return StateT();}

//protected:
public:
   typedef TrjT<StateT> Base;
   C eTot()const{ return (Base::x_|v_)*QMBase::get_ihDirac();}
      // expectation value of the operator which (in the sense of Stone's
      // theorem) is the generator of the trajectory.

protected:
   virtual void adjust_(){
      Base::adjust_();
      v_=op(Base::x_,Base::t_)*QMBase::get_ihDiracInv();
      TrjBase::enr()=eTot();
   }

public:
   typedef TrjDMI<StateT> Type;
   virtual TrjBase* clone()const{ return new TrjDMI(*this);}
   CPM_IO_V
   TrjDMI():TrjT<StateT>(){}
   TrjDMI(StateT const& x, TrjCon const& tc):TrjT<StateT>(x,tc)
   {cpmmessage("calling ctor of TrjDMI");}
   virtual Word text()const{ return Word("TrjDMI<>");}
   void step_(R h);
};

template <class StateT>
void TrjDMI<StateT>::step_(R h)
{
   static C fac=QMBase::get_ihDiracInv()*QMBase::get_ihDiracInv();
   static C fac2=QMBase::get_ihDirac();
   static Word loc("TrjDMI<>::step_(R h)");
   Z mL=3;
   CPM_MA
   if (!initialized){
      adjust_();
      initialized=B(true);
   }
```

```
   if (TrjBase::autoStep()){ // due to time dependence of
      // the generator, the proposed time step will vary with time
      // and has to be computed for each step (rather cheaply)
      R n1=Base::x_.norm();
      R n2=v_.norm();
      TrjBase::dt()=QMBase::step(n1,n2);
   }
   else TrjBase::dt()=h;

   R tau=0.5*TrjBase::dt();
   StateT xi=Base::x_+v_*tau;
   R ti=Base::t()+tau;
   StateT dv=op(op(xi,ti),ti)*(fac*TrjBase::dt());
   Base::x_+=(v_*TrjBase::dt()+dv*tau);
   v_+=dv;
   TrjBase::t()+=TrjBase::dt();
   TrjBase::stepCon()+=1;
   Base::norIfEnf_();
   TrjBase::enr()=eTot(); // very cheap function here
   CPM_MZ
}

template <class StateT>
bool TrjDMI<StateT>::prnOn(ostream& str)const
{
   cpmwt("TrjDMI<>");
   if (! TrjT<StateT>::prnOn(str)) return false;
   cpmp(v_);
   return true;
}

template <class StateT>
bool TrjDMI<StateT>::scanFrom(istream& str)
{
   if (! TrjT<StateT>::scanFrom(str)) return false;
   cpms(v_);
   return true;
}

//////////////////////// Trj2DMI<> ////////////////////////////////////////
// Trajectory class which adds to TrjT<> the essential state
// change descriptor v_ , a Hamiltonian, and a decoherence operator.
// It allows a final implementation of all virtual functions in TrjT<>.

template <class StateT, class ObsT, class DecT >
class Trj2DMI: public TrjT<StateT>{//adds velocity and hamiltonian to TrjT

   B initialized;
   StateT v_; // velocity
```

```
   ObsT ham_; // hamiltonian
   DecT dec_; // decoherence operator
   StateT op(StateT const& x, R t)const{ t; return ham_(x);}
   //void dePhase_(){x_=dec_.dePhase(x_,tc_.dph1_,tc_.dph2_,tc_.nBins_);
   // v_=op(x_,t());}
   //void dePhase_(){
   //   X2<R,StateT> xrs=dec_.reduction(x_,tc_.dph1_);
    //  x_=xrs.c2();v_=op(x_,t());
   //}
   void dePhase_(){
      bool act=Base::x_.col_(IvZ(40,41),Base::tc_.dph1_);
      if (act){
         Base::x_.smt_(4);
         v_=op(Base::x_,Base::t());
      }
   }
//protected:
public:
   typedef TrjT<StateT> Base;
   C eTot()const{ return (Base::x_|v_)*QMBase::get_ihDirac();}
      // expectation value of the operator which (in the sense of Stone's
      // theorem) is the generator of the trajectory.

protected:
   virtual void adjust_(){
      Base::adjust_();
      v_=op(Base::x_,Base::t_)*QMBase::get_ihDiracInv();
      TrjBase::enr()=eTot();
   }

public:
   typedef Trj2DMI<StateT,ObsT,DecT> Type;
   virtual TrjBase* clone()const{ return new Trj2DMI(*this);}
   CPM_IO_V
   Trj2DMI():TrjT<StateT>(){}
   Trj2DMI(StateT const& x, ObsT const& ham, DecT const& dec,
      TrjCon const& tc):TrjT<StateT>(x,tc),ham_(ham),dec_(dec)
   {cpmmessage("calling ctor of Trj2DMI");}
   Word text()const{ return Word("Trj2DMI<>");}
   void step_(R h);
};

template <class StateT, class ObsT, class DecT>
void Trj2DMI<StateT,ObsT,DecT>::step_(R h)
{
   static C fac=QMBase::get_ihDiracInv()*QMBase::get_ihDiracInv();
   static Word loc("Trj2DMI<>::step_(R h)");
   Z mL=3;
   CPM_MA
   if (!initialized){
```

```
      adjust_();
      initialized=B(true);
   }

   if (TrjBase::autoStep()){ // due to time dependence of
      // the generator, the proposed time step will vary with time
      // and has to be computed for each step (rather cheaply)
      R n1=Base::x_.norm();
      R n2=v_.norm();
      TrjBase::dt()=QMBase::step(n1,n2);
   }
   else TrjBase::dt()=h;

   R t=TrjBase::t();
   R dt=TrjBase::dt();
   R tau=0.5*dt;

   t+=tau;
   Base::x_+=v_*tau;
   v_+=op(op(Base::x_,t),t)*(fac*dt);
   Base::x_+=v_*tau;
   t+=tau;

   TrjBase::t()+=dt;
   TrjBase::stepCon()+=1;
   dePhase_();
   Base::norIfEnf_();
   TrjBase::enr()=eTot(); // very cheap function here
   CPM_MZ
}

template <class StateT, class ObsT, class DecT>
bool Trj2DMI<StateT,ObsT,DecT>::prnOn(ostream& str)const
{
   cpmwt("Trj2DMI<>");
   if (! TrjT<StateT>::prnOn(str)) return false;
   cpmp(v_);
   return true;
}

template <class StateT, class ObsT, class DecT>
bool Trj2DMI<StateT,ObsT,DecT>::scanFrom(istream& str)
{
   if (! TrjT<StateT>::scanFrom(str)) return false;
   cpms(v_);
   return true;
}

//////////////////////// TrjB ////////////////////////////////////////
// Basis for trajectory classes i.e. classes dealing with the
```

```
//'initial value problem'.
// In order to be able to hold various derivatives of this class
// as components of polymorphic arrays of type Vp<TrjB>
// (see class Trj2) it is important to define the virtual function clone.

typedef TrjT<State> TrjB;

////////////////////// class TrjSR //////////////////////////
// a trajectory holds a SpectralObject from a Observable and an
// initial state. From the spectral data it calculates future states
// based on the spectral theorem and does not need iterated
// matrix multiplication.

class TrjSR: public TrjB{//trajectory from a SpectralObject
   SpectralObject so_;
   Vl<C> cof_;
protected:
   virtual void adjust_();
public:
   typedef TrjSR Type;
   virtual TrjBase* clone()const{ return new TrjSR(*this);}
   CPM_IO_V
   TrjSR(){}
   TrjSR(const Observable& h, const State& x, TrjCon const& tc );
      // creating a trajectory from hamiltonian and initial state
   TrjSR(const SpectralObject& so, const State& x, TrjCon const& tc );
      // creating a trajectory from the spectral representation of its
      // hamiltonian and an initial state

   C eTot()const{ return (x_|so_*x_);}
      // see declaration in TrjB

   virtual Word text()const{ return Word("TrjSR");}

   virtual void step_(R h);
   // if autoStep_==false, h is the time step that will be applied
   // to the state of *this. For autoStep_==true the time step to be
   // done is dt_ independent of the value of h.
};

////////////////////// class TrjDMI_Obs ////////////////////////////////
class TrjDMI_Obs: public TrjDMI<State>{ // trajectory from observable

   Observable obs_;
   virtual State op(State const& x, R t)const{ return obs_*x;}
public:
   typedef TrjDMI_Obs Type;
   virtual TrjBase* clone()const{ return new TrjDMI_Obs(*this);}
   CPM_IO_V
   TrjDMI_Obs(){}
```

```
   TrjDMI_Obs(Observable const& obs, State const& x, TrjCon const& tc):
   TrjDMI<State>(x,tc),obs_(obs){}
   virtual Word text()const{ return Word("TrjDMI_Obs");}
};

////////////////////// class Trj2 ////////////////////////////////////////
// This class holds two trajectories that have the same
// initial state and evolve in time according to the two
// methods considered above. The runAsCurves method aimes
// at making a comparison of the two methods easy. Notice that
// the SR-method is exact mathematically (which does not
// exclude numerical inaccuracies) whereas the DMI-method
// is a approximation (however, a simple,stable, and accurate one)
// After the reimplementation of 2004-09-26 we can compare all
// trajectory types that are derived from the newly introduced
// trajectory base class TrjB. Introduction of this class
// eliminated all major code doublications

class Trj2{ // array of trajectories from various methods for comparison

   Vp<TrjBase> rep_;

public:
   Trj2(){}
   Trj2(TrjBase const& trj1, TrjBase const& trj2 ):rep_(2)
   { rep_[1]=trj1; rep_[2]=trj2;}

   Trj2(Vp<TrjBase> const& rep):rep_(rep){}
   void step_(R h);
   Z dim()const{ return rep_(2).dim();}

   R n_1()const{ return rep_(2).n_1();}

   State getRep1()const{ return rep_(1).getRep();}
   State getRep2()const{ return rep_(2).getRep();}

   Word text1()const{ return rep_(1).text();}
   Word text2()const{ return rep_(2).text();}

   R t()const{ return rep_(2).t();}
   R dt()const{rep_(2).dt();}
   bool vwnAlarm_(){ return rep_(2).vwnAlarm_();}
   bool filAlarm_(){ return rep_(2).filAlarm_();}

   void fileState_(ostream& str1, ostream& str2)
   { rep_(1).fileState_(str1); rep_(2).fileState_(str2);}

   void run(Graph& g, ostream& str1, ostream& str2);
      // running n steps of length n and representing this as a series of
      // curves. Time range as in previous function. Each frame is at
```

```
      // least shown for a time of waitTime seconds.
      // The y-range is set so that if the initial state would be
      // multiplied by secFac (security factor) it would fill the frame.
      // So setFac>1 is needed to see the whole state *this.
      // If autoAdj (auto-adjust) is true, the y-range gets reset
      // if needed to let the state not leave the range of g. If secFac
      // is only a bit larger than 1, this adjustment will take place
      // often, which could be annoying; so something like secFac=1.25
      // should be chosen.
   void play(Graph& g, istream& str1, istream& str2);
};

} // namespace
#endif
```

207 cpmtrj.cpp

```
//? cpmtrj.cpp
//? Status of work 2008-10-25.
//?
#include <cpmtrj.h>
#include <cpmframes.h>
#include <cpmrmatrix.h>

   using namespace CpmArrays;
   using CpmRootX::Record;
   using CpmGraphics::Frames;
   using CpmGraphics::Graph;
   using CpmQuantumMechanics::State;
   using CpmQuantumMechanics::Observable;
   using CpmQuantumMechanics::SpectralObject;
   using CpmQuantumMechanics::TrjCon;
   using CpmQuantumMechanics::TrjBase;
   using CpmQuantumMechanics::TrjSR;
   using CpmQuantumMechanics::TrjDMI;
   using CpmQuantumMechanics::TrjDMI_Obs;
   using CpmQuantumMechanics::Trj2;
   using std::vector;

////////////////// struct TrjCon ////////////////////////////////////////
TrjCon::TrjCon(Record const& rc)
// here in alphabetic order for fast retreival
{
   Word sec("TrjCon");
   Z pedRed=0; // pedantry reduced
   rc.get(sec,"autoAdj",autoAdj_);
   rc.get(sec,"autoStep",autoStep_);
   rc.get(sec,"dt",dt_);
   rc.get(sec,"enfNor",enfNor_);
   rc.get(sec,"ignorePhases",ignPhs_);
   rc.get(sec,"nSteps",nSteps_);
   rc.get(sec,"displayMode",disMode_,pedRed);
   rc.get(sec,"secFac",secFac_);
   rc.get(sec,"tWait",tWait_);
   rc.get(sec,"viewingPeriod",vwnPer_);
   rc.get(sec,"filingPeriod",filPer_,pedRed);
   rc.get(sec,"imagingPeriod",imgPer_,pedRed);
   rc.get(sec,"dph1",dph1_,pedRed);
   rc.get(sec,"dph2",dph2_,pedRed);
   rc.get(sec,"dphMethod",dphMethod_,pedRed);
   rc.get(sec,"corGrnMet",corGrnMet_,pedRed);
   rc.get(sec,"nPhi",nPhi_,pedRed);
   rc.get(sec,"nBins",nBins_,pedRed);
   rc.get(sec,"gam",gam_,pedRed);
```

```
   rc.get(sec,"intMet",intMet_,pedRed);
}

bool TrjCon::prnOn(ostream& str)const
{
   cpmwt("TrjCon");
   cpmp(autoAdj_);
   cpmp(autoStep_);
   cpmp(dt_);
   cpmp(enfNor_);
   cpmp(ignPhs_);
   cpmp(nSteps_);
   cpmp(disMode_);
   cpmp(secFac_);
   cpmp(tWait_);
   cpmp(vwnPer_);
   cpmp(filPer_);
   cpmp(imgPer_);
   cpmp(dph1_);
   cpmp(dph2_);
   cpmp(dphMethod_);
   cpmp(corGrnMet_);
   cpmp(nPhi_);
   cpmp(nBins_);
   cpmp(gam_);
   cpmp(intMet_);
   return true;
}

bool TrjCon::scanFrom(istream& str)
{
   cpms(autoAdj_);
   cpms(autoStep_);
   cpms(dt_);
   cpms(enfNor_);
   cpms(ignPhs_);
   cpms(nSteps_);
   cpms(disMode_);
   cpms(secFac_);
   cpms(tWait_);
   cpms(vwnPer_);
   cpms(filPer_);
   cpms(imgPer_);
   cpms(dph1_);
   cpms(dph2_);
   cpms(dphMethod_);
   cpms(corGrnMet_);
   cpms(nPhi_);
   cpms(nBins_);
   cpms(gam_);
```

```
   cpms(intMet_);
   return true;
}

//////////////////// class TrjBase ////////////////////////////////////
// code fragments: "shared code"

#define CPM_SC1 R waitTime=tc_.tWait_;\
   Z nCol=g.getCol();\
   Z width=4;\
   Z nImgLin=nCol/width;\
   Z nRow=g.getLin();\
   Z iFr=2;\
   Z height=(nRow-iFr)/d_;\
   Z ii=iFr,jj=-width;\
   bool fullFrame=false;\
   Z countImg=0;\
   Z showCount=0;

#define CPM_SC2 if(vwnAlarm_()){\
         jj+=width;\
         showCount++;\
         if (showCount==nImgLin) fullFrame=true;\
         Z4 z4(height,width,ii,jj);\
         showAsLine(g,z4);\
         if (fullFrame){\
            fullFrame=false;\
            showCount=0;\
            jj=-width;\
            if (imgAlarm_()){\
               countImg++;\
               Word imf="TrjBase_runAsLine";\
               g.write(imf&cpm(countImg,3));\
               }\
         }\
         cpmwait(waitTime);\
      }

void TrjBase::runAsLine(Graph& g, ostream& str)
{
   Z mL=1;
   Word loc("TrjBase::runAsLine(Graph)");
   CPM_MA
   Z n=tc_.nSteps_;
   if (n<1) return;
   R nInv=1./n;
   R dt=tc_.dt_;
   CPM_SC1
   Z i=0;
   while(keepGoing()){ // shows n states
```

```
      cpmprogress("Loop",i*nInv);
      i++;
      CPM_SC2
      if (filAlarm_()) fileState_(str);
      step_(dt);
   }
   cpmprogress("Loop",1);
   CPM_MZ
}

void TrjBase::playAsLine(Graph& g, istream& str)
{
   Z mL=1;
   Word loc("TrjBase::playAsLine(Graph)");
   CPM_MA
   bool goOn=readState_(str);
   // notice that CPM_SC1 needs to know d_; we should not
   // assume the calling state to give this information
   // in a way that is consistent with the data from str
   CPM_SC1
   Z frame=1;
   while (goOn && keepGoing()){
      cpmvalue("frame#",++frame,2);
      goOn=readState_(str);
      CPM_SC2
   }
   CPM_MZ
}
#undef CPM_SC1
#undef CPM_SC2

void TrjBase::runAsArea(Graph& g, ostream& str )
{
   using namespace CpmImaging;
   using namespace CpmArrays;
   using namespace CpmSystem;

   Z mL=1;
   Word loc("TrjBase::runAsArea(Graph)");
   CPM_MA

   R waitTime=tc_.tWait_;
   R secFac=tc_.secFac_;
   bool autoAdj=tc_.autoAdj_;
   bool ignPhs=tc_.ignPhs_;
   R dt=tc_.dt_;
   R tTot=tc_.tTot_;
   Z nSteps=tc_.nSteps_;
   Z nStepsMod=nSteps+2;
```

```
   R_Vector times(nStepsMod);
   R_Vector norms(nStepsMod);
   V<C> energies(nStepsMod);
   R aspRatMem=g.getAspRat();
   g.clr();
   g.setAspRatHoldHeight(1.);

   Z i,d=dim();
   Z countAct=1;
   Z countImg=0;

   times[countAct]=t();
   norms[countAct]=n_1();
   energies[countAct]=enr();

   Z ncl=2;
   V<Color> cl(ncl);
   cl[1]=RED;  // color for real part
   cl[2]=GREEN;   // color for imaginary part
// 'dynamics' loop (showing a one-parameter group)
   R nStepsInv=1./nSteps;
   Z count=0;
   showAsArea(g);
   cpmmessage("showing the initial state");
   R tIniWait=2.5;
   Z mesPane=2; // message pane
   cpmwait(tIniWait,mesPane);
   while (keepGoing()){
      cpmprogress("Loop",count*nStepsInv);
      step_(dt);
      count++;
      if (filAlarm_()) fileState_(str);
// graphics part
      if (vwnAlarm_()){
         countAct++;
         times[countAct]=t();
         norms[countAct]=n_1();
         energies[countAct]=enr();
         showAsArea(g);
         cpmwait(waitTime);
      }
      if(imgAlarm_()){
         countImg++;
         Word imf="TrjBase_runAsArea";
         g.write(imf&cpm(countImg,3));
      }
   }
   cpmmessage("keep going loop finished");
   times=times.resize(countAct);
   norms=norms.resize(countAct);
```

```
   energies=energies.resize(countAct);
   g.clr();
   g.setAspRatHoldHeight(aspRatMem);
   Frames frs(g,Z(1),Z(2));
   Graph g1(frs[1][1]);
   Graph g2(frs[1][2]);
   g1.setWithOrigin(0);
   g2.setWithOrigin(0);
   g1.addText("(norm of state) - 1");
   g1.show(times,norms);
   C e0=energies[1];
   R e0abs=e0.absVal();
   bool b0=(e0abs==0.);
   R fac0= b0 ? 1. : 1./e0abs;
   R_Matrix en(ncl,countAct);
   cpmmessage("next loop comes soon");
   for (i=1;i<=countAct;++i){
      C val=energies[i];
      val-=e0;
      en[1][i]=val[1]*fac0;
      en[2][i]=val[2]*fac0;
   }
   if (b0)
       g2.addText("deviation of total energy from initial value");
   else
       g2.addText(
         "relative deviation of total energy from initial value");
   g2.show(times,en,cl);
   cpmmessage("addText done");

   CPM_MZ
}

void TrjBase::playAsArea(Graph& g, istream& str )
{
   Z mL=1;
   Word loc("TrjBase::playAsArea(Graph)");
   CPM_MA
   R waitTime=tc_.tWait_;
   R secFac=tc_.secFac_;
   bool ignPhs=tc_.ignPhs_;
   Z ncl=ignPhs ? 1 : 3;
   V<Color> cl(ncl);
   cl[1]=BLACK; // color for absolute value
   if (ncl==3){
      cl[2]=RED;  // color for real part
      cl[3]=GREEN;   // color for imaginary part
   }

   vector<R> tim;
```

```
   vector<R> nor;
   vector<C> en;
   bool goOn=true;
   Z countImg=0;
   Z frame=0;
   while (goOn && keepGoing()){
      cpmvalue("frame#",++frame,2);
      goOn=readState_(str);
      if(vwnAlarm_()){
         tim.push_back(t());
         nor.push_back(n_1());
         en.push_back(enr());
         showAsArea(g);
         cpmwait(waitTime);
      }
      if(imgAlarm_()){
         countImg++;
         Word imf="TrjBase_runAsArea";
         g.write(imf&cpm(countImg,3));
      }
   }

   Z countAct=(Z)tim.size();
   R_Vector times(tim);
   R_Vector norms(nor);
   g.clr();
   Frames frs(g,Z(1),Z(2));
   Graph g1(frs[1][1]);
   Graph g2(frs[1][2]);
   g1.setWithOrigin(0);
   g2.setWithOrigin(0);
   g1.show(times,norms);
   g1.addText("norm of state - 1");
   C e0=en[0];
   R fac0=1./e0.absVal();
   R_Matrix em(ncl,countAct);
   for (Z i=1;i<=countAct;++i){
      C val=en[i-1];
      val-=e0;
      em[1][i]=val.absVal()*fac0;
      if (ncl==3){
         em[2][i]=val[1]*fac0;
         em[3][i]=val[2]*fac0;
      }
   }
   g2.show(times,em,cl);
   g2.addText("relative deviation of total energy from initial value");
   CPM_MZ
}
```

```
void TrjBase::runAsCur(Graph& g, ostream& str )
{
   using namespace CpmImaging;
   using namespace CpmArrays;
   using namespace CpmSystem;

   Z mL=1;
   Word loc("TrjBase::runAsCur(Graph)");
   CPM_MA

   R waitTime=tc_.tWait_;
   R secFac=tc_.secFac_;
   bool autoAdj=tc_.autoAdj_;
   bool ignPhs=tc_.ignPhs_;
   R dt=tc_.dt_;
   R tTot=tc_.tTot_;
   Z nSteps=tc_.nSteps_;
   Z nStepsMod=nSteps+2;

   R_Vector times(nStepsMod);
   R_Vector norms(nStepsMod);
   V<C> energies(nStepsMod);

// showing the initial condition
   Z i,d=dim();
   R secFacInv=1./secFac;
   R yMax=sup();
   R yScale=yMax*secFac;
   Iv band;
   if (autoAdj) band=Iv(yMax*secFacInv,yMax*secFac);
   g.setAutoScale(0);
   g.setX(Iv(0,d));
   if (autoAdj){
      g.setY(Iv(-band.sup(),band.sup()));
   }
   else{
      g.setY(Iv(-yScale,yScale));
   }

   Z ncl=ignPhs ? 1 : 3;
   V<Color> cl(ncl);
   cl[1]=BLACK; // color for absolute value
   if (ncl==3){
      cl[2]=RED;  // color for real part
      cl[3]=GREEN;   // color for imaginary part
   }

   Z countAct=1;
   Z countImg=0;
```

```
   times[countAct]=t();
   norms[countAct]=n_1();
   energies[countAct]=enr();

   R_Matrix res(ncl+1,d);
   for (i=1;i<=d;i++){
      res[1][i]=i-1;
      C val=getRep()[i-1];
      res[2][i]=val.absVal();
      if (ncl==3){
         res[3][i]=val[1];
         res[4][i]=val[2];
      }
   }
   g.clr();
   g.addText(
      "Initial state for evolution. Real part red, Imaginary part green");
   g.show(res,cl);
   cpmwait(2.+3.*waitTime);
// 'dynamics' loop (showing a one-parameter group)
   R nStepsInv=1./nSteps;
   Z count=0;
   while (keepGoing()){  // one state already shown,
      cpmprogress("Loop",count*nStepsInv);
      step_(dt);
      count++;
      if (count>nSteps){ // no longer happens, but also does not hurt
         ostringstream ost;
         ost<<"t_="<<t_<<" tTot="<<tTot<<" count="<<count<<endl;
         cpmwarning(ost);
         cpmerror("bad time range");
      }
      if (filAlarm_()) fileState_(str);
// graphics part
      if (vwnAlarm_()){
         countAct++;
         times[countAct]=t();
         norms[countAct]=n_1();
         energies[countAct]=enr();

         for (i=1;i<=d;i++){
            C val=getRep()[i-1];
            res[2][i]=val.absVal();
            if (ncl==3){
               res[3][i]=val[1];
               res[4][i]=val[2];
            }
         }
         if (autoAdj){
            yMax=sup();
```

```
            bool bc=band.contains(yMax);
            if (!bc){
               band=Iv(yMax*secFacInv,yMax*secFac);
               g.setY(Iv(-band.sup(),band.sup()));
             //  g.writeGridText();
            }
         }
         g.clear();
         g.addText(text());
         g.show(res,cl);

         if(imgAlarm_()){
            countImg++;
            Word imf="TrjBase_runAsCurves";
            g.write(imf&cpm(countImg,3));
         }
         cpmwait(waitTime);
      } // vwnAlarm_
   }
   cpmmessage("keep going loop finished");
   times=times.resize(countAct);
   norms=norms.resize(countAct);
   energies=energies.resize(countAct);
   g.clr();
   Frames frs(g,Z(1),Z(2));
   Graph g1(frs[1][1]);
   Graph g2(frs[1][2]);
   g1.setWithOrigin(0);
   g2.setWithOrigin(0);
   g1.addText("(norm of state) - 1");
   g1.show(times,norms);
   C e0=energies[1];
   R e0abs=e0.absVal();
   bool b0=(e0abs==0.);
   R fac0= b0 ? 1. : 1./e0abs;
   R_Matrix en(ncl,countAct);
   cpmmessage("next loop comes soon");
   for (i=1;i<=countAct;++i){
      C val=energies[i];
      val-=e0;
      en[1][i]=val.absVal()*fac0;
      if (ncl==3){
         en[2][i]=val[1]*fac0;
         en[3][i]=val[2]*fac0;
      }
   }

   if (b0)
       g2.addText("deviation of total energy from initial value");
   else
```

```
        g2.addText(
          "relative deviation of total energy from initial value");
   g2.show(times,en,cl);
   cpmmessage("addText done");

   CPM_MZ
}

void TrjBase::playAsCur(Graph& g, istream& str )
{
   using namespace CpmImaging;
   using namespace CpmArrays;
   using namespace CpmSystem;

   Z mL=1;
   Word loc("TrjBase::playAsCur(Graph)");
   CPM_MA

   R waitTime=tc_.tWait_;
   R secFac=tc_.secFac_;
   bool autoAdj=tc_.autoAdj_;
   bool ignPhs=tc_.ignPhs_;

   vector<R> tim;
   vector<R> nor;
   vector<C> en;

   bool goOn=readState_(str);
   cpmassert(goOn,loc);

   Z i,d=dim();
   R secFacInv=1./secFac;
   R yMax=sup();
   R yScale=yMax*secFac;
   Iv band;
   if (autoAdj) band=Iv(yMax*secFacInv,yMax*secFac);
   g.setAutoScale(0);
   g.setX(Iv(0,d));
   if (autoAdj){
      g.setY(Iv(-band.sup(),band.sup()));
   }
   else{
      g.setY(Iv(-yScale,yScale));
   }
   Z ncl=ignPhs ? 1 : 3;
   V<Color> cl(ncl);
   cl[1]=BLACK; // color for absolute value
   if (ncl==3){
      cl[2]=RED;  // color for real part
      cl[3]=GREEN;   // color for imaginary part
```

```
   }

 //  Z countAct=1;
   Z countImg=0;

   tim.push_back(t());
   nor.push_back(n_1());
   en.push_back(enr());

   R_Matrix res(ncl+1,d);
   for (i=1;i<=d;i++){
      res[1][i]=i-1;
      C val=getRep()[i-1];
      res[2][i]=val.absVal();
      if (ncl==3){
         res[3][i]=val[1];
         res[4][i]=val[2];
      }
   }
   g.clr();
   g.addText("Initial state for evolution.\
 Real part red, Imaginary part green)");
   g.show(res,cl);
   cpmwait(2.+3.*waitTime);
   Z frame=0;
   while (goOn && keepGoing()){
      goOn=readState_(str);
      cpmvalue("frame#",++frame,2);
// graphics part
      if (vwnAlarm_()){
         tim.push_back(t());
         nor.push_back(n_1());
         en.push_back(enr());

         for (i=1;i<=d;i++){
            C val=getRep()[i-1];
            res[2][i]=val.absVal();
            if (ncl==3){
               res[3][i]=val[1];
               res[4][i]=val[2];
            }
         }
         if (autoAdj){
            yMax=sup();
            bool bc=band.contains(yMax);
            if (!bc){
               band=Iv(yMax*secFacInv,yMax*secFac);
               g.setY(Iv(-band.sup(),band.sup()));
             //  g.writeGridText();
            }
```

```
         }
         g.clear();
         g.addText(text());
         g.show(res,cl);
         if(imgAlarm_()){
            countImg++;
            Word imf="TrjBase_runAsCurves";
            g.write(imf&cpm(countImg,3));
         }
         cpmwait(waitTime);
      } // vwnAlarm_
   }
   R_Vector times(tim);
   R_Vector norms(nor);
   Z countAct=times.dim();

   g.clr();
   Frames frs(g,Z(1),Z(2));
   Graph g1(frs[1][1]);
   Graph g2(frs[1][2]);
   g1.setWithOrigin(0);
   g2.setWithOrigin(0);
   g1.addText("norm of state - 1");
   g1.show(times,norms);

   C e0=en[0];
   R e0abs=e0.absVal();
   bool b0=(e0abs==0.);
   R fac0= b0 ? 1. : 1./e0abs;
   R_Matrix em(ncl,countAct);
   for (i=1;i<=countAct;++i){
      C val=en[i-1];
      val-=e0;
      em[1][i]=val.absVal()*fac0;
      if (ncl==3){
         em[2][i]=val[1]*fac0;
         em[3][i]=val[2]*fac0;
      }
   }
   if (b0)
      g2.addText("deviation of total energy from initial value");
   else
      g2.addText("relative deviation of total energy from initial value"\
      )\
      ;
   g2.show(times,em,cl);
   CPM_MZ
}

bool TrjBase::prnOn(ostream& str)const
```

```
{
   cpmwt("TrjBase");
   cpmp(tc_);
   cpmp(t_);
   cpmp(dt_);
   cpmp(n_1_);
   cpmp(enr_);
   return true;
}

bool TrjBase::scanFrom(istream& str)
{
   cpms(tc_);
   cpms(t_);
   cpms(dt_);
   cpms(n_1_);
   cpms(enr_);
   adjust_(); // makes cyc_, d_, n1_
   return true;
}

//////////////////////// class TrjSR ////////////////////////////////

void TrjSR::adjust_()
      // sets dt_ in accordance with so, x_, and QMBase::phiStep
{
   TrjB::adjust_();
   static R fac=QMBase::get_hDiracInv();
   Z d=so_.dim();
   cpmassert(d==d_,"TrjSR::adjust_()");
   R n1=1.+n_1_;
   State sox=so_*x_;
   enr_=eTot();
   R n2=sox.norm()*fac;
   dt_=QMBase::step(n1,n2);
   cof_=Vl<C>(d);
   for (Z i=0;i<d;i++) cof_[i]=(so_[i]|x_);
}

TrjSR::TrjSR(Observable const& a, State const& x, TrjCon const& tc):
TrjB(x,tc), so_(a.spectralRepresentation()){}

TrjSR::TrjSR(const SpectralObject& so, const State& x, TrjCon const& tc):
TrjB(x,tc),so_(so){}

void TrjSR::step_(R h)
//experiment: dephasing in addition to dynamics
{
   static R fac=-QMBase::get_hDiracInv();
   if (!autoStep()) dt_=h;
```

```
   if (dt_!=0.){
      SpectralObject so(so_);
      so.expi(dt_*fac);
      x_=so*x_;
   }
   Z im=trjCon().dphMethod_;
   Z corGrnMet=trjCon().corGrnMet_;
   if (im>0){
      R p1=trjCon().dph1_;
      R p2=trjCon().dph2_;
      x_.dePhase(p1,p2,im);
   }
   if (corGrnMet){
      Z nPhi=trjCon().nPhi_;
      Z nBins=trjCon().nBins_;
      x_=x_.corGrn(nPhi,nBins);
   }
   t_+=dt_;
   stepCon_+=1;
   norIfEnf_();
   enr_=eTot();
}

bool TrjSR::prnOn(ostream& str)const
{
   cpmwt("TrjSR");
   if (! TrjB::prnOn(str)) return false;
   cpmp(so_);
   return true;
}

bool TrjSR::scanFrom(istream& str)
{
   if (! TrjB::scanFrom(str)) return false;
   cpms(so_);
   adjust_();
   return true;
}

////////////////// class TrjDMI_Obs ///////////////////////

bool TrjDMI_Obs::prnOn(ostream& str)const
{
   cpmwt("TrjDMI_Obs");
   if (! TrjDMI<State>::prnOn(str)) return false;
   cpmp(obs_);
   return true;
}

bool TrjDMI_Obs::scanFrom(istream& str)
```

```
{
   if (! TrjDMI<State>::scanFrom(str)) return false;
   cpms(obs_);
   adjust_();
   return true;
}

//////////////////////class Trj2 ////////////////////////////////

void Trj2::step_(R h)
{
// For the not very probable situation that the two
// trajectories should ask for rather different auto
// steps, here is a reasonable method to handle the
// case. Essential for reliable comparison is that
// both trajectories do the same time step so that we have
// a common time parameter for both trajectories.
   static Z toggle=1;
   static R dtLast=0;
   toggle=-toggle;
   bool as1=rep_(1).autoStep();
   bool as2=rep_(2).autoStep();
   if (toggle==1){
      rep_(2).step_(h);
      R dt=rep_(2).dt();
      if (dtLast>0 && dt>dtLast) dt=0.5*(dt+dtLast);
      rep_(1).setAutoStep_(false);
      rep_(1).step_(dt);
      rep_(1).setAutoStep_(as1);
      dtLast=dt;
   }
   else{
      rep_(1).step_(h);
      R dt=rep_(1).dt();
      if (dtLast>0 && dt>dtLast) dt=0.5*(dt+dtLast);
      rep_(2).setAutoStep_(false);
      rep_(2).step_(dt);
      rep_(2).setAutoStep_(as2);
      dtLast=dt;
   }
}

/*
void Trj2::step_(R h)
{
   rep_(1).step_(h);
   rep_(2).step_(h);
}
*/
```

```
void Trj2::run(Graph& g, ostream& str1, ostream& str2)
{
   using namespace CpmImaging;
   using namespace CpmArrays;
   using namespace CpmSystem;
   rep_(1).adjust_();
   rep_(2).adjust_();

   TrjCon tc=rep_(2).trjCon();

   R waitTime=tc.tWait_;
   R secFac=tc.secFac_;
   bool autoAdj=tc.autoAdj_;
   Z nSteps=tc.nSteps_;
   R dt=tc.dt_;
   bool ignPhs=tc.ignPhs_;

   g.clr();
   Frames frs(g,Z(1),Z(2));
   Graph g1(frs[1][1]);
   Graph g2(frs[1][2]);
   Frames frs2(g2,Z(2),Z(1));
   Graph g12(frs2[1][1]);
   Graph g22(frs2[2][1]);

 // showing the initial condition
   Z i,d=dim();
   R secFacInv=1./secFac;
   R y1=rep_(1).sup();
   R y2=rep_(2).sup();
   R yMax=CpmRootX::sup<R>(y1,y2);
   R yScale=yMax*secFac;
   Iv band;
   if (autoAdj) band=Iv(yMax*secFacInv,yMax*secFac);

   g1.setAutoScale(0);
   g1.setX(Iv(0,d));
   g12.setAutoScale(0);
   g12.setX(Iv(0,d));
   g22.setAutoScale(0);
   g22.setX(Iv(0,d));
   if (autoAdj){
      g1.setY(Iv(-band.sup(),band.sup()));
      g12.setY(Iv(-band.sup(),band.sup()));
      g22.setY(Iv(-band.sup(),band.sup()));
   }
   else{
      g1.setY(Iv(-yScale,yScale));
      g12.setY(Iv(-yScale,yScale));
      g22.setY(Iv(-yScale,yScale));
```

```
}
Z ncl=ignPhs ? 1 : 3;
V<Color> cl(ncl);
cl[1]=BLACK; // color for absolute value
if (ncl==3){
   cl[2]=RED;  // color for real part
   cl[3]=GREEN;   // color for imaginary part
}

Z ncl1=ncl+1;
Z nSteps1=nSteps+2;
R_Vector times(nSteps1);
R_Vector n_1(nSteps1);
R_Vector diff(nSteps1);

R_Matrix res1(ncl1,d);
R_Matrix res2(ncl1,d);
R_Matrix res3(ncl1,d);
for (i=1;i<=d;i++){
   res1[1][i]=i-1;
   res2[1][i]=i-1;
   res3[1][i]=i-1;
   C val1=rep_(1).getRep()[i-1];
   C val2=rep_(2).getRep()[i-1];
   C val3=val1-val2;
   res1[2][i]=val1.absVal();
   res2[2][i]=val2.absVal();
   res3[2][i]=val1.absVal()-val2.absVal();
   if (ncl1==4){
      res1[3][i]=val1[1];
      res1[4][i]=val1[2];
      res2[3][i]=val2[1];
      res2[4][i]=val2[2];
      res3[3][i]=val3[1];
      res3[4][i]=val3[2];
   }
}
g1.clr();
g1.addText("Initial state for evolution. Real part red, Imaginary\
 part\
 green");
g1.show(res1,cl);
g12.clr();
g12.addText("Initial state for evolution. Real part red, Imaginary\
 part green");
g12.show(res2,cl);
cpmwait(2.+3.*waitTime);

Z countAct=0;
R valn=rep_(2).n_1();
```

```
   R valDiff=(getRep2()-getRep1()).absVal();
    countAct++;
    times[countAct]=t();
    n_1[countAct]=valn;
    diff[countAct]=valDiff;
// 'dynamics' loop (showing a one-parameter group)
   R nStepsInv=1./nSteps;
   for (Z count=0;count<nSteps;count++){  // one state already shown,
        // so we show nSteps more, making up nSteps+1 states and nSteps
        // integration steps
      cpmprogress("Loop",count*nStepsInv);
      step_(dt);
      if (filAlarm_()) fileState_(str1,str2);
// graphics part
      if (vwnAlarm_()){
         valn=rep_(2).n_1();
         valDiff=(getRep2()-getRep1()).absVal();
         countAct++;
         times[countAct]=t();
         n_1[countAct]=valn;
         diff[countAct]=valDiff;
         for (i=1;i<=d;i++){
            res1[1][i]=i-1;
            res2[1][i]=i-1;
            res3[1][i]=i-1;
            C val1=getRep1()[i-1];
            C val2=getRep2()[i-1];
            C val3=val1-val2;
            res1[2][i]=val1.absVal();
            res2[2][i]=val2.absVal();
            res3[2][i]=val1.absVal()-val2.absVal();
            if (ncl1==4){
               res1[3][i]=val1[1];
               res1[4][i]=val1[2];
               res2[3][i]=val2[1];
               res2[4][i]=val2[2];
               res3[3][i]=val3[1];
               res3[4][i]=val3[2];
            }
         }
         if (autoAdj){
            y1=getRep1().sup();
            y2=getRep2().sup();
            yMax=CpmRootX::sup<R>(y1,y2);
            bool bc=band.contains(yMax);
            if (!bc){
               band=Iv(yMax*secFacInv,yMax*secFac);
               g1.setY(Iv(-band.sup(),band.sup()));
//               g1.writeGridText();
               g12.setY(Iv(-band.sup(),band.sup()));
```

```
                g22.setY(Iv(-band.sup(),band.sup()));
            //   g12.writeGridText();
            //   g22.writeGridText();
            }
         }
      }
      g.clear();
      g1.addText(text1());
      g12.addText(text2());
      g22.addText("Difference");
      g1.show(res1,cl);
      g12.show(res2,cl);
      g22.show(res3,cl);
      cpmwait(waitTime);
   }
   times=times.resize(countAct);
   n_1=n_1.resize(countAct);
   diff=diff.resize(countAct);

   g12.clr();
   g22.clr();
   g12.setAutoScale(1);
   g22.setAutoScale(1);
   g22.setWithOrigin(0);
   g12.addText("deviation of norm from 1");
   g12.show(times,n_1);
   g22.addText("distance between trajectories");
   g22.show(times,diff);
}

void Trj2::play(Graph& g, istream& str1, istream& str2)
{
   Z mL=2;
   Word loc="Trj2::play(Graph,istream,istream)";
   CPM_MA
   cpmwarning(loc&": not yet implemented");
   CPM_MZ
}//? end of cpmtrj.cpp, version 2004-11-15
```

208 cpmtypes.h

```
//? cpmtypes.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_TYPES_H_
#define CPM_TYPES_H_
/*

   Description: Declares basic infrastructure functions, macros,
      and ---most importatly--- the class versions B, Z1, and R1
      of bool, Z, and R.
*/
#include <cpmsystem.h> // includes cpmwords.h and
   // thus also cpmnumbers.h
#include <cpmmacros.h>
#include <cpmx.h>

#include <cmath>

namespace CpmRootX{ // CpmRoot extended
      // a statement
      //    using namespace CpmRoot;
      // makes live much easier when working with C+-. It is therefore
      // desirable that this does not inject to many names in the users
      // scope. Thus we separate the 'true root' from an 'extended root'
      // that hosts also names such as 'order', 'inf', and 'sup' which
      // are more likely to interfere with a user's names.

   using namespace CpmStd;

   using CpmRoot::Z;
   using CpmRoot::N;
   using CpmRoot::R;
   using CpmRoot::L;
   using CpmRoot::Word;

   const R hugeNumber=1e64;
   const R tinyNumber=1./hugeNumber;

#define cpmcheck(X)\
 if (!CpmRoot::isVal(X)){\
   ostringstream ost;\
   ost<<" bad value: "<<#X "="<< X;\
   cpmerror(Word(ost.str()));}
   // checks whether a quantity of type Z,N,R,Rh is valid
   // and stops the program

#define cpmdebug(X) cpmcerr<<endl<<"C+- debug: "<<#X "="<< X <<endl
```

```
   // useful for placing temporary debugging statements into one's code.
   // From Bruce Eckel: Thinking in C++, Prentice Hall 1995, p. 325
   // typical usage:
   //
   // R_Matrix y=....;
   // cpmdebug(y);          // ';' needed !
   // The part 'debug:' helps to find the debug info in the potentially
   // very large file cpmcerr.txt

#define cpmnote(X) cpmdata<<endl<<"C+- note: "<<#X "="<< X <<endl
   // Works as cpmdebug, but but writes on cpmdata. More suitable
   // for data which are of temporary interest, so that it would
   // not pay to provide a more elaborate documentation for them.
   // Notice that cpmdata is more easily surveyed than cpmcerr.

#define cpmis(X) std::string(#X)
   // 'is' stands for 'identifier to string'.
   // This can't be written as a C or C++ function. Such a function
   // always looks on the value of a variable, wheras the present
   // construct looks on the name of the variable and returns it as a
   // quantity of type string.
   // Consider:
   //    R x=3.14;
   //    cout<<cpmis(x);
   // This will write 'x' and not '3.14'

#define cpmiw(X) Word(#X)
   // much alike previous macro

typedef void (*fpWordToVoid)(const Word&);
   // function pointer for function Word to void as a type

typedef void (*fpVoidToVoid)(void);
   // function pointer for function void to void as a type

Z sig(Z i);
R sig(R i);
   // added 03-11-05, 0 for i==0, -1 for i<0, +1 for i>0

R krn(Z i, Z j);
   // Kronecker; defined as 1. for i==j, 0. else
   // added 2004-08-26

Z parity(Z k);
/*{
   if (k%2) return -1; else return 1;
}*/

// Auxiliar functions.
```

```
void copyTextFile(ifstream& from, ofstream& to);
   // here, the streams from and to have to exist already

void copyTextFile(const Word& nameOfSource, const Word& nameOfCopy );
   // a file of the name nameOfSource has to exist and the copy with the
   // name indicated by the second argument will exist after the function
   // body has been executed

string toString(ifstream& from);
   // copies the content of a text file into a standard library string
   // and thus opens up rich manipulation capabilities mainly by writing
   // these to stringstreams which finally can easily be filed back. In
   // this manner e.g. concatenation of text files becomes nearly trivial

// addition 97-6-5, notice that names min and max clash with
// macro names of Windows. Thus we choose presumably save naming.
// Version with const T& arguments is not always working. Sometimes one
// gets unability to resolve function overload

//////////////////////////// sup ////////////////////////////////////////////
   template <class T>
   T sup(T t1, T t2)
      // sup=supremum=maximum
   {
      if (t1<t2) return t2; else return t1;
   }

   template <class T>
   T sup(T t1, T t2, T t3)
   {
      T temp;
      if (t1<t2) temp=t2; else temp=t1;
      if (temp<t3) return t3; else return temp;
   }

   template <class T>
   T sup(T t1, T t2, T t3, T t4)
   {
      T temp1=sup<T>(t1,t2);
      T temp2=sup<T>(t3,t4);
      return sup<T>(temp1,temp2);
   }

//////////////////////////// inf ////////////////////////////////////////////

   template <class T>
   T inf(T t1, T t2)
      // inf=infimum=minimum
   {
      if (t1<t2) return t1; else return t2;
```

```
}

template <class T>
T inf(T t1, T t2, T t3)
   // inf=infimum=minimum
{
   T temp;
   if (t1<t2) temp=t1; else temp=t2;
   if (temp<t3) return temp; else return t3;
}

template <class T>
T inf(T t1, T t2, T t3, T t4)
{
   T temp1=inf<T>(t1,t2);
   T temp2=inf<T>(t3,t4);
   return inf<T>(temp1,temp2);
}

template <class T>
void swap(T& t1, T& t2)
// After function call t1 has the value that t2 had before
// and t2 has the value that t1 had before.
// If T is a class (not built-in type) we assume that T defines
// operator = and a copy constructor.
{
   T temp=t2;
   t2=t1;
   t1=temp;
}

template <class T>
void order(T& t1, T& t2)
   // after function call we have t1<=t2
   // If T is a class (not built-in type) we assume that T defines
   // a '<=' operation such that always at least one of t1<=t2 ,
   // t2<=t1 is valid. Also operator = and a copy constructor have to
   // be defined.
{

   if (t1<=t2){
      return;
   }
   else{
      T temp=t2;
      t2=t1;
      t1=temp;
   }
}
```

```
// Class equivalents of R , Z, bool.
// These can conveniently be used for class data members
// since they get initialized properly.
////////////////////////////// class R1 ////////////////////////////

   using CpmArrays::X2;

class R1{  // real numbers as a class
   // since there is no need for 1-tupels, we
   // define this a class endowed with infratructure similar than R2,
   // R3, ... but with arithmetics gained from the non-class type R via
   // automatic conversion (in one direction only to avoid ambiguities)
   // Most functions use R-arguments and not R1-ones since R1's have to
   // be converted to R anyway in most situations.
public:
   static const R1 min;
   static const R1 max;
   static const R1 eps0;
   static const R1 eps1;
   static const R1 logmax;
   static const R1 log10max;
   static const R1 pi;
   static const R1 piInv;

   R x1;
   typedef R1 Type;
// constructors
   R1():x1(0){}
   explicit R1(R a):x1(a){} // no automatic conversion wanted
   operator R()const{ return x1;}
      // auto-conversion to R is wanted here
   R1& operator=(R a){ x1=a; return *this;}
   R1& operator+=(R a){ x1+=a; return *this;}
   R1& operator-=(R a){ x1-=a; return *this;}
   R1& operator*=(R a){ x1*=a; return *this;}
   R1& operator/=(R a){ x1*=R1(a).inv(); return *this;}

   R absVal()const{ if (x1<0) return -x1; return x1;}
   R disVal(const R1&)const;
   X2<R,R1> polDec()const;
      // polar decomposition: First component (res.first if res is the
      // return value) is the absolute value r and the second component
      // is sign(x1)
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 1;}
   R1 inf(R1 a)const{ return x1<=a.x1 ? R1(x1) : R1(a.x1);}
   R1 sup(R1 a)const{ return x1>=a.x1 ? R1(x1) : R1(a.x1);}
   R& operator[](Z i){ i; return x1;}  // defining components in analogy
      // to what is done for R2, R3, R4
```

```
   const R& operator[](Z i)const{ i; return x1;}
   void set(R t){ x1=t;}
   void makePos_(){ if (x1<0) x1=-x1;}
      //: make positive
   Word nameOf()const{ return "R1";}
   Word toWord()const{ return Word::write(x1);}
   bool isVal()const{ return CpmRoot::isVal(x1);}
// functions referring to R and R1 as a discrete sets ('lattice')
   R1& next_();
      // next (in lattice)
   R1& prv_();
      // previous (in lattice)
   Z dz(R a)const;
      // distance on the lattice of represented numbers
      // counts the points exactly and thus is very slow for
      // typical pairs of numbers. If two numbers arrose from
      // getting the same 'mathematical number' by different
      // numerical expressions, the values should be close enough
      // so that dz should be small (apploximately the number of
      // operations in the algorithm ???). Function dr is a much
      // faster approximate representation of this discrete distance.
   R eps(Z dir=1)const;
      // distance to the next lattice point in positive direction
      // for dir>=0 and in negative direction for dir<0.
      // In this latter case also the result is returned as negative
      // Thus x+x.eps(1) is the next lattice neighbor of x in
      // positive direction and x+x.eps(-1) is that in negative
      // direction
   R dr(R a);
      // approximate number of lattice intervals between a and *this
      // (always >=0)
   Z run(R xFinal, ostream& str, Z writePeriod=1000);
      // runs from *this to xFinal in steps over all represented
      // numbers and prints out ( to stream o) each n'th of
      // them with n given by writePeriod
      // Returns the number of printed numbers.
// save functions
   R1 inv()const;
      // inverse, returns R1::max.x1 for x1==0
   R exp()const;
      // Returns R1::max.x1 for all x1 that would normally give infinity.
      //
      // Example of usage: instead of exp(x) write R1(x).exp() if it is
      // important to get a valid, non-exceptionl, reult
   R log()const;
   R log10()const;
      // these four functions are defined on whole R1.
      // logs are log of absolute value, a combination which
      // is common in mathematics.
```

```
   R sign()const{ return (x1> 0. ? 1. : (x1==0. ? 0. : -1.));}
         //: sign (English),
         // akin to Latin signum

   R sign(R b)const{ return x1>0 ? x1*R1(b).sign() : -x1*R1(b).sign();}
        // 'takes the modulus from *this and the sign from b'
};

//////////////////////////////// class Z1 ////////////////////////////////

class Z1{ // integer numbers as a class
   // tuples of 1 elements, for formal
   // symmetry with others
public:
   Z x1;
   static const Z1 min;
   static const Z1 max;
   typedef Z1 Type;
// constructors
   Z1():x1(0){}

   explicit Z1(long int a):x1(a){} // no automatic conversion wanted
   explicit Z1(int a):x1(a){} // no automatic conversion wanted
   explicit Z1(R r):x1((Z)(r+0.5)){} // no automatic conversion wanted
   operator Z()const{ return x1;}

   Z1& operator=(Z a){ x1=a; return *this;}
   Z1& operator+=(Z a){ x1+=a; return *this;}
   Z1& operator-=(Z a){ x1-=a; return *this;}
   Z1& operator*=(Z a){ x1*=a; return *this;}

   Z1 pow(Z p)const;
      // Z1(x1^p), cpmassert p>=0

   Z absVal()const{ if (x1<0) return -x1; return x1;}
   R disVal(const Z1&)const;
   X2<Z,Z1> polDec()const;
      // polar decomposition: First component (res.first if res is the
      // return value) is the absolute value r and the second component
      // is sign(x1)
   CPM_IO
   CPM_ORDER
   Z dim()const{ return 1;}
   Z1 inf(Z1 a)const{ return x1<=a.x1 ? Z1(x1) : Z1(a.x1);}
   Z1 sup(Z1 a)const{ return x1>=a.x1 ? Z1(x1) : Z1(a.x1);}
   Z& operator[](Z i){ i; return x1;}  // defining components in analogy
   // to
      // what is done for Z2
   const Z& operator[](Z i)const{i; return x1;}
   void set(Z t){ x1=t;}
```

```
   Word nameOf()const{ return "Z1";}
   Word toWord()const{ return Word::write(x1);}
   bool isVal()const{ return CpmRoot::isVal(x1);}
   Z parity()const{ return CpmRootX::parity(x1);}

   bool isPowOfTwo()const;
      // Let N:={1,2,3,4,...} (no zero!) and
      // P2:={ 2,4,8,16,...}
      // Then we return true iff *this belongs to P2

   Z1 nextPowOfTwo()const;
      // We return the smallest y \in P2 such that *this <= y

   Z1 nextLog2()const;
      // We return the conventional dual logarithm of
      // nextPowOfTwo() ( which then is always \in N)
};

///////////////////////// class B ////////////////////////////
//
// class version for type bool.
// This is not denoted B1 despite the analogy to R1 and Z1.
// Reason: what is introduced as R1 here would be denoted R
// if it could replace the usage of the built-in type double.
// Earlier experiments showed that this would result in a speed loss
// by a factor of about ten and thus is not acceptable. So we have
// to use the type 'double' very often and therefore need an alias which
// can't be anything different from R. So the class version of 'double'
// has to have a name different from R. The name R1 is natural when
// seen in relation to the classes R2, R3, R4 defined in cpmlinalg.h
//
// For the Boolean type the native name 'bool' asks not for an alias
// and so the natural name B for the class version is still free.
//
// Although - as mentioned above - it is not practicable to avoid the
// usage of the built-in types completely (against Eccle's advice in
// his nice book 'Thinking in C++'), my experience strongly favors
// to avoid them as types of data members of user-defined classes.
// This makes writing constructors very convenient, since only the
// non-trivial initializations have to be formulated. Adding a data
// member of type R1, Z1, or B then needs no change in constructors
// to assure proper initialzation. Also I/O can be uniformly coded
// in terms of prnOn and scanFrom.

class B{ // boolean values as a class
public:
   bool x1;
public:
   typedef B Type;
// constructors
```

```
   B():x1(false){}
      // default is false !!!
      // corresponds to default 0 in Z1,R1,...

   explicit B(bool a):x1(a){} // no automatic conversion wanted
   explicit B(Z a):x1(a!=0){} // no automatic conversion wanted
   explicit B(Z1 a):x1(a!=0){} // no automatic conversion wanted
   operator bool()const{ return x1;}

   B& operator=(bool a){ x1=a; return *this;}
   B& operator=(Z a){ x1=(a!=0); return *this;}

   CPM_IO
   CPM_ORDER
   Word nameOf()const{ return "B";}
   Word toWord()const{ return (x1 ? "true" : "false");}
   B operator!()const{ return B(!x1);}
   B operator&(const B& b)const{ return B(x1&&b.x1);}
   B operator|(const B& b)const{ return B(x1||b.x1);}
// escaping to Latin since 'not', 'and', and 'or' now
// are restricted in usage very much like C++ key words
   B non()const{ return B(!x1);}
   B et(const B& b)const{ return B(x1&&b.x1);}
   B vel(const B& b)const{ return B(x1||b.x1);}
   B ranVal(Z j=0){ return B(CpmRoot::ranVal(x1,j));}
};

} //namespace

namespace CpmRoot{
// All similar declarations are in cpmword.h There, the present
// type is not yet defined. So we have to defer the declaration till
// here. That this works, is a triumph of C++'s templates.

   template<>
   class Name<CpmRootX::fpVoidToVoid>{ // also this type needs a name
   public:
      Name(){}
      Word operator()(CpmRootX::fpVoidToVoid const& t)const
      { t; return Word("fpVoidToVoid");}
   };

   template<>
   class Name<CpmRootX::fpWordToVoid>{ // also this type needs a name
   public:
      Name(){}
      Word operator()(CpmRootX::fpWordToVoid const& t)const
      { t; return Word("fpWordToVoid");}
   };
```

```
}// namespace

// new names
#define cpminf       CpmRootX::inf
#define cpmsup       CpmRootX::sup

#define cpmord      CpmRootX::order
#define cpmswp      CpmRootX::swap
#define cpmtin      CpmRootX::tinyNumber
#define cpmhug      CpmRootX::hugeNumber

#endif
```

209 cpmtypes.cpp

```
//? cpmtypes.cpp
//? Status of work 2008-10-25.
//?
#include <stdlib.h>
#include <sstream>

#include <cpmtypes.h>

using namespace CpmSystem;
using namespace CpmRoot;
using CpmArrays::X2;

//////////////////////////////////////////////////////////////////

void CpmRootX::copyTextFile(ifstream& from, ofstream& to)
{
   char ch;
   while (from.get(ch)) to.put(ch);
   if (!from.eof() || to.bad())
   cpmerror("copyTextFile(const ifstream&, ofstream& to) failed");
     // Stroustrup Section 10.5 p. 351 top of page
}

void CpmRootX::copyTextFile(const Word& nameOfSource,
   const Word& nameOfCopy)
{
   Z mL=1;
   Word loc("copying "&nameOfSource&" to "&nameOfCopy);
   cpmassert(nameOfSource!="cpmcerr.txt",loc);
      // copying "cpmcerr.txt" caused problems, which are hard to fully
      // understand
   cpmmessage(mL,loc&" started");
   IFileStream from(nameOfSource);
   OFileStream to(nameOfCopy);
   CpmRootX::copyTextFile(from(),to());
   cpmmessage(mL,loc&" done");
}

string CpmRootX::toString(ifstream& from)
{
   if (!from) cpmerror("toString(): cannot open input file");
   ostringstream to;
   if (!to) cpmerror("toString(): cannot create output file stream");
   char ch;
   while(from.get(ch)) to.put(ch);
   if (!from.eof() || !to) cpmerror("failure in toString");
   return to.str();
```

```
}

//////////////////// sig ////////////////////////////////////////////////

Z CpmRootX::sig(Z i)
{ if (i>0) return 1; else if (i<0) return -1; else return 0;}

R CpmRootX::sig(R i)
{ if (i>0.) return 1.; else if (i<0.) return -1.; else return 0.;}

R CpmRootX::krn(Z i, Z j){ return i==j ? 1. : 0.;}

Z CpmRootX::parity(Z k){ if (k%2) return -1; else return 1;}

//////////////////////// class R1 ////////////////////////////////

using CpmRootX::R1;
using CpmRootX::Z1;

const R1 R1::max(std::numeric_limits<R>::max());
const R1 R1::eps1(std::numeric_limits<R>::epsilon());
const R1 R1::eps0(std::numeric_limits<R>::min());

const R1 R1::min(-R1::max);
const R1 R1::logmax(::log(R1::max));
const R1 R1::log10max(::log10(R1::max));
const R1 R1::pi(4*::atan(1.));
const R1 R1::piInv(1./R1::pi);

bool R1::prnOn(ostream& out)const
{
   return CpmRoot::write(x1,out);
}

bool R1::scanFrom(istream& in)
{
   return CpmRoot::read(x1,in);
}

Z R1::com(const R1& a)const
{
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   return 0;
}

X2<R,R1 > R1::polDec()const
{
   if (x1>=0) return X2<R,R1 >(x1,R1(1.));
   else return X2<R,R1 >(-x1,R1(-1.));
```

```
}

R R1::disVal(const R1& x)const
{
   return CpmRoot::disVal<R>(x1,x.x1);
}

// functions referring to R as a discrete 'lattice'

R1& R1::next_(void)
{
   R eps_=eps0.x1;
   R xMem=x1;
   while (x1==xMem){
      x1+=eps_;
      eps_*=2;
   }
   return *this;
}

R1& R1::prv_(void)
{
   R eps_=eps0.x1;
   R xMem=x1;
   while (x1==xMem){
      x1-=eps_;
      eps_*=2;
   }
   return *this;
}

namespace{

   void step_p(R& x, R& e)
   {
      R xMem=x;
      Label:
      x+=e;
      if (x==xMem){
         e*=2;
         goto Label;
      } // thus we end up with x!=xMem
   }

   void step_m(R& x, R& e)
   {
      R xMem=x;
      Label:
      x-=e;
      if (x==xMem){
```

```
         e*=2;
         goto Label;
      } // thus we end up with x!=xMem
   }
}

Z R1::dz(R a)const
{
   Z res=0;
   R e=R1::eps0.x1;
   if (x1>=a){
      while (x1>a){
         step_p(a,e);
         ++res;
      }
   }
   else{
      while (x1<a){
         step_m(a,e);
         --res;
      }
   }
   return res;
}

R R1::eps(Z dir)const
{
   R res= (x1>=1 || x1<=-1) ? eps1.x1 : eps0.x1;
   if (dir<0) res*=-1;
   R xc=x1;
   Label:
   xc+=res;
   if (xc==x1){
      res*=2;
      goto Label;
   }
   return res;
}

R R1::dr(R a)
{
   if (a==x1) return 0;
   R dis=x1-a;
   if (dis<0) dis*=-1;
   if (x1>a){
      R ex=eps(-1);
      R ea=R1(a).eps(1);
      return 2*dis/(ea-ex);
   }
   else{
```

```
      R ex=eps(1);
      R ea=R1(a).eps(-1);
      return 2*dis/(ex-ea);
   }
}

Z R1::run(R final, ostream& str, Z writePeriod)
{
   R xMem=x1;
   R e=R1::eps0.x1;
   Z wrt=0;
   Z count=0;
   if (final>x1){
      while(x1<final){
         step_p(x1,e);
         count++;
         if (count==writePeriod){
            str<<"x1-xMem="<<x1-xMem<<" e="<<e<<endl;
            ++wrt;
            count=0;
         }
      }
   }
   else if (final<x1){
      while(x1>final){
         step_m(x1,e);
         count++;
         if (count==writePeriod){
            str<<"x1-xMem="<<x1-xMem<<" e="<<e<<endl;
            ++wrt;
            count=0;
         }
      }
   }
   else;
   return wrt;
}

// save functions

R R1::log()const
{
   if (x1>0) return ::log(x1);
   else if (x1<0) return ::log(-x1);
   else return -logmax;
}

R R1::log10()const
{
   if (x1>0) return ::log10(x1);
```

```
   else if (x1<0) return ::log10(-x1);
   else return -log10max;
}

R R1::exp()const
{
   if (x1>logmax) return max;
   else if (x1<-logmax) return 0;
   else return ::exp(x1);
}

R1 R1::inv()const
{
   R e0=eps0;
   if (x1>e0 || x1<-e0) return R1(1./x1);
   else if (x1>=0) return R1(max);
   else return R1(min);
}

//////////////////////// class Z1 ////////////////////////////////

const Z1 Z1::max(std::numeric_limits<Z>::max());
const Z1 Z1::min(std::numeric_limits<Z>::min());

Z1 Z1::pow(Z p)const
{
   cpmassert(p>=0,"Z1 Z1::power(Z p)");
   if (p==0) return Z1();
   else if (p==1) return Z1(x1);
   else if (p==2) return Z1(x1*x1);
   else{
      Z res=x1;
      Z nMult=p-1;
      for (Z i=1;i<=nMult;i++) res*=x1;
      return Z1(res);
   }
}

bool Z1::prnOn(ostream& out)const
{
   return CpmRoot::write(x1,out);
}

bool Z1::scanFrom(istream& in)
{
   return CpmRoot::read(x1,in);
}

Z Z1::com(const Z1& a)const
{
```

```
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   return 0;
}

X2<Z,Z1 > Z1::polDec()const
{
   if (x1>=0) return X2<Z,Z1 >(x1,Z1(1));
   else return X2<Z,Z1 >(-x1,Z1(-1));
}

R Z1::disVal(const Z1& x)const
{
   return CpmRoot::disVal<Z>(x1,x.x1);
}

bool Z1::isPowOfTwo()const
      // Let us write x for *this, and let N={1,2,3,...} (no zero!).
      // We return true iff x=2^n with some n \in N
{
   if (x1<2) return false;
   Z f=2;
   while (f<x1) f*=2;
   return f==x1;
}

Z1 Z1::nextPowOfTwo()const
{
   if (x1<2) return Z1(2);
   Z f=2;
   while (f<x1) f*=2;
   return Z1(f);
}

Z1 Z1::nextLog2()const
{
   Z res=1;
   Z f=2;
   while (f<x1){ // after the loop f>=x1
      res++;
      f*=2; // guarantees that f grows over all bounderies
   }
   return Z1(res);
}

//////////////////////// class B ////////////////////////////////
using CpmRootX::B;

bool B::prnOn(ostream& out)const
{
```

```
   return CpmRoot::write(x1,out);
}

bool B::scanFrom(istream& in)
{
   return CpmRoot::read(x1,in);
}

Z B::com(const B& a)const
{
   if (x1<a.x1) return 1;
   if (x1>a.x1) return -1;
   return 0;
}
```

210 cpmuc.h

```
//? cpmuc.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_UC_H_
#define CPM_UC_H_
/*

Purpose: Class UseCount is a tool for implementing reference
counting

Credit: Modified from Andrew Koenig: Ruminations on C++,
AT&T Ch. 7. Authoritative and enlightening treatment. However,
compared to the present version, some essentials are missing
in Koenig's book. Desastrous misprints in the code there!

misprints in Koenig:
(p. 72 reattach():
replace --p by --p*,
p. 73 makeonly(): eliminate p=new ...; )
*/

#include <cpmbasictypes.h>

//////////////////////////// class UseCount ////////////////////////////////

namespace CpmArrays{

   using CpmRoot::Z;

class UseCount { // supporting copy on write
   // use count class capable of supporting copy on write
   // class UseCount of Koenig. See explanations there.
public:
   UseCount():p(new Z(1)){}
   UseCount(UseCount const& u):p(u.p){++*p;}
   ~UseCount(){ if (--*p==0) delete p;}
   bool only(){return *p==1;}
   bool makeonly(){if (*p==1) return false; --*p; return true;}
      // named unalias() by Bruce Eckel (p. 437)
      // notice that calling makeonly with a true result will be
      // followed by actions that are anticipated in the --*p statement.
      // (creating a copy which will take one use unit with it)
   bool reattach(UseCount const&);
   void startNew(){ p=new Z(1);}
      // my addition
   Z getCount(void)const{return *p;}
```

```
private:
   UseCount& operator=(UseCount const&);
      // not implemented,to make sure that this will never be used
   Z* p;
};

//////////////////////  class P< >  ////////////////////////////////////
// class of constant smart pointers, no arrays (thus no indexing),
// reference counting, but no copy on write. Tool for implementation
// of function class F<X,Y>.
// see cpmp.h for a derived class which implements copy on write

template <class T>
   // T is a any non-abstract class or any built-in type.
   // If T is immutable, i.e. an instance of T cannot be modified after
   // it
   // has been created, (see Andrew Koenig: Ruminations on C++, AT&T
   // 1997, p. 178, 190 for the relevance of immutable types) P<T>
   // implements the 'strict value interface' (see file cpmv.h).
   // In the notation of predicate logic:
   // non-abstract T & immutable T ==> value class P<T>
   //
   // Let T(A const& a) be a constructor of type T, then
   // the normal syntax for creating an instance of P<T> is
   // A a=...;
   // P<T> pt=new T(a)
   // or
   // P<T> pt;
   // pt=P<T>(new T(a));
   // or
   // P<T> pt(new T(a));

   class P { //  'P' as in 'pointer', constant smart pointer
   // Handle class the instances of which have to be initialized by
   // 'new'-generated pointers. The class takes the full responsibility
   // for deleting these pointers properly.

public:

   P(T *p_=0):p(p_){}
     // { if (p_!=0) cpmassert((long)p==(long)p_,"P(T* p_=0)");}
      // Constructor, including default constructor.
      // important !!!: use this constructor only
      // for pointers initialized with a 'new' statement (on free store,
      // not static!) Important also: This allows to create P<T>'s from
      // pointers X* x where X is derived from T.
      // P<T>(new X(..arguments..)) will hold all information, that
      // the object X x(..arguments..) contains allthough T does not know
      // about the additional members of X.
```

```
   P(P<T> const& h):u(h.u),p(h.p){}
      // copy constructor

   virtual ~P(){ if (u.only()) delete p; }
      // destructor

   P<T>& operator=(P<T> const& h);
      // assignment

// access operator, which does not allow to change p
   T const& operator()(void)const{ return *p;}
      // Let pt an instance of P<T>; then it is conventional to denote
      // the value as *pt. According to my aim to avoid pointers in
      // public
      // interfaces, I also tend to avoid pointer notation. So I denote
      // the value of pt by pt(). This looks very natural since 'getting
      // to the data content' of a container-like quantity is much like
      // evaluating a function. Since no argument is needed (as for
      // standard rand(), for example) a void pair of brackets gives the
      // right association. Assume that T = V<X>, then we may form
      // pt()[1]. In dereferencing notation we had to write (*pt)[1].

   Z loc()const{ return (Z)p;}
      // returns the location in storage, expressed
      // by an integer number. This shall not be used to retrieve
      // this location, so it is only expect to be different for
      // different objects allocated in an application. Thus Z
      // should be an appropriate typing also in 64-bit systems.
protected:
   T *p;
   UseCount u;
};

//////////////////////////// Implementation ////////////////////////

template <class T>
P<T>& P<T>::operator=(P<T> const& h)
{
   if (u.reattach(h.u)) delete p;
   p=h.p;
   return *this;
}

} // namespace

#endif
```

211 cpmuc.cpp

```
//? cpmuc.cpp
//? Status of work 2008-10-25.
//?
#include <cpmuc.h>   // see for explanations

bool CpmArrays::UseCount::reattach(const UseCount& u)
{
   ++*u.p;
      // no conflict with constantness !!
   if (--*p==0){     // --p==0 in Koenigs book
      delete p;
      p=u.p;
      return true;
   }
   else{
      p=u.p;
      return false;
   }
}
```

212 cpmunits.h

```
//? cpmunits.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_UNITS_H_
#define CPM_UNITS_H_

// Values of non-SI physical units expressed as multiples of the
// corresponding SI units

// Source: (1) For 'emu' matters: John Crangle, Mike Gibbs: Units and
// unity in magnetism: a call for consistency, Physics World November
// 1994 p. 31-32
// (2) Abramowitz, Stegun: Handbook of Mathematical Functions, Chapter
// 2.2 (p. 6)
// In the explanation of an entry, we give the SI unit first, and then an
// description of the physical quantity measured in this unit

#include <cpmnumbers.h>

namespace CpmUnits{

   using CpmRoot::R; // real numbers

const R gauss=1e-4;
   // T=Tesla=V*s/m^2 (magnetic polarization J, and magnetic flux density
   // B, B = B0 + J)

const R oersted=1000.;   // divide this number by 4*pi if 'oersted'
// refers to the
   // 'rationalized CGS-system' instead of the 'unrationalized' one.
   // A/m        (magnetic field strength H )

const R emu_per_cm=1000.;
   // J * T^-1 * m^-3 = A/m ( M = Magnetization  =
   // = (Ampere's)magnetic moment 'per unit volume',  B = B0 + mu0*M )
   // notice [M]=[H],  mu0 = 4*pi*1e-7 V*s/(A*m).
   // For convenience, we mention here the sibling of mu0 too:
   // eps0=1/(c^2 * mu0)  = permittivity of vacuum  = dielectric constant
   // of vacuum, c = speed of light in vacuo := 299792458 m/s
   // [eps0] = (A*s)/(V*m)
   // Remember eps0 * mu0 * c^2 =1. Since c is large in SI units , both
   // eps0 and mu0 are very small in SI units:
   // mu0 = 1.256637e-6 H/m (this is to infer H=Henry=V*s/A)
   // eps0 = 8.854188e-12 F/m (F=Farad=A*s/V)
   // Also remember Wb=Weber=T*m^2

const R emu_per_g = 1.;
```

```
   // J/(T*kg) = A * m^2 / kg  ( sigma = M/rho = Ampere's magnetic moment
   // 'per unit mass' ). Example: Electrophotographic carrier particles
   // are reported to have a remanent magnetization of 30 emu/g and a
   // density of 3.5 g/cm^3. What is the polarization J (Coulomb's
   // magnetic moment 'per unit volume') ?
   // sigma=30 emu/g = 30 A*m^2/kg, M = sigma * rho = 30 * 3500 A/m
   // J = M*mu0 = 30 * 3500 * 4pi*1e-7 = 0.132 V*s/m^2 = 0.132 T

const R inch=2.54e-2;
   // m

const R foot=12*inch;
   // m

const R mil=0.001*inch;
   // m

const R gallon=3.785e-3;
   // m^-3
   // abbreviation gal, this is the US gallon, there is a UK gallon too

const R pound=0.4536;
   // kg

const R ounce=pound/16;
   // kg

const R RPM=2*CpmRoot::Pi/60;
   // rad/s

//const R degree=CpmRoot::Pi/180;
   // rad, now CpmGeo::degree defined in cpmangle.h

const R cm=1e-2;
   // m

const R mm=1e-3;
   // m

const R micrometer=1e-6;
   // m

const R nm=1e-9;
   // m

} // namespace

#endif
```

213 cpmv.h

```
//? cpmv.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_V_H_
#define CPM_V_H_
/*
This defines a template V<T> of T-valued lists or 'vectors with
T-valued components'. It is similar to std::vector<T> of the STL
but it differs from this in some respect. In describing
V<T> and these differences, I'll introduce some notions and notations
that will be used over and over in defining and stating properties of
other CPM classes
( CPM = 'C+-' or 'Classes for Physics and Mathematics').

1. V<> instances (called simply 'Vectors' here) never request more
   memory than necessary. Thus appending elements always needs new
   allocation of memory.
   V<> is more adapted to the role of vectors as compound
   (structured) quantities than to the role as a sequential container for
   its components. Therefore, appending components at the end is
   not basic to the concept and is not worth special optimizing effords.
   Notice that V<> is not a 'polymorphic container'
   i.e. the components can hold only T typed objects and not the
   additional information content which instances of classes derived
   from T may carry. We may, however, form V<T*> for any type or class
   to support polymorphism in the old-fashioned pointer-based way.
   A safer way is to use the polymorphic vector template Vp mentioned
   in item 5.

2. V<> implements 'reference counting' and 'copy on write'. Thus
   there is no need to work with references to vectors to avoid
   superfluous copying. Assume that we nevertheless observe the
   rule to declare function arguments always as references
   or constant references. Then, the main effect of reference
   counting is that large objects built within the body of a
   function (such as matrix multiplication) will not be copied
   to the outside but simply referenced. If they are no longer
   needed, they nevertheless get deleted automatically.
   If such a referenced large object gets changed by program
   action and some old client still needs the unchanged version
   for his reference, a copy of the large object is made automatically
   and made available to its user.

3. Requirements on T in order to allow the formation of V<T>:
   To illustrate the point up front: std::vector<T> will only be built
   (at least with the MS Visual C++ 5.0 compiler) if T<T and T==T are
   defined. For V<T> there is no such restriction. To be sure: T<T is an
```

obvious abbreviation of the statement:
'for all objects t1 and t2 of type T, the expression t1<t2 is defined' (compare David R. Musser, Atul Saini: STL Tutorial and Reference Guide, Addison-Wesley 1996, p. 246). Now the pertinent statement: V<T> is well defined if (not iff!) T is a built-in type or a class which 'implements the value interface'. Then, V<T> also implements the value interface. If, moreover, T 'implements the strict value interface', then V<T> also implements the strict value interface. Here, a class T is said to implement the value interface if it publicly defines T(), T(const T&), and T& operator=(const T&). T is said to implement the strict value interface if, in addition, it has value semantics (as opposed to pointer semantics, see Andrew Koenig: Ruminations on C++, AT&T 1997, p. 62. and Bjarne Stroustrup: The C++ Programming Language, 3. edition, Addison-Wesley 1997, p. 294). Classes that implement the strict value are most convenient to use. Code only using such quantities is normally easy to understand. To have an even shorter denotation for this favorable species we call such a class a value class or a bit more general:

```
T is a value class : <==> T is a built-in type or a class
that implements the strict value interface.
```

As was stated already, we have:

```
value class T ==> value class V<T>
```

The predicates 'T satisfies the value interface' and 'T satisfies the strict value interface' can be evaluated for classes providing random generators by means of the test classes Test_v<T> and Test_sv<T> defined in file cpmtestr.h. Although V<> provides no random generator (Vr<>, to be mentioned soon, does) the code of Test_sv<T> can be read as an operational definition of the concept 'strict value interface'. An interesting (or common-place ?) observation: syntactic aspects of a program are characterized by the program's interaction with the compiler; semantic aspects ('pointer sematics',...) are characterized by the data created during execution.

Requiring order operators or even arithmetic operators in T, allows to define T-'valued' Vectors which themselves carry natural order/arithmetic operators.
Therefore V is only the starting point in a series of vector templates with increasing functionality, accompanied by increasing assumptions on the structure of T. Presently this hierarchy looks as follows:

```
V<T> , Vo<T> , Va<T> , Vr<T>
```

Every such class is derived from its predecessor in this series

```
   by derivation without adding new data members. Thus there are
   unambiguous casts between all these classes.

      Vo : Order related methods added to V
      Va : arithmetic operations and I/O operations added to Vo
      Vr : rich infrastructure supporting automated testing of
         internal consistency.

   This approach of integrating the functions (algorithms) into an
   inheritance tree of template classes is radically different from the
   STL approach. STL keeps algorithms as separate entities outside the
   classes and uses iterators and adaptors for making them work together.
   Both methods have their advantages and disadvantages. It is probably
   fair to say that the C+- approach is less universal but more
   convenient to use within the framework it fits.

4. Iterating over the components of a vector is simply done by
   a loop over the components. E.g.

   for (int i=1; i<=v.dim(); i++) v[i]=a[i]+b[i];

   Notice the start of the loop with 1 instead the C native 0.
   The access to components via operator[] involves range check.
   The present implementation of V is based on a 'lean' template Vl<>
   having native C index range. A set of member functions fa,...fg
   of this class helps to code algoritms in a way that the underlying
   pointers are used directly and not via operator []. The speed gained
   by this seems to be marginal. The previous loop can be written in a
   format which makes no difference between the situation that v,a,b
   are all of type V<T> or all of type Vl<T>:

   for (Z i=v.b(); i<=v.e(); ++i) v[i]=a[i]+b[i];

5. On polymorphism: V<T*> may be formed, but doesn't implement the value
   interface for T and derived classes. However, with the smart pointer
   templates P<T>, Pp<T>, and Po<T> defined in file cpmp.h we can form
   e.g. V<Pp<T> > and get the functionality (together with some 'extras')
   which one would expect from  V<T*>. See file cpmp.h for details and
   the polymorphic version Vp of the vector templates mentioned so far.
   See test class PolymorphicMulti<*,*,*> in cpmtestr.h for a operational
   definition of polymorphism of container classes.
*/

#include <cpmvl.h>

///////////////// class V<> ////////////////////////////////////////////
// array with index range check (first valid index is 1, not 0),
// reference counting, and copy on write, and 'value semantics'
// (as opposed to 'pointer semantics')
```

```
namespace CpmArrays{

template <class T>
   // We assume that T provides (explicitely or implicitely)
   // copy constructor, and assignment
   // or that T is a built-in type.

class V{ // vector template, indexing starts with 1
   // since the only data member belongs to a class which 'implements
   // the value interface' (i.e. defines copy constructor,
   // and assignment operator), there is no need to define these
   // operators here (will be done correctly by the compiler).

public:
   typedef V<T> Type;
   CPM_IO

explicit V(Z n=0):vl_(n){}
      // initialization also for built-in types.

explicit V(Vl<T> const& vl):vl_(vl){}
      // means to transfer results from Vl<T> to V<T>

explicit V(std::vector<T> const& v):vl_(Vl<T>(v)){}
      // efficient way to get the content of a STL-vector into a
      // cpm-vector

   V(Z n, T const& t):vl_(n,t){}
      // all n components are equal to t

   V(Z n, F<Z,T> const& f);
      // construction from a function. Of course,
      // V<T> v(n,f);
      // implies v[i]==f(i) for all valid indexes i of v.

// constructors from explicit lists
// The first argument avoids to mistake such an constructor with any
// other constructor for any choice of T (simply 'Z != Word')
// For instance
// V<Z> v("",1,-1,2,-2,3,-3);
// defines a V<Z> instance v with v.dim()==6, v[1]=1,v[2]=-1,...,v[6]=-3.
// This is nearly as comfortable as initializing C-arrays. The number of
// arguments is limited so far to 10, could be extended.
// Extremely convenient in many programming situations!

   V(Word ind, T t1);
   V(Word ind, T t1, T t2);
   V(Word ind, T t1, T t2, T t3);
   V(Word ind, T t1, T t2, T t3, T t4);
   V(Word ind, T t1, T t2, T t3, T t4, T t5);
```

```
   V(Word ind, T t1, T t2, T t3, T t4, T t5, T t6);
   V(Word ind, T t1, T t2, T t3, T t4, T t5, T t6, T t7);
   V(Word ind, T t1, T t2, T t3, T t4, T t5, T t6, T t7, T t8);
   V(Word ind, T t1, T t2, T t3, T t4, T t5, T t6, T t7, T t8, T t9);
   V(Word ind, T t1, T t2, T t3, T t4, T t5, T t6, T t7, T t8, T t9,
      T t10);

   virtual V<T>* clone(void)const{ return new V(*this);}

   V<T> toClnBase()const{ return V<T>(vl_);}
      // to clone base

   std::vector<T> std()const{ return vl_.std();}
      // efficient way to create a STL-vector from a cpm-vector

   Z dim(void)const{ return vl_.dim();}

   virtual Word nameOf()const{
      Word wi="V<";
      Word nt=CpmRoot::Name<T>()(T());
      return wi&nt&">";
   }

   virtual Z size(void)const{ return vl_.dim();}
      // for uniformity with STL-code, should be
      // re-defined in classes of multi-indexed quantities
      // to mean the total number of components

   bool isVoid()const{ return vl_.isVoid();}
      // short answer on whether the dimension is zero

   bool valInd(Z i)const{ return vl_.valInd(--i);}
      //: valid index
      // returns the validity of i as an index of *this

   IvZ dom()const{ return vl_.dom()+1;}
      //: domain ; natural domain for a function i->(*this)[i]

// access by operator[]. Index range in 1,2,....dim(), not
// 0,1,...dim()-1, as in class Vl<T>.
// Index access is checked (by class Vl<T> ).

   const T& operator[](Z i)const{ return vl_[--i];}
      // read access

   T& operator[](Z i){ return vl_[--i];}
      // write access

   T const& cyc(Z i)const{ return vl_.cyc(--i);}
   T& cyc(Z i){ return vl_.cyc(--i);}
```

```
T const& cui(Z i)const{ return vl_.cui(--i);}
T& cui(Z i){ return vl_.cui(--i);}
   // unchecked indexing, sometimes good for speed, and since it
   // does not assume nameOf<T> being defined

T const& con(Z i)const{ return vl_.con(--i);}
T& con(Z i){ return vl_.con(--i);}

T const& read(Z i, Outside mode=DEFAULT)const
   { return vl_.read(--i,mode);}
   // reading components in an efficient (as &'s), safe,
   // and flexible manner
   // mode DEFAULT: values outside assumed to be given as T()
   // mode CYCLIC: values outside defined by cyclic continuation
   // else: values outside assumed to be given by constant continuation

T& first(){ return vl_.fir();}
   // old name
T& fir(){ return vl_.fir();}
   //: first

T const& first()const{ return vl_.fir(); }
   // old name
T const& fir()const{ return vl_.fir(); }
   //: first

T& last(){ return vl_.last(); }
   //: last

T const& last()const{ return vl_.last(); }
   //: last

T operator()(Z i, Outside mode=DEFAULT)const
   // read access defined for all i. By this definition, a vector
   // becomes a mapping from Z to T. For i's outside the proper range,
   // the default T is returned for periodic==false
   // and cyc(i) for periodic==true.
   // Notice that the return value is not a reference in
   // accordance with function behavior.
   { return vl_(--i,mode);}

F<Z,T> fnc(Outside mode=DEFAULT)const;
   // returns the function Z --> T, i |--> (*this)(i,periodic)

bool get(Z i, T& x)const;
   // for i out of range we return false and don't change x.
   // else we have x=(*this)[i] after call and true gets returned.

template <class X>
```

```
   V<X> operator()(F<T,X> const& g)const{ return V<X>(vl_(g));}
      // transforming the components by a function

// building new objects from given ones (generative methods)

// concatenating vectors and appending components. These operations have
// always O(sz) complexity.

   void merge(const V< V<T> >& vecList, bool reversed=false);
      // changes *this by appending the vectors of the list either
      // in normal order or in revesed order
      // ( *this changed to *this&vL[1]&...&vL[n] for reversed ==false
      // and *this&vL[n]&...&vL[1] for reversed==true.
      // Classes S<> and M<> have related functions. Attempts to
      // implement this functionality as a constructor or as a
      // static function lead to unexpected difficulties.

   V<T> append(const V<T>& v )const{ return V<T>(vl_.append(v.vl_));}
   V<T> prepend(const V<T>& v)const{ return V<T>(vl_.prepend(v.vl_));}

   V<T> append(const T& t )const{ return V<T>(vl_.append(t));}
   V<T> prepend(const T& t)const{ return V<T>(vl_.prepend(t));}

   V<T> operator &(const V<T>& v )const{ return V<T>(vl_.append(v.vl_));}
   V<T> operator &(const T& t )const{ return V<T>(vl_.append(t));}
// appending as a mutating operation (in analogy to operations in class
// Word). Here the mutating methods are the derived ones, and less
// efficient than the generating ones.
   V<T>& operator &=(const V<T>& v ){ return *this=(*this)&v;}
   V<T>& operator &=(const T& t ){ return *this=(*this)&t;}

   void push_back(T const& t){ vl_.push_back(t);}
      // for uniformity with STL

   void operator<<(T const& t){ vl_<<t; }
      // appending an element in Ruby-style

   void operator<<(V<T> const& v) { vl_<<v.vl_;}
      // appending an array in Ruby-style

   T in_(T const& t, bool reversed=false){ return vl_.in_(t,reversed);}
      // This operations treats *this as a shift register:
      // All components get shifted by one position 'to the right' and
      // the total length (dim) of the vector remains the same. So what
      // was the last component prior to the operation has to be removed
      // from the vector, in order to not wasting information, this
      // removed component will shine up as the return value of the
      // function. After the operation, the first component of the vector
      // will be t.
      // If reverse==true, t gets inputted at the end, and all shift
```

```
      // operations go 'to the left'.

// resizing

   V<T> resize(Z newDim)const{ return V<T>(vl_.resize(newDim));}
      // returns a vector res such that res.dim()=newDim. If newDim is
      // smaller than dim(), the end of *this will be cut away.
      // If newDim is larger, T()'s will be added

   V<T> cut(Z i)const{ return V<T>(vl_.cut(i));}
      // returned is a V<T> which results from *this by removing i
      // components from the end

// re-indexing (composition of vectors considered as mappings)
   V<T> permute(const V<Z>&)const;
      // Let res=v.permute(w), then
      // 1. res.dim() == v.dim();
      // 2. res[i]==v[i] if there is no k such that k=w[i]
      // 3. res[i]==v[k] if k=w[i]

// composition of vectors considered as mappings
// cleaner version of the intent of permute
   V<T> compose(const V<Z>&)const;
      // Let res=v.compose(w), then
      // 1. res.dim() == w.dim();
      // 2. res[i]==v(w[i]);
      // notice that v(i) is defined for all i

// condensation (contracting equivalent components into one)

   pair< V<T>, V<Z> > condense( bool (*equi)(const T&, const T&))const;
      // given an equivalence relation equi on T (t1~t2 <==>
      // equi(t1,t2)==true )
      // we divide the components of *this into equivalence classes.
      // Let the resturned pair be written as (res1,res2) . Then
      // (i) (*this)[i]~res1[res2[i]]
      // (ii) there is a j such that  (*this)[j]==res1[res2[i]]
      // Actually, the construction is made such that this j is the
      // smallest j for which (*this)[j]~res1[res2[i]].

// elimination and insertion

   V<T> eli(Z i, Z nEli=1)const;
      // eli stands for eliminate
      // returned is a vector which is obtained from *this by
      // eliminating the i-th component and the nEli-1 following ones
      // (thus nEli components are removed) and shifting all later
      // components (if there are such components left) 'to the left' to
      // close the gap.
      // The i-th component of *this is (*this)[i] defined by operator[]
```

```
   // Thus component count starts with 1.

V<T> eliFirst()const{ return eli(1,1);}
   // returned is a vector which is obtained from *this by
   // eliminating first component.

V<T> eliLast()const{ return eli(dim(),1);}
   // returned is a vector which is obtained from *this by
   // eliminating the last component.

V<T> eli1(V<T>& h, Z i)const{ return V<T>(vl_.eli1(h.c()));}
   // We return a vector which results from *this by eliminating
   // h.dim() components starting at the i'th component (i understood
   // as above). After the call h will hold the eliminated
   // components in due order. The function name ends in '1'
   // to indicate that the first argument is a non-constant
   // reference, used for communication of a partial result.

V<T> ins(Z i, const T& t)const;
   // ins stands for insert
   // returned is a vector which is obtained from *this by
   // inserting t as the i-th component and shifting all later
   // components 'to the right' to give room for t

V<T> ins(Z i, const V<T>& h)const;
   // returned is a vector which is obtained from *this by
   // inserting h as the i-th and following component,
   // and shifting all later components of *this
   // 'to the right' to give room for h

V<T> rev()const{ return V<T>(vl_.rev());}
   // returned is the reversed vector (indexing in the opposite
   // direction).

Z find_sim(T const& t)const;
   // find simple
   // if the return value is 0, there is no i=1,...n
   // such that (*this)[i]==t. Otherwise, the return value i is
   // such that (*this)[i]==t. The components are compared against t
   // starting from 1. The first occurrence of t gets reported and
   // stops the search.

V<T> select(V<bool> const& s)const
   { return V<T>(vl_.select(s.c()));}
   // returned is a list which is obtained from *this by eliminating
   // all components (*this)[i] for which s[i] is defined and
   // statisfies s[i]==false. Beside of this removal, the order of the
   // components in *this is retained. So if s is defined
   // by a condition s[i]=Condition((*this)[i]), for the vector
   // component, this condition has to express a property we like to
```

```
      // have fulfilled for the result-vector of the select-operation.
      // 's expresses the favorable condition' and  n o t  the one to be
      // eliminated. Notice, that the select operation makes sense for
      // all values of s.dim().

   pair<V<T>,V<Z> > select2(V<bool> const& s)const;
      // Let V<T> x=...;
      // V<bool> s=...;
      // V<T> xT=x.select2(s).first;
      // V<Z> xZ=x.select2(s).second;
      // bool b1=(xT.dim()==xZ.dim());
      // bool b2=(xT[i]==x[xZ[i]]);
      // always b1==true and
      // 1 <= i <= xT.dim() ==>  b2==true
      // In effect we return a list sel such that
      // x.select(s) equals x.permute(sel) and x.compose(sel).
      // The present function creates both objects in an efficient
      // way in one run.

    void set_(const T& t){ vl_.set_(t);}
      // setting all components of *this equal to t

    V<T> set(Z i, const T& t){ return V<T>(vl_.set(i,t));}
      // ( i in vl_.set(i,t) refers to 'natural counting', so this is
      // OK!)

   Vl<T>& c(){ return vl_;}
      // returning a reference to vl_ for change
      // accessing vl_ is usefull when using functions
      // that expects 0-starting arrays

   Vl<T> const& c()const{ return vl_;}
      // returning a const reference to vl_ for efficiency

   Z b()const{ return 1;}
      // begin
   Z e()const{ return vl_.dim();}
      // end
      // allows to write loops over all components as
      // for (Z i=v.b();i<=v.e();i++) v[i]=...;

   V<IvZ> valOn(F<T,bool> const& f)const;

protected:
   Vl<T> vl_;
      // representation by a Vl<T>.
};

inline V<Word> comLine(int argc, char* argv[])
//: command line
```

```
// definition needs to be inline since it has to be in the header file.
// Otherwise we would need a file cpmv.cpp - only for defining
// this function.
{ return V<Word>(comLineLean(argc,argv));}

namespace{
   Z fpp(Z i){ return ++i;}
   // notice: we need fpp(0)==1
   Z fmm(Z i){ return --i;}
}

template <class T>
V<T>::V(Z n, F<Z,T> const& f):vl_(n,F<Z,Z>(fpp)&f){}
// v[1]=vl_[0] should be f(1) thus (fpp&f)(0)==f(1);
// Now (fpp&f)(0)==f(fpp(0))==f(1) if fpp(0)==1

template <class T>
F<Z,T> V<T>::fnc(Outside mode)const
// for V<T> f one should have f.fnc(1) = vl_[0]=vl_.fnc(0)
// Thus fpp&(f.fnc())==vl_.fnc(). Hence f.fnc = fmm&vl_.fnc
{
   return F<Z,Z>(fmm)&vl_.fnc(mode);
}

template <class T>
V<IvZ> V<T>::valOn(F<T,bool> const& f)const
{
   V<IvZ> r(vl_.valOn(f)); // V<IvZ> from Vl<IvZ>
   for (Z i=r.b();i<=r.e();++i) r[i]+=1;
   return r;
}

template <class T>
V<T>::V(Word ind, T t1):vl_(1)
{
   ind;
   vl_[0]=t1;
}

template <class T>
V<T>::V(Word ind, T t1, T t2):vl_(2)
{
   ind;
   vl_[0]=t1;
   vl_[1]=t2;
}

template <class T>
V<T>::V(Word ind, T t1, T t2, T t3):vl_(3)
{
```

```
   ind;
   vl_[0]=t1;
   vl_[1]=t2;
   vl_[2]=t3;
}

template <class T>
V<T>::V(Word ind, T t1, T t2, T t3, T t4):vl_(4)
{
   ind;
   vl_[0]=t1;
   vl_[1]=t2;
   vl_[2]=t3;
   vl_[3]=t4;
}

template <class T>
V<T>::V(Word ind, T t1, T t2, T t3, T t4, T t5):vl_(5)
{
   ind;
   vl_[0]=t1;
   vl_[1]=t2;
   vl_[2]=t3;
   vl_[3]=t4;
   vl_[4]=t5;
}

template <class T>
V<T>::V(Word ind, T t1, T t2, T t3, T t4, T t5, T t6):vl_(6)
{
   ind;
   vl_[0]=t1;
   vl_[1]=t2;
   vl_[2]=t3;
   vl_[3]=t4;
   vl_[4]=t5;
   vl_[5]=t6;
}

template <class T>
V<T>::V(Word ind, T t1, T t2, T t3, T t4, T t5, T t6, T t7):vl_(7)
{
   ind;
   vl_[0]=t1;
   vl_[1]=t2;
   vl_[2]=t3;
   vl_[3]=t4;
   vl_[4]=t5;
   vl_[5]=t6;
   vl_[6]=t7;
```

```
}

template <class T>
V<T>::V(Word ind, T t1, T t2, T t3, T t4, T t5, T t6, T t7, T t8):vl_(8)
{
   ind;
   vl_[0]=t1;
   vl_[1]=t2;
   vl_[2]=t3;
   vl_[3]=t4;
   vl_[4]=t5;
   vl_[5]=t6;
   vl_[6]=t7;
   vl_[7]=t8;
}

template <class T>
V<T>::V(Word ind, T t1, T t2, T t3, T t4, T t5, T t6, T t7, T t8,
      T t9):vl_(9)
{
   ind;
   vl_[0]=t1;
   vl_[1]=t2;
   vl_[2]=t3;
   vl_[3]=t4;
   vl_[4]=t5;
   vl_[5]=t6;
   vl_[6]=t7;
   vl_[7]=t8;
   vl_[8]=t9;
}

template <class T>
V<T>::V(Word ind, T t1, T t2, T t3, T t4, T t5, T t6, T t7, T t8,
      T t9, T t10):vl_(10)
{
   ind;
   vl_[0]=t1;
   vl_[1]=t2;
   vl_[2]=t3;
   vl_[3]=t4;
   vl_[4]=t5;
   vl_[5]=t6;
   vl_[6]=t7;
   vl_[7]=t8;
   vl_[8]=t9;
   vl_[9]=t10;
}

template <class T>
```

```
bool V<T>::get(Z i, T& x)const
{
   if (i<1 || i>dim()) return false;
      // now i is in range
   x=vl_.cui[i-1];
   return true;
}

// implementation taken from Vl<T>:

// re-indexing

template <class T>
V<T> V<T>::permute(const V<Z>& w)const
{
   Z n=w.dim(),nt=dim(),i,j;
   V<T> res=*this;
   for (i=1;i<=n;i++){
      j=w[i];
      if (j<1 || j>nt) cpmerror
         ("V<T> V<T>::permute(const V<Z>&): index out of range");
      res[i]=(*this)[j];
   }
   return res;
}

template <class T>
V<T> V<T>::compose(const V<Z>& w)const
{
   Z n=w.dim(),i;
   V<T> res(n);
   for (i=1;i<=n;i++){
      res[i]=operator()(w[i]);
   }
   return res;
}

template <class T>
pair< V<T>, V<Z> > V<T>::condense(
   bool (*equiv)(const T&, const T&))const
// implementation based on function eclazz of the Numerical Receipes of
// Press et al.
{
   const Z mL=3;
   static Word loc("V<T>::condense()");
   CPM_MA
   Z n=dim();
   if (n<1){ // addition 2002-02-23
      CPM_MZ
      return pair< V<T>, V<Z> >(V<T>(0),V<Z>(0));
```

```
   }
   V<Z> res2(n);
   Z k,j;
   res2[1]=1;
   for (j=2;j<=n;j++) {
      res2[j]=j;
      for (k=1;k<=(j-1);k++) {
         res2[k]=res2[res2[k]];
         if ((*equiv)((*this)[j],(*this)[k])) res2[res2[res2[k]]]=j;
      }
   }
   for (j=1;j<=n;j++) res2[j]=res2[res2[j]];
   Z m=-1;
   Z rj;
   for (j=1;j<=n;j++){
      rj=res2[j];
      if (rj>m) m=rj;
   }
   V<Z> aux(m,0); // unfortunately the NR algorithm does not guarantee
      // that there are no gaps between the valid values of rj. Therfore
      // we find out the valid ones by an additional loop
   V<Z> found(m,0);
   for (j=1;j<=n;j++){
      rj=res2[j];
      if(found[rj]==0){
         found[rj]=1;
         aux[rj]=j;
      }
   }
   Z mAct=0;
   for (j=1;j<=m;j++) mAct+=found[j];
   V<T> res1(mAct);
   Z jAct=1;
   for (j=1;j<=m;j++){ // notice that aux[j] was initialized as 0
      if (aux[j]>0) res1[jAct++]=(*this)[aux[j]];
   }
   CPM_MZ
   return pair< V<T>, V<Z> >(res1,res2);
}

template <class T >
Z V<T>::find_sim(T const& t)const
{
   Z n=dim();
   if (n==0) return 0;
   Z iRes=0; // correct result if the next loop doesn't find t
   for (Z i=1;i<=n;i++){
     if ((*this)[i]==t){ iRes=i; break;}
   }
   return iRes;
```

```
}

// elimination, insertion

template <class T>
V<T> V<T>::eli(Z i, Z nEli)const
{
   return V<T>(vl_.eli(i,nEli));
}

template <class T>
V<T> V<T>::ins(Z i, const V<T>& h)const
{
   return V<T>(vl_.ins(i,h.vl_));
}

template <class T>
V<T> V<T>::ins(Z i, const T& t)const
{
   return V<T>(vl_.ins(i,t));
}

template <class T >
pair<V<T>,V<Z> > V<T>::select2(const V<bool>& s)const
{
   Z i,n=dim(),ns=s.dim();
   V<T> resPrelT(n); // the actual result may have less components
   V<Z> resPrelZ(n); // the actual result may have less components
   Z iAct=0;
   for (i=1;i<=n;i++){
      bool keepIt= (i<=ns ? s[i] : true);
      if (keepIt){
         T tAct=(*this)[i];
         iAct++;
         resPrelT[iAct]=tAct;
         resPrelZ[iAct]=i;
      }
   }
   if (iAct==n)
      return pair< V<T>,V<Z> >(resPrelT,resPrelZ);
      // no element was purged
   V<T> resT(iAct);
   V<Z> resZ(iAct);

   for (i=1;i<=iAct;i++){
      resT[i]=resPrelT[i];
      resZ[i]=resPrelZ[i];
   }
   return pair< V<T>,V<Z> >(resT,resZ);
}
```

```
template <class T >
void V<T>::merge(const V< V<T> >& vecList, bool reversed)
{
   Z i,d=vecList.dim();
   if (!reversed)
      for (i=1;i<=d;i++) vl_=vl_.append(vecList[i].vl_);
   else
      for (i=d;i>=1;i--) vl_=vl_.append(vecList[i].vl_);
}

template <class T>
bool V<T>::prnOn(ostream& str)const
{
   cpmwbt; // write automatic begin indicator
   bool wrtTitMem=CpmRoot::wrtTit;
   CpmRoot::wrtTit=false; // it could be tedious to see again the
   // title now for Vl instead of V. This, however, switches of
   // title writing for all components (of components ...). This
   // implements the the idea that titles are not indispensible but
   // if they are there, they have to be correct. So it would not be
   // acceptable to let V not write its own title, thus pretending that
   // the object would be a Vl.
   cpmp(vl_);
   CpmRoot::wrtTit=wrtTitMem;
   cpmwet; // write automatic end indicator
   return true;
}

template <class T>
bool V<T>::scanFrom(istream& str)
{
   cpms(vl_);
   return true;
}

/************************ iterated templates *************************/
// describing multi-indexed quantities
// Notice that these all have automatically the member functions of V
// defined!

#define CPM_V1 V<T>
#define CPM_V2 V<V<T> >
#define CPM_V3 V<V<V<T> > >
#define CPM_V4 V<V<V<V<T> > > >

/************************* class VV ***********************************/
template <class T>
class VV: public CPM_V2{ // matrices
```

```
   typedef CPM_V2 Base;

public:

// constructors

   VV(Z d1, Z d2):CPM_V2(d1,CPM_V1(d2)){}
   VV(Z d1, Z d2, const T& t):CPM_V2(d1, CPM_V1(d2,t)){}
      // all components are equal to t
   VV(void):CPM_V2(){}
   VV(const CPM_V2& x):CPM_V2(x){}

   V<T> lin()const;
      // 'linear version of matrix'
      // by appending all rows

// dimension access
   Z dim1()const{ return Base::dim();}
   Z dim2()const{ return (*this)[1].dim();}

   virtual Z size()const // virtual in base V<...>
      { return dim1()==0 ? 0 : dim1()*dim2();}

// component access

   const T& operator()(Z i, Z j)const
   { return (*this)[i][j];}

   T& operator()(Z i, Z j)
   { return (*this)[i][j];}
};

template <class T>
V<T> VV<T>::lin()const
{
   if (dim1()==0) return V<T>(0);
   else{
      V<T> res=(*this)[1];
      for (Z i=2;i<=dim1();i++) res&=(*this)[i];
      return res;
   }
}

/************************* class VVV***********************************/
// tensors of rank 3

template <class T>
class VVV: public CPM_V3{ // tensors of rank 3

      typedef CPM_V3 Base;
```

```
public:

// constructors

   VVV(Z d1, Z d2, Z d3, const T& t):CPM_V3(d1,CPM_V2(d2,CPM_V1(d3,t))){}
   VVV(Z d1, Z d2, Z d3):CPM_V3(d1,CPM_V2(d2,CPM_V1(d3))){}
   VVV(void):CPM_V3(){}
   VVV(const CPM_V3& x):CPM_V3(x){}

// dimension access
   Z dim1()const{ return Base::dim();}
   Z dim2()const{ return (*this)[1].dim();}
   Z dim3()const{ return (*this)[1][1].dim();}

   virtual Z size()const; // virtual in base V<...>

// component access

   const T & operator()(Z i, Z j, Z k)const
   { return (*this)[i][j][k];}

   T & operator()(Z i, Z j, Z k)
   { return (*this)[i][j][k];}
};

template <class T>
Z VVV<T>::size()const
{
   Z d1=dim1();
   if (d1==0) return d1;
   else{
      Z d2=dim2();
      if (d2==0) return d2;
      else{
         Z d3=dim3();
         if (d3==0) return d3;
         else return d1*d2*d3;
      }
   }
}

/************************ class VVVV ***********************************/
template <class T>
class VVVV: public CPM_V4{ // tensors of rank 4

      typedef CPM_V4 Base;

public:
```

```
// constructors

   VVVV(Z d1, Z d2, Z d3, Z d4, const T& t):
   CPM_V4(d1,CPM_V3(d2,CPM_V2(d3,CPM_V1(d4,t)))){}
   VVVV(Z d1, Z d2, Z d3, Z d4):
   CPM_V4(d1,CPM_V3(d2,CPM_V2(d3,CPM_V1(d4)))){}
   VVVV(void):CPM_V4(){}
   VVVV(const CPM_V4& x):CPM_V4(x){}

// dimension access
   Z dim1()const{ return Base::dim();}
   Z dim2()const{ return (*this)[1].dim();}
   Z dim3()const{ return (*this)[1][1].dim();}
   Z dim4()const{ return (*this)[1][1][1].dim();}

   virtual Z size()const; // virtual in base V<...>

// component access

   T const& operator()(Z i, Z j, Z k, Z l)const
   { return (*this)[i][j][k][l];}

   T& operator()(Z i, Z j, Z k, Z l)
   { return (*this)[i][j][k][l];}
};

template <class T>
Z VVVV<T>::size()const
{
   Z d1=dim1();
   if (d1==0) return d1;
   else{
      Z d2=dim2();
      if (d2==0) return d2;
      else{
         Z d3=dim3();
         if (d3==0) return d3;
         else{
            Z d4=dim4();
            if (d4==0) return d4;
            else return d1*d2*d3*d4;
         }
      }
   }
}
#undef CPM_V4
#undef CPM_V3
#undef CPM_V2
#undef CPM_V1
```

```
} // namespace

#endif
```

214 cpmva.h

```
//? cpmva.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_VA_H_
#define CPM_VA_H_
/*
   Purpose: Defining a array class Va<T> which exploits the assumtion that
   T supports arithmetics. See cpmv.h
*/
#include <cpmvo.h>
#include <cpmtypes.h>

//////////////////////// class Va<> ////////////////////////////

namespace CpmArrays{

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::Word;
   using CpmRootX::inf;

template <class T>
   // We assume that T provides (explicitely or implicitely)
   // copy constructor, and assignement
   // or that T is a built-in type.
   // T > T, T < T, T += T, -T, T *= T, T /=T, ostream << T, istream >> T
   // further we assume that a_clone<T> can be built

class Va: public Vo<T>{ // version of Vo with arithmetic operations

public:
   typedef Va<T> Type;
   typedef T ScalarType;
   typedef Vo<T> Base;

explicit   Va(Z n=0):Vo<T>(n){}
      // note that Va(0) is defined and is different from Va()

   Va(Z n,T const& t):Vo<T>(n,t){}
      // initializes all n components with t

   Va(const V<T>& h):Vo<T>(h){}
   // 'down-cast'-constructor

   Va(const Vo<T>& h):Vo<T>(h){}
   // 'down-cast'-constructor
```

```
// access to the functionality of Va in a framework in which
// Vl is considered basic. Notice that the conversion
// from Va back to Vl gets performed gy the function c()
// which is defined in V
   explicit Va(Vl<T> const& vl):Vo<T>(V<T>(vl)){}

   virtual Word nameOf()const{
      Word wi="Va<";
      return wi&CpmRoot::Name<T>()(T())&">";
   }

   T sum(void)const;
      // returns the sum over all components

   R frac(T const& t)const;
      // returns nt/dim() where nt=card{ j | (*this)[j]==t}
      // For dim()==0 we return 0.

   R frac(const V<T>& v)const;
      // returns nt/dim() where nt=card{ j | (*this)[j]==v[k] for some k}
      // For dim()==0 or v.dim()==0 we return 0.

   virtual V<T>* clone(void)const{ return new Va(*this);}

   CPM_SUM_M
   CPM_PRODUCT_M
   CPM_DIFFERENCE
   CPM_DIVISION
   CPM_SCALAR_M
   CPM_DOT_PRODUCT
   CPM_CONJ
   CPM_NORMALIZE
   CPM_IO

private:
   static T sumFunc(T const& t1, T const& t2){ return t1+t2;}
   static T prodFunc(T const& t1, T const& t2){ return t1*t2;}
   static T negFunc(T const& t) {  return -t;}
   static T idFunc(T const& t) {  return t;}
   static T invFunc(const T& t) {  return CpmRoot::inv<T>(t);}
   static T zeroFunc(const T& t)
      {  return CpmRoot::net<T>(t,0);}
   static T unitFunc(T const& t)
      {  return CpmRoot::net<T>(t,1);}
   static T conjFunc(T const& t){ return CpmRoot::con<T>(t);}
   static T dotFunc(T const& t1, T const& t2)
      { return CpmRoot::con<T>(t1)*t2;}
   static void accFunc(T& z, T const& zAdd){ z+=zAdd;}
};
```

```
// class functions:

template <class T >
T Va<T>::sum(void)const
{
   return Base::vl_.fAcc1(idFunc,accFunc);
}

template <class T>
T Va<T>::operator | (Va<T> const& h)const
{
   return Base::vl_.fAcc2(dotFunc,accFunc,h.vl_);
}

template <class T >
CpmRoot::R Va<T>::frac(T const& t)const
{
   Z n=Base::dim();
   if (n==0) return 0;
   Z sum=0;
   for (Z i=1;i<=n;i++) if ((*this)[i]==t) sum++;
   return ((CpmRoot::R)(sum))/n;
}

template <class T >
CpmRoot::R Va<T>::frac(const V<T>& v)const
{
   Z n=Base::dim();
   if (n==0) return 0;
   Z nv=v.dim();
   if (nv==0) return 0;
   Z i,j,sum=0;
   for (i=1;i<=n;i++){
      T ti=(*this)[i];
      for (j=1;j<=nv;j++){
         if (ti==v[j]) sum++;
      }
   }
   return ((CpmRoot::R)(sum))/n;
}

// main task

template <class T >
Va<T> Va<T>::neg(void)const
{
   Vl<T> vl_Res=Base::vl_.fa(negFunc);
   return Va<T>(V<T>(vl_Res));
}
```

```
template <class T >
Va<T> Va<T>::inv(void)const
{
   Vl<T> vl_Res=Base::vl_.fa(invFunc);
   return Va<T>(V<T>(vl_Res));
}

template <class T >
Va<T> Va<T>::con(void)const
{
   Vl<T> vl_Res=Base::vl_.fa(conjFunc);
   return Va<T>(V<T>(vl_Res));
}

template <class T>
Va<T> Va<T>::net(Z i)const
{
   Vl<T> vl_Res= i==1 ? Base::vl_.fa(unitFunc) : Base::vl_.fa(zeroFunc);
   return Va<T>(V<T>(vl_Res));
}

template <class T >
Va<T>& Va<T>::operator +=(const Va<T>& s)
{
   Base::vl_.ff_(sumFunc,s.c());
   return *this;
}

template <class T >
Va<T>& Va<T>::operator +=(T const& t)
{
   Base::vl_.fc_(sumFunc,t);
   return *this;
}

template <class T >
Va<T>& Va<T>::operator *=(const Va<T>& s)
{
   Base::vl_.ff_(prodFunc,s.c());
   return *this;
}

template <class T >
Va<T>& Va<T>::operator *=(T const& t)
{
   Base::vl_.fc_(prodFunc,t);
   return *this;
}

template <class T>
```

```
bool Va<T>::prnOn(ostream& str)const
{
   return Base::c().prnOn(str);
}

template <class T>
bool Va<T>::scanFrom(istream& str)
{
   return Base::c().scanFrom(str);
}


//////////////////////////// iterated templates ////////////////////////////
// describing multi-indexed quantities
//////////////////////////////// class VVa ////////////////////////////////
// matrices without matrix product
// tensors of rank 2
// We derive VVa<T> from VVo<T> and not from Va<Va<T> >. In the
// latter case Va<T> would have to be the scalar type for VVa<T>
// which is not the case. I tried this 2005-08-19 but came deep
// 'in the woods'. Since thus the arithmetics interface is not
// yet fixed by the definition, we are free to define it anew in
// a more lean way as that of Va. Especially we avoid to
// define friend functions. Also simply taking over the interface
// of Va would not be adequate from the point of view of
// implementation efficiency.
// The implementation of VVa is such that it can be carried over to
// higher numbers of indexes iteratively

template <class T>
class VVa: public VVo<T>{ // version of VVo with arithmetic operations

   typedef VVo<T> Base;
   typedef VVa<T> Type;
   typedef T ScalarType;

public:
   CPM_LIN
      // notice that function con() declared here will be
      // implemented as conjugation of the matrix elements
      // only and not as accompanied by a matrix
      // transposition. The present implementation is natural
      // for an interpretation of VVa's as T-valued areal
      // distributions (as in images or wave functions)
      // Unlike the case for Va<T>, we do not include CPM_IO
      // in the interface of VVa<T>
// constructors
   VVa(Z d1, Z d2):Base(d1,d2){}
   VVa(Z d1, Z d2, T const& t):Base(d1,d2,t){}
      // all components are equal to t
```

```
   VVa(void):Base(){}
   VVa(V< V<T> >  const& x):Base(x){}
   VVa(VV<T> const& x):Base(x){}
   VVa(VVo<T> const& x):Base(x){}
};

template <class T >
VVa<T> VVa<T>::operator+(VVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   VVa<T> res(d,0);
   for (Z i=1;i<=d;i++){
      Va<T> ti=(*this)[i];
      Va<T> si=s[i];
      res[i]=ti+si;
   }
   return res;
}

template <class T >
VVa<T> VVa<T>::operator-(VVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   VVa<T> res(d,0);
   for (Z i=1;i<=d;i++){
      Va<T> ti=(*this)[i];
      Va<T> si=s[i];
      res[i]=ti-si;
   }
   return res;
}

template <class T >
VVa<T> VVa<T>::operator*(T const& r)const
{
   Z d=Base::dim();
   VVa<T> res(d,0);
   for (Z i=1;i<=d;i++){
      Va<T> ti=(*this)[i];
      res[i]=ti*r;
   }
   return res;
}

template <class T >
VVa<T> VVa<T>::operator-(void)const
{
   Z d=Base::dim();
```

```
   VVa<T> res(d,0);
   for (Z i=1;i<=d;i++){
      Va<T> ti=(*this)[i];
      res[i]=-ti;
   }
   return res;
}

template <class T >
VVa<T> VVa<T>::con(void)const
{
   Z d=Base::dim();
   VVa<T> res(d,0);
   for (Z i=1;i<=d;i++){
      Va<T> ti=(*this)[i];
      res[i]=ti.con();
   }
   return res;
}

template <class T >
T VVa<T>::operator|(VVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   T res=T();
   for (Z i=1;i<=d;i++){
      Va<T> ti=(*this)[i];
      Va<T> si=s[i];
      res+=(ti|si);
   }
   return res;
}

//////////////////////// class VVVa ////////////////////////////////////
// tensors of rank 3

template <class T>
class VVVa: public VVVo<T>{// version of VVVo with arithmetic operations

   typedef VVVo<T> Base;
   typedef VVVa<T> Type;
   typedef T ScalarType;

public:
   CPM_LIN
      // notice that function con() declared here will be
      // implemented as conjugation of the components.
      // The present implementation is natural
      // for an interpretation of VVVa's as T-valued spatial
```

```
      // distributions (as in wave functions).
      // Unlike the case for Va<T>, we do not include CPM_IO
      // in the interface of VVVa<T>
// constructors
   VVVa(Z d1,Z d2,Z d3):Base(d1,d2,d3){}
   VVVa(Z d1,Z d2,Z d3,T const& t):Base(d1,d2,d3,t){}
      // all components are equal to t
   VVVa(void):Base(){}
   VVVa( V< V< V<T> > >  const& x):Base(x){}
   VVVa(VVV<T> const& x):Base(x){}
   VVVa(VVVo<T> const& x):Base(x){}
};

template <class T >
VVVa<T> VVVa<T>::operator+(VVVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   VVVa<T> res(d,0,0);
   for (Z i=1;i<=d;i++){
      VVa<T> ti=(*this)[i];
      VVa<T> si=s[i];
      res[i]=ti+si;
   }
   return res;
}

template <class T >
VVVa<T> VVVa<T>::operator-(VVVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   VVVa<T> res(d,0,0);
   for (Z i=1;i<=d;i++){
      VVa<T> ti=(*this)[i];
      VVa<T> si=s[i];
      res[i]=ti-si;
   }
   return res;
}

template <class T >
VVVa<T> VVVa<T>::operator*(T const& r)const
{
   Z d=Base::dim();
   VVVa<T> res(d,0,0);
   for (Z i=1;i<=d;i++){
      VVa<T> ti=(*this)[i];
      res[i]=ti*r;
   }
```

```
   return res;
}

template <class T >
VVVa<T> VVVa<T>::operator-(void)const
{
   Z d=Base::dim();
   VVVa<T> res(d,0,0);
   for (Z i=1;i<=d;i++){
      VVa<T> ti=(*this)[i];
      res[i]=-ti;
   }
   return res;
}

template <class T >
VVVa<T> VVVa<T>::con(void)const
{
   Z d=Base::dim();
   VVVa<T> res(d,0,0);
   for (Z i=1;i<=d;i++){
      VVa<T> ti=(*this)[i];
      res[i]=ti.con();
   }
   return res;
}

template <class T >
T VVVa<T>::operator|(VVVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   T res=T();
   for (Z i=1;i<=d;i++){
      VVa<T> ti=(*this)[i];
      VVa<T> si=s[i];
      res+=(ti|si);
   }
   return res;
}

//////////////////////// class VVVVa ////////////////////////////////////
// tensors of rank 4

template <class T>
class VVVVa: public VVVVo<T>{//version of VVVVo with arithmetic operations

   typedef VVVVo<T> Base;
   typedef VVVVa<T> Type;
   typedef T ScalarType;
```

```
public:
   CPM_LIN
      // notice that function con() declared here will be
      // implemented as conjugation of the components.
      // The present implementation is natural
      // for an interpretation of VVVVa's as T-valued spatial
      // distributions (as in wave functions).
      // Unlike the case for Va<T>, we do not include CPM_IO
      // in the interface of VVVVa<T>
// constructors
   VVVVa(Z d1,Z d2,Z d3,Z d4):Base(d1,d2,d3,d4){}
   VVVVa(Z d1,Z d2,Z d3,Z d4,T const& t):Base(d1,d2,d3,d4,t){}
      // all components are equal to t
   VVVVa(void):Base(){}
   VVVVa( V< V< V< V<T> > > > const& x):Base(x){}
   VVVVa(VVVV<T> const& x):Base(x){}
   VVVVa(VVVVo<T> const& x):Base(x){}
};

template <class T >
VVVVa<T> VVVVa<T>::operator+(VVVVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   VVVVa<T> res(d,0,0,0);
   for (Z i=1;i<=d;i++){
      VVVa<T> ti=(*this)[i];
      VVVa<T> si=s[i];
      res[i]=ti+si;
   }
   return res;
}

template <class T >
VVVVa<T> VVVVa<T>::operator-(VVVVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   VVVVa<T> res(d,0,0,0);
   for (Z i=1;i<=d;i++){
      VVVa<T> ti=(*this)[i];
      VVVa<T> si=s[i];
      res[i]=ti-si;
   }
   return res;
}

template <class T >
VVVVa<T> VVVVa<T>::operator*(T const& r)const
```

```
{
   Z d=Base::dim();
   VVVVa<T> res(d,0,0,0);
   for (Z i=1;i<=d;i++){
      VVVa<T> ti=(*this)[i];
      res[i]=ti*r;
   }
   return res;
}

template <class T >
VVVVa<T> VVVVa<T>::operator-(void)const
{
   Z d=Base::dim();
   VVVVa<T> res(d,0,0,0);
   for (Z i=1;i<=d;i++){
      VVVa<T> ti=(*this)[i];
      res[i]=-ti;
   }
   return res;
}

template <class T >
VVVVa<T> VVVVa<T>::con(void)const
{
   Z d=Base::dim();
   VVVVa<T> res(d,0,0,0);
   for (Z i=1;i<=d;i++){
      VVVa<T> ti=(*this)[i];
      res[i]=ti.con();
   }
   return res;
}

template <class T >
T VVVVa<T>::operator|(VVVVa<T> const& s)const
{
   Z d1=Base::dim(),d2=s.dim();
   Z d=inf<Z>(d1,d2);
   T res=T();
   for (Z i=1;i<=d;i++){
      VVVa<T> ti=(*this)[i];
      VVVa<T> si=s[i];
      res+=(ti|si);
   }
   return res;
}

} // namespace
#endif
```

215 cpmvc.h

```
//? cpmvc.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_VC_H_
#define CPM_VC_H_
/*
    Description: Lean arithmtics for Vl<C>  and Vl< Vl<C> >
*/

#include <cpmvl.h>
#include <cpmc.h>

using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::C;
using CpmRoot::Word;

using CpmArrays::Vl;
using CpmArrays::Outside;
using CpmFunctions::F;

// interfaces as declaration macros

#define CPM_VC_PURE_VECTOR_SPACE_INTERFACE(X,S)\
public:\
   X operator - (void)const;\
   X& operator *=(S);\
   X& operator +=(X const&);\
   X& operator -=(X const&);\
   X operator *(S)const;\
   X operator +(X const&)const;\
   X operator -(X const&)const;

#define CPM_VC_PURE_SCALAR_INTERFACE(X,S)\
public:\
    X& operator *=(S);\
    X operator *(S)const;

#define CPM_VC_SCALAR_PRODUCT_INTERFACE(X,S)\
public:\
   X con(void)const;\
   S operator |(X const&)const;\
   R normSqr(void)const{ return (operator|(*this)).real();}\
   R norm(void)const{ return ::sqrt(normSqr());}\
   R absVal(void)const{ return norm();}\
   R absSqr(void)const{ return normSqr();}
```

```
namespace CpmArrays{

//////////////////////////// class VC ////////////////////////////////////

   class VC{
// vectors with C-valued components, rich interface including arithmetics
      Vl<C> rep_;
         //: representation
      typedef VC Type;
      typedef C ScalarType;
   public:
      CPM_NAM(VC)
      CPM_VC_PURE_VECTOR_SPACE_INTERFACE(Type,C)
      CPM_VC_SCALAR_PRODUCT_INTERFACE(Type,C)
      CPM_VC_PURE_SCALAR_INTERFACE(Type,R)
      CPM_NORMALIZE

      explicit VC(Z d=0): rep_(d){}
      VC(Z d, C z): rep_(d,z){}
      explicit VC(Vl<C> const& v): rep_(v){}
         // construction from data

      Vl<C> rep()const{ return rep_;}
         //: representation

      Z dim()const{ return rep_.dim();}
         //: dimension

      Z b()const{ return rep_.b();}
      Z e()const{ return rep_.e();}

      C const& operator[](Z i)const{ return rep_[i];}
      C& operator[](Z i){ return rep_[i];}

      C const& cui(Z i)const{ return rep_.cui(i);}
      C& cui(Z i){ return rep_.cui(i);}

      C operator()(Z i, Outside mode=DEFAULT)const{ return rep_(i,mode);}
      // Read access defined for all i.
      C const& read(Z i, Outside mode=DEFAULT)const
         { return rep_.read(i,mode);}
      //: read ; Reading components in an efficient (as &'s), safe,
      // and flexible manner.

      VC& makeReal_(R p=2);
      // returns a VC with all components of *this
      // replaced by their absolute value to power p

      VC ranVal(Z j=0)const;
         //: random values
```

```
   };

} // namespace

#endif
```

216 cpmvc.cpp

```
//? cpmvc.cpp
//? Status of work 2008-10-25.
//?
/*
   cpmvc.cpp
*/

#include <cpmvc.h>
#include <cpmvr.h>
using CpmArrays::VC;
using CpmArrays::Vr;

////////////////////////////// class VC ////////////////////////////
namespace{
   void accFunc(C& z, C const& zAdd){ z+=zAdd;}
   C conjFunc(C const& z){ return z.con();}
   C negFunc(C const& z1){  return -z1;}
   C sumFunc(C const& z1, C const& z2){ return z1+z2;}
   C diffFunc(C const& z1, C const& z2){ return z1-z2;}
   C prodFunc(C const& z1, C const& z2){ return z1*z2;}
   C prodFuncConj1(C const& z1, C const& z2){return z1.con()*z2; }
   C prodFuncConj2(C const& z1, C const& z2){ return z1*z2.con();}
   C prodFuncReal(C const& z1, C const& z2){ return z1*z2.real();}
}

C VC::operator |(VC const& x2)const
{
   return rep_.fAcc2(prodFuncConj1,accFunc,x2.rep_);
}

VC VC::con(void)const
{
   return VC(rep_.fa(conjFunc));
}

VC VC::operator - (void)const
{
   return VC(rep_.fa(negFunc));
}

namespace{
   C fpFunc(C const& z, R const& p)
   {
      return p==2 ? C(z.absSqr()) : C(::pow(z.absVal(),p));
   }
}
```

```
VC& VC::makeReal_(R p)
{
   F<C,C> fp=F1<C,R,C>(p)(fpFunc);
   rep_.fa_(fp,rep_.dom());
   return *this;
}

VC& VC::operator +=(VC const& x)
{
   rep_.ff_(sumFunc,x.rep_);
   return *this;
}

VC& VC::operator -=(VC const& x)
{
   rep_.ff_(diffFunc,x.rep_);
   return *this;
}

VC& VC::operator *=(C z)
{
   rep_.fc_(prodFunc,z);
   return *this;
}

VC& VC::operator *=(R r)
{
   rep_.fc_(prodFuncReal,C(r));
   return *this;
}

VC VC::operator +(VC const& x)const
{
  VC res(*this);
  return res+=x;
}

VC VC::operator -(VC const& x)const
{
   VC res(*this);
   return res-=x;
}

VC VC::operator *(C x2)const
{
  VC res(*this);
  return res*=x2;
}

VC VC::operator *(R x2)const
```

```
{
  VC res(*this);
  return res*=x2;
}

VC VC::ranVal(Z j)const
// combining ranVal from Vr<C> with normalization
// exemplifies the interaction of Vl<C> and Vr<C>.
// Actually also the arithmetics could  be taken
// from Vr<C> if a single source of algorithms
// is desired. If so, it would be more natural to
// use Vr<C> directly instead of VC.
{
   Vr<C> temp=Vr<C>(rep_);
   temp=temp.ranVal(j);
   Vl<C> r=temp.c();
   VC res=VC(r);
   res.nor_();
   return res;
}
```

217 cpmvectorgraphics.h

```
//? cpmvectorgraphics.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_VECTORGRAPHICS_H_
#define CPM_VECTORGRAPHICS_H_
/*

Purpose: Declaration of graphic class VecGraph representing
   vector graphics

*/

#include <cpmgraph.h>
#include <cpmdigraph.h>
#include <cpmrfunc.h>

namespace CpmGraphics{

   using namespace CpmStd;

   using CpmRoot::Word;
   using CpmRoot::Z;
   using CpmRootX::Z1;
   using CpmRoot::R;
   using CpmRoot::C;
   using CpmArrays::V;
   using CpmImaging::Color;
   using CpmFunctions::R_Func;

   using CpmGeo::DiGraph;
   using CpmGeo::Angle;


//////////////////// class Figure ////////////////////////////////////////
// defines the analogon of a screen window as a 'paper window'
// on which graphical LATEX commands work.

class Figure{
// the analogon of a screen window as a 'paper window', related to LaTex

// independent data (all belong to a class with default constructor)

   Iv px_;  // picture range in x
   Iv py_;  // picture range in y

   Iv rx_;  // range in x
   Iv ry_;  // range in y
```

```
   R_Func fx_;
   R_Func fy_;
   R_Func gx_;
   R_Func gy_;


   Word nx_; // name of the quantity to which the x-axis refers
   Word ny_; // name of the quantity to which the y-axis refers

   Word tit_; // title of the diagram

   Word cap_; // caption of the figure

   Z1 withOri_;
      // withOrigin!=0 controls autoscaling such that the origin
      // is always on the diagram
   Z1 newGrid_;
      // only =0 or !=0 matters. If it is 0 (should be an acception) one
      // let the grid remain unchanged and draw curves on the frame with
      // a meaning of object space wich is no longer in agreement with
      // the grid and the grid text
   Z1 autoScl_;
      // only =0 or !=0 matters. Influences show(..) such that CL CU are
      // adjusted to the data to be represented prior to drawing them.
      // 'autoscaling'
      // The change of CL and CU remains till new values are assigned
      // by new action
   Word gridText_;
   R2 gridTextPos_;
      // text position is understood in the Frame coordinate system
      // (xF,yF)
      // i.e. (0,0) is left lower corner, (1,1) is right upper corner
public:

   Figure(Iv px, Iv py, Iv rx=Iv(0,1), Iv ry=Iv(0,1));
   void draw(R2 a, R2 b, ostream& str, Z np);
   void drawLine(R2 a, R2 b, ostream& str, Z np);

   R getOx()const{ return px_[1];}
   R getOy()const{ return py_[1];}
   R getLx()const{ return px_[2]-px_[1];}
   R getLy()const{ return py_[2]-py_[1];}
   const Word& cap()const{ return cap_;}
   Word& cap(){ return cap_;}
};

//////////////////////// class VecGraph ////////////////////////////

class VecGraph:public DiGraph<C>{
```

```
// vector graphics, directed graph with vertices of type C
// An instance of this class is a list of points in R2 with some
// of them indicated as connected. The function mark
// creates a graphical representation of the object by drawing
// a line of the color given by attribute 'color' between all
// connected points (here the translation from R2 to Screen points
// is the one provided by the argument Graph gr of function mark.
// As constructed by the constructors of this class, the points
// have no connections at all. One may create such connections
// using the functions set_neg and addEdge of the base class.
// The most powerful way of generating VecGraphs is by
// using a object camera of type CpmCamera2::Camera or
// CpmCamera3::Camera and to use  camera.capt(obj), where obj
// is of type CpmGeo::Skeleton or CpmGeo::WireFrame. By moving
// object camera, one gets all projective images of the original
// so that there is no need for defining geometrical transforms
// of VecGraphs (it is not without meaning that we are here
// in namespace CpmGraphics and not in CpmGeo !)

   typedef DiGraph<C> Base;
   typedef DiGraph<C> CloneBase;
   typedef VecGraph Type;
   Color color;
public:
   virtual DiGraph<C>* clone()const{ return new VecGraph(*this);}

   CPM_IO
      // I/O from Base
   VecGraph(Z n=0):Base(n){}
   explicit VecGraph(const V<C>& vsp, Z connect=1):Base(vsp,connect){}

   Word nameOf()const{ return "VecGraph";}
   void mark(Graph& gr)const;
   void markTriAng(Graph& gr)const;
      // mark a filled triangle, determined by the first 3 components
      // of array x_. No action if there are less than 3 points.
   Color getColor()const{ return color;}
   void setColor(Color c){ color=c;}
   void setColor(Z i){ color=Color(i);}

// 3 functions not yet implemented
   VecGraph shift(C s)const;
   VecGraph rot(C p, Angle rot)const;
   VecGraph scale(R fac)const;

   static VecGraph arc(C p1, C center, Angle rot, Z np=10);
      // circular arc obtained by applying a rotation rot arround
      // center in n steps on point p1.

   static VecGraph line(C pi, C pf, Z np=10);
```

```
      // straight line starting at pi and ending at pf
      // represented as a chain of n equidistant connected
      // points. (n-1 edges); cpmassert for n>1.

   static VecGraph circle(C pc, R r, Z np=64);
      // circle arround pc with radius r

   static VecGraph ellipse(C pc, R a1, R a2, Z np=64);
      // ellipse with center pc, semi-axis a1 in 1-direction
      // and a2 in 2-direction

   static VecGraph parabola(C pi, C pc, C pf, Z np=16);
      // parabola with initial point pi, final point pf, and
      // Bezier control point pc. The intermediate points
      // correspont to equidistant positions in time, where the time span
      // needed for the whole path is 1. The meaning of the control
      // point is to express the initial velocity as (pc-pi)/(1/2)
      // and the final velocity as (pf-pc)/(1/2).
      // cpmassert for n>2.

   static VecGraph bezier(V<C> const& p, Z n=256);
      // Bezier curve connecting p.first() with p.last() represented as a
      // polygon of n points
};

//////////////////////// class VecGraphs ////////////////////////////

class VecGraphs{ // array of VecGraph objects
   // An array of VecGraph objects is considered a new
   // unity; notice that it would be probably quite inefficient to
   // convert such an array into a single VecGraph by
   // forming the obvious set union.
   CpmArrays::V<VecGraph> rep;

   typedef VecGraphs CloneBase;
   typedef VecGraphs Type;
public:
   CPM_IO
   virtual VecGraphs* clone()const{ return new VecGraphs(*this);}

explicit VecGraphs(const V<VecGraph>& vsk):rep(vsk){}
   VecGraphs():rep(){}

// combining VecGraphss
   VecGraphs& operator &=(const VecGraphs& wf)
      { rep&=wf.rep; return *this;}

   VecGraphs& operator &=(const VecGraph& sk)
      { rep&=sk; return *this;}
```

```
// access
   const VecGraph& operator[](Z i)const{ return rep[i];}
   VecGraph& operator[](Z i){ return rep[i];}

   Z dim()const{ return rep.dim();}

   Word nameOf()const{ return "VecGraphs";}

   void mark(Graph& gr)const
   { for (Z i=1;i<=dim();i++) rep[i].mark(gr);}

   void epic(ostream& str, Figure& fg);
};

}  // namespace

#endif
```

218 cpmvectorgraphics.cpp

```
//? cpmvectorgraphics.cpp
//? Status of work 2008-10-25.
//?
#include <cpmvectorgraphics.h>
#define CPM_DIMDEF2
#include <cpmdim.h>

using namespace CpmGraphics;
using namespace CpmRoot;
using namespace CpmDim2;

Figure::Figure(Iv px, Iv py, Iv rx, Iv ry):px_(px),py_(py),rx_(rx),
   ry_(ry)
{
   fx_=R_Func(rx_,px_);
   fy_=R_Func(ry_,py_);
}

void Figure::drawLine(R2 a, R2 b, ostream& str, Z np)
{
   R2 a2(fx_(a[1]),fy_(a[2]));
   R2 b2(fx_(b[1]),fy_(b[2]));
   str<<endl<<"   "<<"\\dottedline{"<<np<<"}("<<a2[1]<<","<<a2[2]<<")\
   ("<<b2[1]<<","<<b2[2]<<")";
   // actually we need only to draw a line; the epic package available to
   // me produces no actaptable quality
}

void Figure::draw(R2 a, R2 b, ostream& str, Z np)
{
   R2 a2(fx_(a[1]),fy_(a[2]));
   R2 b2(fx_(b[1]),fy_(b[2]));
   R2 c=(a2+b2)*0.5;
   str<<endl<<"   "<<"\\qbezier["<<np<<"]("<<a2[1]<<","<<a2[2]<<")("
      <<c[1]<<","<<c[2]<<")("<<b2[1]<<","<<b2[2]<<")";
   // actually we need only to draw a line; the epic package available to
   // me produces no actaptable quality
}

void VecGraph::mark(Graph& gr)const
{
   using CpmArrays::S;
   using CpmLinAlg::Z2;
   S<Z2> edg=edges();
   for (Z i=1;i<=edg.card();i++){
      Z2 zi=edg[i];
      R2 a(x_[zi[1]]); // x is an protected array of C's
```

```
         // in base class DiGraph<C>
      R2 b(x_[zi[2]]);
      gr.draw(a,b,color);
         // thus Graph::draw is the basic 'machine'
         // for drawing polygons (simulating curved
         // lines) in my graphics system
   }
}

void VecGraph::markTriAng(Graph& gr)const
{
   if (dim()<3) return;
   gr.fillTriangle(R2(x_[1]),R2(x_[2]),R2(x_[3]),color);
}

bool VecGraph::prnOn(ostream& str)const
{
   cpmwt("VecGraph");
   if (!Base::prnOn(str)) return false;
   cpmp(color);
   return true;
}

bool VecGraph::scanFrom(istream& str)
{
   if (!Base::scanFrom(str)) return false;
   cpms(color);
   return true;
}

VecGraph VecGraph::arc(C p1, C center, Angle rot, Z np)
{
   return Skeleton::arc(Spc(p1),Spc(center),AxVec(rot),np).flatten(1);
}

VecGraph VecGraph::line(C pi, C pf, Z np)
{
   return Skeleton::line(Spc(pi),Spc(pf),np).flatten(1);
}

VecGraph VecGraph::circle(C pc, R r, Z np)
{
   return Skeleton::circle(Spc(pc),r,np).flatten(2);
}

VecGraph VecGraph::ellipse(C pc, R a, R b, Z np)
{
   return Skeleton::ellipse(Spc(pc),a,b,np).flatten(2);
}
```

```
VecGraph VecGraph::parabola(C pi, C pc, C pf, Z np)
{
   return Skeleton::parabola(Spc(pi),Spc(pc),Spc(pf),np).flatten(1);
}

VecGraph VecGraph::bezier(V<C> const& p, Z n)
{
   Z pd=p.dim();
   V<Spc> ps(pd);
   for (Z i=1;i<=pd;++i) ps[i]=Spc(p[i]);
   return Skeleton::bezier(ps,n).flatten(1);
}

void VecGraphs::epic(ostream& str, Figure& fg)
{
   using CpmArrays::S;
   using CpmLinAlg::Z2;
   R ox=fg.getOx();
   R oy=fg.getOy();
   R lx=fg.getLx();
   R ly=fg.getLy();
   str.precision(6);
   str.setf(ios_base::fixed,ios_base::floatfield);
   str<<endl<<"% Output of function VecGraphs::epic(...)";
   str<<endl<<"\\setlength{\\unitlength}{1mm}";
   str<<endl<<"\\begin{figure}";
   str<<endl<<"\\begin{picture}("<<lx<<","<<ly<<")("<<ox<<","<<oy<<")";
   for (Z i=1;i<=dim();i++){
      VecGraph vgi=rep[i];
      S<Z2> edg=vgi.edges();
      for (Z j=1;j<=edg.card();j++){
         Z2 zj=edg[j];
         Z zj1=zj[1];
         Z zj2=zj[2];
         R2 a(vgi[zj1]);
         R2 b(vgi[zj2]);
         fg.drawLine(a,b,str,1);
      }
   }
   str<<endl<<"\\end{picture}";
   str<<endl<<"\\caption{"<<fg.cap()<<"}";
   str<<endl<<"\\end{figure}";
}

#undef CPM_DIMDEF2
```

219 cpmvectormore.h

```
//? cpmvectormore.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_VECTOR_MORE_H_
#define CPM_VECTOR_MORE_H_
/*
       additional topics for R_Vector (and also to R_Matrix)
*/

#include <cpmrmatrix.h>
#include <cpmalgorithms.h>

namespace CpmArrays{
   using CpmLinAlg::Z2;
   using CpmLinAlg::R2;

R evalPoly(R const& x, R_Vector const&  c);
   // evaluates the real polynomial function determined by data c
   // at position x. This polynomial function is:
   //    p(x)=c[3]+z*c[4]+z*z*c[5]+z*z*z*c[6]+ ...,
   //    where
   //       z:=(x-x0)*rInv, x0:=c[1], rInv:=c[2].
   // This is the same format in which the next fit function
   // returns its result. Notice that writing the polynomial function
   // simply as a polynomial in x leads to coefficients varying
   // over many order of magnitude with the index position in cases where
   // the value of x is large in the whole range under consideration.
   // Especially the determination of these coefficients by any fit
   // becomes numerically unstable.
   // Actually, we deal here with a collective description of a type of
   // functions, which calls for becoming transformed into a class!
   // Later ??

R_Vector polyFit(R_Vector const& x, R_Vector const& y,
   Z deg, R x0=0, R r=1, bool autoAdjust=true );
   // Least mean square fit of a polynomial function of degree deg
   // to data (x[i],y[i]).
   // Let res be the return value, then
   // t |--> evalPoly(t,res) is the function that fits the data.
   // The first 2 components res[1], res[2] of res are either equal
   // to x0 and 1/r (for autoAjust==false), or they are
   // (x[1]+x[n])/2 and absVal(2/(x[n]-x[1])) (for autoAjust==true).
   // Let res be the return value. Then res.dim()=deg+3 and
   // the polynomial function x |--> res[3]+z*res[4]+z*z+res[5]+...,
   //    z=(x-x0)/r
   // is the least mean square fit to the list (x[i],y[i]).
   // Elegant application of the pseudo-inverse. Let us explain it in
```

```
   // the general case of linear fit with respect to an arbitrary system
   // of functions (not just powers of x as in the present case). Let
   // f[i], i=1,...,d be the system of functions. We define the matrix
   // a[i][j]:=f[i](x[j]). We ask for coefficients c[i],i=1,...,d such
   // that y[j]-Sum(i=1,d) c[i]*f[i](x[j])=y[j]-(~a*c)[j]
   // becomes as small as possible in the mean over j. Thus we search the
   // vector c of coefficients such that y '=' ~a * c. The best solution
   // to this is c = !~a * y , where !a is the pseudo-inverse of matrix
   // a.
   // For the case of f[i](x):=((x-x0)/r)^(i-1) we can obtain the matrix
   // a by the elegant algorithm (see implementation code).

R_Vector linFit(const V<F<R,R> >& f, R_Vector const& x,
   R_Vector const& y);
   // Least mean square fit by a linear combination
   // c[1]*f[1]+...+c[n]*f[n], n=f.dim()
   // of a given family f of functions to data (x[i],y[i]).
   // The array c is the return value.

R_Vector nonLinFit(const F<X2<R,R_Vector>,R>& f,
   R_Vector const& x, R_Vector const& y, R_Vector const&  pStart,
   R fracTol=1e-3, bool verbose=false);
   // Least mean square fit by a function x|--> f(X2(x,p)) to data
   // (x[i],y[i]). Here, R_Vector p is determined from minimizing misfit.
   // p is the return value.
   // The minimizing method needs a starting value for p and a tolerance
   // limit on input, these things are provided with the last two
   // arguments. Notice that the dimension of p equals that of pStart
   // by definition.
   // If verbose=true, some diagnostic information is written on cpmcerr.

R polyInte_4(R_Vector const& x, R_Vector const& y, R x0, R& error);
   // Polynomial interpolation.
   // Name appendix _4 says that the fourth argument is a non-constant
   // reference.
   // Modified from polint() of Press et al.
   // x is a list of x-values, all are assumed to be different (i.e. list
   // is injective). y is a list of at least equal length of
   // corresponding y-values. Returned is the value of the interpolating
   // polynom at/ x-position x0. After function call, error is an error
   // estimate for the function value as indicated in Press et al.

R polyInte(R_Vector const& x, R_Vector const& y, R x0);
   // Version of the last function which has more natural argument list
   // by not indicating an error estimate (which is not too attractive an
   // concept here anyway).

R ratiInte_4(R_Vector const& x, R_Vector const& y, R x0, R& error);
   // rational interpolation
   // Name appendix _4 says that the fourth argument is a non-constant
```

```
   // reference.
   // 'Diagonal rational interpolation' according to Bulirsch and Stoer.
   // Modified from function ratint() of Press et al.
   // Same meaning of arguments as in polyInte(). Creates a warning and
   // returns a large value if x0 is located at a pole of the
   // interpolating function.

R ratiInte(R_Vector const& x, R_Vector const& y, R x0);
   // As previous function but no error indication.

Z  locate(R_Vector const& x, R x_);
// Let x be ascending:  x[1] < x[2] <...< x[n]. For a given
// real number x_, we find the largest integer j such that x[j] <= x_.
// This j is the return value of the function. If all x[...] are larger
// than x_, we return 0. Algorithm is most effective for equidistant or
// nearly equidistant lists.

R locatePeak(R_Vector const& x, R p=2.);
// The values |x[i]|^p are considered defining a probability
// distribution over [1,....n] (n=x.dim()) and the expectation value
// of the 'position i' is calculated. If all x[i]=0, no distribution
// is defined and we return 0 (from definition all regular return
// values are >= 1 and <= n)

R2 peakAndSigma(R_Vector const& x, R p=2.);
// Just as previous function but returns the standard deviation
// (half-width of the peak) as the second component of the result

// The following functions smoothen() ... interpolateInverse() interprete
// R_Vector& x as a list of function values belonging to a argument
// values 1,....x.dim().

R_Vector smoothen(R_Vector const& x, bool periodic=false);
// fast convolution with binomial kernel (1 2 1)/4
// end points treated as if data would be extended as constant
// unless argument periodic equals true

R_Vector smoothen(R_Vector const& x, Z p, bool periodic=false);
// applying previous function p times, p>=1
// Returns x for p<=0

X2< V<Z>, V<Z> >  locExt(R_Vector const& x);
//: local extrema
// Let res be the return value of the function. Then
// res.c1() is a list of local minima, and res.c2() a list
// of local maxima. Take, for instance, the first:
// V<Z> a=res.c1();
// Then there are a.dim() local minima of x at indexes
// i=a[1], a[2], ...
// and the x values there are, of course
```

```
// x[a[1]], x[a[2]], ...
// In order to obtain only robust extremal values, not created
// by numerical noise, it may be advisable to apply function
// smoothen(...) to x before handing x over to the present function.
// Extrema here are strict: i rightfully appears in the minimum
// list res.c1() iff
//    i>=2 and i<x.dim() and x[i-1]>x[i] and x[i+1]>x[i]
// and in the maximum list res.c2() iff
//    i>=2 and i<x.dim() and x[i-1]<x[i] and x[i+1]<x[i]
// Thus the lists res.c1() and res.c2() are void for all monotone
// lists x, not only for strictly monotone ones.

R_Vector derive(R_Vector const& x);
// foreward derivation, function assumed to be constant outside
// defined area

R_Vector integrate(R x1, R_Vector const& x);
// Integrate a function from left to right. Start value for result
// at left end given as x1.

R_Vector accumulate(R_Vector const& x, bool average=false);
// res[i]=x[1]+x[2]+...+x[i] for i>2 and mutandis mutatis for
// smaller i and smaller dimension
// Needed e.g. for cumnulative histograms.
// notice derive(accumulate(x))=x
// If the newly introduced second argument is 'true', the
// defining formula is replaced by
// res[i]=(x[1]+x[2]+...+x[i])/i so that a limit may
// exist x's which represents a mildly varying function.

R extMean(R_Vector const& x);
//: extrapolated mean
// Estimating 'the DC component of x' without making use of FFT.
// Estimates a limit of the sequence
// x[1], (x[1]+x[2])/2, (x[1]+x[2]+x[3])/3, ...
// by determining the local maxima and the local minima of this sequence,
// and then returning the mean of the last minimum and the last maximum.
// To make local extrema independent of numerical noise, it may be
// advisable to apply this function to a smoothened version of the
// original data x.

void makePositive(R_Vector& x, R tiny=1.e-3);
// set to zero where function is negative
// more precisely set equal to tiny > 0 where the function is <tiny.
// This is done to obtain stricly monotone (even only slightly
// increasing) functions in monotonize

R_Vector reverse(R_Vector const& x);
// 'interchange left and right'
```

```
R_Vector monotonize(R_Vector const& x, Z pSmooth=3);
// Change empiric data which should represent a monotone increasing
// function into a slightly modified data set which is strictly
// increasing.

R interpolate(R ic, R_Vector const& x);
// ic 'continuous index' for i in Z we should get back x[i]
// linear interpolation and constant extrapolation

R interpolate(R xp, R_Vector const& x, R_Vector const& y);
// Interpolation from non-equidistant lists:
// x is a list of x-values and y a list of y-values, xp is the value
// for which an interpolated value is needed.
// linear interpolation and constant extrapolation

R interpolateInverse(R xc, R_Vector const& x);
// we try to find a R ic such that xc=interpolate(ic,x).
// We assume that the list x is strict monotonely ascending.
// This is achieved by getting x from function monotonize.

// The following functions moments() and statistics() interpret
// R_Vector& x as a list of empirical data resulting from a stochastic
// experiment(or process).

R_Vector moments(R_Vector const&  data);
// modified from Press et al. moment().
/*
Given a list of R data (forming a R_Vector) we get a result list res
which gives the most important statistical descriptors for this data set.
These are the  descriptors:

     res[1] average of data (mean value)
     res[2] average deviation (estimating the uncertainty of the mean)
     res[3] standard deviation
     res[4] variance (square of standard deviation)
     res[5] skewness
     res[6] kurtosis
   See function statistics() for definition of skewness and kurtosis.
   Notice that in the present function the conventional (not normalized)
   values of skewness and curtosis  are given just as defined in Press et
   al. as Skew() and Kurt().
*/

R_Vector statistics(R_Vector const&  dat);
/*
Given a list of R data (forming a R_Vector) we get a result list res
which gives the most important statistical descriptors for this data set.
These are the  descriptors:

     res[1] average of data (mean value)
```

```
     res[2] average deviation (estimating the uncertainty of the mean)
     res[3] standard deviation
     res[4] variance (square of standard deviation)
     res[5] skewness divided by sqrt(6/n)
     res[6] kurtosis divided by sqrt(24/n)

     res[7] confidence level for the hypothesis that data follow a
      normal distribution (with mean and standard deviation from above).
      This is found from the Kolmogorov-Smirnov test which is considered
      reliable for n>=20.

     res[8] minimum of data
     res[9] median of data
     res[10] maximum of data

Skewness and kurtosis are zero for normal distributions. The
normalization denominators in res[5] and res[6] are the standard
deviation of the skewness and kurtosis of a normal distribution, so that
values greater, say, 2 indicate significant deviation from normality.
Negative skewness indicates that the distribution is broader for small
values than for large ones. Negative kurtosis indicates a 'Tafelberg' and
positive kurtosis a 'Matterhorn'. See Press, Flannery et al. Chapter 13.1
p. 474. for moments and p.490-94 for Kolmogorov-Smirnov test.
*/

R_Vector bimodalAnalysis(R_Vector const&  dat);
// This is still an ad hoc device working well for a set of data
// which strongly cluster arround two different values. The actual
// application case helps to understand the situation. We sample the
// electrical potential along a surface which consists of uniformely
// charged areas with two different values of the areal charge density.
// Due to the presence of field-creating particles the potentials
// are not stricly constant on the sub-areas with constant charge
// density. We want to get the mean values over these subareas without
// knowing to what a position and hence to what a charge density
// the samled potential values (=dat) belong.
// What is actually done is to divide dat into "upper values"
// being closer to dat.sup() and lower values being closer to dat.inf()
// and returning res=R_Vector(2), res[1]=lower_values.mean(),
// res[2]=upper_values.mean().
// Presently no test is implemented whether dat is actually bimodal
// or not.

R_Vector gradient( const R_Matrix& x, R_Vector const& y);
// Gradient vector from sampled values of a scalar function.
// Consider a function f such that y[i]=f(x[i]) i=1,...n
// Then we determine the gradient vector gr such that
// f(x[i])=f(x[1])+(gr|x[i]-x[1]).  i=2,....n, (.|.) scalar product
// Let d the dimension of the x[i], then gr has d components. In order to
// have enough equations one has to have n>=d+1. For the method of
```

```
// computation to be used here, more equations than unknowns poses no
// problem. gr is the gradient vector of the function f at point x[1].

R_Matrix gradient( const R_Matrix& x, const R_Matrix& y);
// Gradient matrix from sampled values of a vector-valued function.
// Consider an the situation as above but f vector-valued
// f(x)=(f1,....fp)
// Then we determine the gradient matrix gr such that
// f(x[i])[k]=f(x[1])[k]+ (x[i]-x[1]|gr[k]).  i=2,....n, k=1...p
// Then the same solution as above. It is worthwhile to have this
// serialized version of the function , since each application of the
// previous function calculates the pseudo-inverse of a matrix which is
// determined by the first argument alone, and which has to be calculated
// only once in the present version

V< X2<Z,R> > levelCrossing(R_Vector const& y, R frac,
   bool autoLevel=true, R levelSpezifier=0.99);
// We consider y as defining a function f :[1,n]-->R the graph of
// which is the polygon (1,y[1]),(2,y[2]), ... (n,y[n]), n=dy.dim().
// Consider
// V<X2<Z,R> > res=levelCrossing(y,frac,autoLevel,levelSpezifier);
// Z m=res.dim();
// Then for each j \in {1,...,m} we have f(res[j].c2())==threshold
// and signum(f'(res[j].c2())) = res[j].c1().
// Here threshold=levelSpezifier*frac for autoLevel==false
// and  threshold=yVraiMax*frac for autoLevel==true
// where yVraiMax is the real number below which the fraction
// fr of all y-components lie. The fraction fr is given by the
// argument levelSpezifier (which thus has quite a different meaning
// for the two cases autoLevel==true and autoLevel==false

Iv pairedLevelCrossing(R_Vector const& y, R frac,
   bool autoLevel=true, R levelSpezifier=0.99);
// returns the position first and the last of a matching pair
// level crossings (ascending crossing matches descending crossing and
// vice versa). Thus, if y resemples a profile through the alps
// from north to the south, we get the positions of the first
// point in the north where height reaches (say) 2000 m and
// the most southern point where we leave the niveau 2000 m.

//////////////////////////////// geometry ////////////////////////////////

void rotation(R_Vector& x, R_Vector const&  a, R_Vector const&  b);
// If neither a nor b is a zero vector, there is a uniquely defined
// minimal rotation which carries a into b. In calling rotation,
// this rotation is applied to vector x, which after the function call
// holds the value of x after rotation.
// See UM/Elmo 94-10-10/3-5 for a derivation of the algorithm.
// To be used as a tool in implementing function simplex
```

```
void simplex(R_Matrix& s, Z d);
// The statements
//
//   R_Matrix s;
//   simplex(s,3);
//
// will turn s into a 4*3 matrix such that s[1],s[2],s[3],s[4] are the
// corners of a regular tetrahedron. The center of gravity of the
// tetrahedron is the zero vector (origin) and the distance of every
// corner from the origin is 1. The position of the corners with respect
// to the axes is not the one which one would normally choose in
// drawing a regular tetrahedron. The present choice comes from a
// construction which is valid in every dimension (the analogue to
// regular tetrahedron then is called a regular simplex) and tries to
// make all corners as close to coordinate axes as possible.
// See UM/Elmo 94-10-10/1-2

//////////////////////////////// miscellaneous ////////////////////////

Z2 linearToMatrix(Z k, Z m, Z n);
// Consider m*n positions that are spatially layed out like the matrix
// elements of a (m,n)-Matrix. Let k be in {1,2,...m*n}. Then the
// return value is the position (i,j) in matrix indexing notation ([i][j]
// in my C++ matrices, indexing starting with 1) of a position that by
// the normal linearization of a matrix (appending the rows (not
// columns!) starting with the first one. Notice that this is the normal
// order in reading letters from a page. Beyond the primary domain for k,
// the function is extended periodically so that no restrictions for k
// have to be obeyed. m>0 and n>0 are required.

Z matrixToLinear(Z i, Z j, Z m, Z n);
// finding the linear position (see previous function) of position (i,j)
// of a (m,n) matrix. Again, i,j are shifted into the proper range and
// m>0 and n>0 are required.

R_Vector binomialCoefficients(Z n);
// returns (n over 0), (n over 1), .... (n over n) as an R_vector
// of dimension n+1. Notice that computing these coefficiens all at
// once maskes much sense since they normally are used together and
// much synergy is available then.


////////////////////////// function tenToMat ////////////////////////////

template <class T>
Vl< Vl<T> > tenToMat(Vl<Vl<Vl<T> > > const& x)
//: tensor to matrix
// transforms a 3-tensor into a matrix. The type of tensor elements
// is handled as a template parameter. Useful for graphical
// representation.
```

```
{
   Word loc("CpmArrays::tenToMat(Vl<Vl<Vl<T>>>)");
   Z d1=x.dim();
   cpmassert(d1>0,loc);
   Z d2=x[0].dim();
   cpmassert(d2>0,loc);
   Z d3=x[0][0].dim();
   cpmassert(d3>0,loc);
   Z2 f=CpmAlgorithms::genSqrRoot(d1);
   Z f1=f[1], f2=f[2];
   Z m=d2*f1, n=d3*f2;
   Vl< Vl<C> > res(m,Vl<C>(n));
   for (Z h=0;h<d1;++h){
      Z2 h2=linearToMatrix(h+1,f1,f2);
      Z si=h2[1]-1,sj=h2[2]-1;
         // the +1 and -1 operations take into account that
         // here indexing starts with 0
      si*=d2; sj*=d3;
      for (Z i=0;i<d2;++i){
         for (Z j=0;j<d3;++j){
            res[i+si][j+sj]=x[h][i][j];
         }
      }
   }
   return res;
}

////////// functions for handling data series ////////////////////////////
// Modelling computations turned out to generate data in many
// cases most naturally as instances of V< V<R2> > i.e.
// as lists of data series, where a data series is a list of
// R2's which typically are pairs (t,y) where t is a point in model
// time to which the computed value y belongs. Notice that
// CpmGraphics::Graph(V< V<R2> > const&, ....) is a flexible function to
// show such data on screen.

V< V<R2> > homDataRan(V< V<R2> > const& x);
//: homogenize data range
// x.dim() has to be even = 2*dh
// x[i] and x[i+dh] contain data of comparable origin.
// Typically they may represent data of various quantities computed
// by two different methods. In the R2-typed data, the first component
// is interpreted as time to which the value of the second component
// refers.
// Since the values of these quantities may vary by
// a large factor, we multiply x[i] and x[i+dh] by a common
// (i-dependent) positive factor which forces these values
// into the interval [-1,1] and let them reach one of the
// boundaries at least once.
// The data series x[i] and x[i+dh] are treated symmetrically, so that
```

```
// absolute values > 1 can't occur in the processed data.
// Corresponding data need not to fit in length or in the points
// in time to which they refer.

V< V<R2> > difOfCorData(V< V<R2> > const& xa, bool homFirst=true);
//: difference of corresponding data
// Same data structure as for the previous function assumed.
// After adjusting the data ranges via the previous function
// (remember: corresponding data are multiplied by a common
// positive factor) we form the difference between corresponding data.
// Since this makes sense only if corresponding data belong to the same
// point in time, appropriate tests are built in.
// If the last argument is set to false, the prepararoty
// call to homDataRan is avoided.

bool toGnuPlot(Word const& filename, V< V<R2> > const& x);
//: to gnu-plot
// Creates an outputfile with name 'fileName' and writes into it
// the elements of x in a manner that the file is suitable
// as a data input file for making gnuplot graphics with them.
// There is a similar function bool R_Matrix::toGnuPlot(Word const&)const
// for filing data held by a R_Matrix.

bool toGnuPlot(ostream& str, V< V<R2> > const& x);
//: to gnu-plot
// Similar to the previous function, but writes on an ostream, that may
// have been created before by opening a file.

} // namespace

#endif
```

220 cpmvectormore.cpp

```
//? cpmvectormore.cpp
//? Status of work 2008-10-25.
//?
#include <cpmvectormore.h>
#include <cpmmini.h>
#include <cpmalgorithms.h>
#include <cpmfile.h>

using namespace CpmRoot;
using namespace CpmSystem;
using namespace CpmArrays;
using CpmAlgorithms::Percentilizer;
using CpmGeo::Iv;
using std::vector;

R_Vector CpmArrays::polyFit(R_Vector const& x, R_Vector const& y,
   Z deg, R x0, R r, bool autoAdjust)
{
   using CpmGeo::Iv;
   using CpmRoot::inv;
     Z n=x.dim(),i;
   if (y.dim()<n)
      cpmerror(
      "polyFit(R_Vector const&, R_Vector const&, Z): dimension mismatch")\
      ;
   if (autoAdjust){
      Iv iv(x[1],x[n]);
      x0=iv.mean();
      r=iv.radius();
   }
   R rInv=inv(r);
   R_Vector xc(n);
   for (i=1;i<=n;i++) xc[i]=(x[i]-x0)*rInv;
   Z d=deg+1;
   R_Matrix a(d,n);
   a[1]=R_Vector(n,1.);
   for ( i=2;i<=d;i++) a[i]=a[i-1]*xc;
   R_Vector res(2);
   res[1]=x0;
   res[2]=rInv;
   res=res&(!~a*y);
   return res;
}


R CpmArrays::evalPoly(const R& x, R_Vector const& a)
{
```

```
   Z n=a.dim();
   R x0=a[1];
   R rInv=a[2];
   R y=a[3];
   R xi=(x-x0)*rInv;
   R xip=xi;
      // power of xi
   for (Z i=4;i<=n;i++){
      y+=a[i]*xip;
      xip*=xi;
   }
   return y;
}

R_Vector CpmArrays::linFit(const V< F<R,R> >& f, R_Vector const& x,
   const R_Vector& y)
{
     Z n=x.dim();
   if (y.dim()<n)
      cpmerror("linFit(const V<F<R,R> >& ,...): dimension mismatch");
   Z m=f.dim(),i,j;
   R_Matrix a(m,n);
   for (i=1;i<=m;i++){
      for (j=1;j<=n;j++){
         a[i][j]=f[i](x[j]);
      }
   }
   return (!~a*y);
}

namespace{

   R auxFunc(R_Vector const& p, const F<X2<R,R_Vector>,R>& f,
      R_Vector const& xList, R_Vector const& yList)
   {
      R sum=0;
      Z n=xList.dim(),i;
      for (i=1;i<=n;i++){
         R val=f(X2<R,R_Vector>(xList[i],p))-yList[i];
         sum+=(val*val);
      }
      return sum;
   }
}

R_Vector CpmArrays::nonLinFit(const F<X2<R,R_Vector>,R>& f,
   R_Vector const& x, R_Vector const& y, R_Vector const& pStart,
   R fracTol, bool verbose)
{
   using CpmFunctions::F3;
```

```
   using CpmAlgorithms::Mini;
     Z n=x.dim();
   if (y.dim()<n)
      cpmerror("nonLinFit(...): dimension mismatch");

   Mini mini(F3<R_Vector, F<X2<R,R_Vector>,R>,
      R_Vector, R_Vector,R>(f,x,y)(auxFunc),fracTol);
   mini.verbose=(verbose ? 1 : 4);
   return mini.getPoint(pStart);
}

// Interpolation functions

R CpmArrays::polyInte_4(R_Vector const& xa, R_Vector const& ya, R x,
   R& intError)
{
   Z n=xa.dim();
   cpmassert(n<=ya.dim(),"CpmArrays::polyInte(...)");
   Z i,m,ns=1;
   R den,dif,dift,ho,hp,w;
   dif=cpmabs(x-xa[1]);
   R_Vector c(n);
   R_Vector d(n);
   R y,dy=0;
   for (i=1;i<=n;i++) {
      if ( (dift=cpmabs(x-xa[i])) < dif) {
         ns=i;
         dif=dift;
      }
      c[i]=ya[i];
      d[i]=ya[i];
   }
   y=ya[ns--];
   for (m=1;m<n;m++) {
      for (i=1;i<=n-m;i++) {
         ho=xa[i]-x;
         hp=xa[i+m]-x;
         w=c[i+1]-d[i];
         if ( (den=ho-hp) == 0.0){
            cpmdebug(xa);
            cpmerror(
               "Error in function polyInt(): two x-values coincide");
         }
         den=w/den;
         d[i]=hp*den;
         c[i]=ho*den;
      }
      y += (dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
   }
   intError=dy;
```

```
   return y;
}

R CpmArrays::polyInte(R_Vector const& xa, R_Vector const& ya, R x)
{
   R intError=1;
   return polyInte_4(xa,ya,x,intError);
}

R CpmArrays::ratiInte_4(R_Vector const& xa, R_Vector const& ya, R x,
   R& intError)
{
   const R hugeUM=1e30;
   const R tinyUM=1e-30;
   Z n=xa.dim();
   Word loc("ratiInte(...)");
   cpmassert(n<=ya.dim(),loc);
   Z m,i,ns=1;
   R w,t,hh,h,dd;
   R_Vector c(n);
   R_Vector d(n);
   hh=cpmabs(x-xa[1]);
   R y,dy=0;
   for (i=1;i<=n;i++) {
      h=cpmabs(x-xa[i]);
      if (h == 0.0) {
         y=ya[i];
         dy=0.0;
            intError=dy;
         return y;
      } else if (h < hh) {
         ns=i;
         hh=h;
      }
      c[i]=ya[i];
      d[i]=ya[i]+tinyUM;
   }
   y=ya[ns--];
   for (m=1;m<n;m++) {
      for (i=1;i<=n-m;i++) {
         w=c[i+1]-d[i];
         h=xa[i+m]-x;
         cpmassert(h!=0.,loc);
         t=(xa[i]-x)*d[i]/h;
         dd=t-c[i+1];
         if (dd == 0.){
            cpmwarning("ratiInte(...): pole found at argument position");
            y=hugeUM;
            dy=hugeUM;
            intError=dy;
```

```
            return y;
         }
         dd=w/dd;
         d[i]=c[i+1]*dd;
         c[i]=t*dd;
      }
      y += (dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
   }
   intError=dy;
   return y;
}

R CpmArrays::ratiInte(R_Vector const& xa, R_Vector const& ya, R x)
{
   R intError=1;
   return ratiInte_4(xa,ya,x,intError);
}

Z  CpmArrays::locate(R_Vector const& x, R x_)
{
   Z n=x.dim();
   R xb=x[1];
   R xe=x[n]; // for shortness
   cpmassert(xb<xe,"CpmArrays::locate(R_Vector,R)");

   if (x_>=xe) return n;
   if (x_<xb)   return 0;

   Z k=(Z)(1.+(n-1.)*(x_-xb)/(xe-xb));
      // varies from 1 to n for x_ from x[1] to x[n];
      // k increases by one if x_ increases by (xe-xb)/(n-1)
   Z j=k;

   if ( x_==x[k] ){    // for equidistant lists this is the case
      ;
   }
   else if ( x_ > x[k] ){         // then there is a chance that
                           // j can be increased
      while ( x_ >= x[j+1] ) j++;
   }                        // x_>=x[j] has to hold and j has to be
                           // the largest value for this to hold
   else { // then x_ < x[j]      thus j has to be decreased
      while ( x_ < x[j] ) j--;        // Since x_ has to be >= x[j],
   }
   return j;
}

namespace{
   void powFast(R& w, R p)
      // changes w into w^p
```

```
   {
      if (p==1.);
      else if (p==2.) w*=w;
      else if(p==3.) w=w*w*w;
      else w=pow(w,p);
   }
}

R CpmArrays::locatePeak(R_Vector const& x, R p)
{
   R a=0, b=0, w;
   Z n=x.dim();
   for (Z i=1;i<=n;i++){
      w=cpmabs(x[i]);
      powFast(w,p);
      a+=w*i;
      b+=w;
   }
   if (b==0.) return 0; else return a/b;
}


R2 CpmArrays::peakAndSigma(R_Vector const& x, R p)
{
   Z mL=4;
   static Word loc("CpmArrays::peakAndSigma(R_Vector,R)");
   cpmmessage(mL, loc&" started");
   Z i,n=x.dim();
   V<R> distr(n);
   cpmassert(n>0,loc);
   R sum=0;
   for (i=1;i<=n;i++){
      R w=cpmabs(x[i]);
      powFast(w,p);
      sum+=w;
      distr[i]=w;
   }
   cpmassert(sum>0,loc);
   R sumInv=1./sum;
   for (i=1;i<=n;i++) distr[i]*=sumInv;
   R iPeak=0;
   for (i=1;i<=n;i++) iPeak+=i*distr[i];
   R sigma=0;
   for (i=1;i<=n;i++){
      R di=(i-iPeak);
      di*=di;
      sigma+=di*distr[i];
   }
   cpmassert(sigma>0,loc);
   sigma=::sqrt(sigma);
```

```
   cpmmessage(mL, loc&" done");
   return R2(iPeak,sigma);
}

R_Vector CpmArrays::smoothen(R_Vector const& x, bool periodic)
{
   Z i,n=x.dim();
   R_Vector res(n);
   if (!periodic){
      for (i=2;i<n;i++){
         res[i]=0.25*x[i-1]+0.5*x[i]+0.25*x[i+1];
      }
      res[1]=x[1]*0.75+x[2]*0.25;
      res[n]=x[n]*0.75+x[n-1]*0.25;
   }
   else{
      for (i=1;i<=n;i++){
         res[i]=0.25*x.cyc(i-1)+0.5*x[i]+0.25*x.cyc(i+1);
      }
   }
   return res;
}

R_Vector CpmArrays::smoothen(R_Vector const& x, Z p, bool periodic)
{
   if (p<1) return x;
   R_Vector res=smoothen(x);
   Z pDone=1;
   while (p>pDone++) res=smoothen(res,periodic);
   return res;
}

X2< V<Z>, V<Z> >  CpmArrays::locExt(R_Vector const& x)
{
   Z n=x.dim();
   V<Z> minima(0),maxima(0);
   for (Z i=2;i<n;++i){ // active only for n > 2
      R xm=x[i-1];
      R x0=x[i];
      R xp=x[i+1];
      if (x0<xm && x0<xp) minima&=i;
      else if (x0>xm && x0>xp) maxima&=i;
      else ;
   }
   return X2< V<Z>,V<Z> >(minima,maxima);
}

R_Vector CpmArrays::derive(R_Vector const& x)
{
   Z n=x.dim();
```

```
   R_Vector res(n);
   for (Z i=1;i<n;i++){
      res[i]=x[i+1]-x[i];
   }
   res[n]=0;
   return res;
}

R_Vector CpmArrays::integrate(R x1, R_Vector const& x)
{
   Z n=x.dim();
   R_Vector res(n);
   R s=x1;
   for (Z i=1;i<=n;i++){
      res[i]=s;
      s+=x[i]; // x[n] is not used, is 0 if x was obtained by
   }
   return res;
}


R_Vector CpmArrays::accumulate(R_Vector const& x, bool average)
{
   Z n=x.dim();
   if (n<2) return x;
   R_Vector res(n);
   res[1]=x[1];
   for (Z i=2;i<=n;i++){
      res[i]=res[i-1]+x[i];
   }
   if (average){
      for (Z i=2;i<=n;i++){
         R iInv=1./i;
         res[i]*=iInv;
      }
   }
   return res;
}

R CpmArrays::extMean(R_Vector const& x)
{
   Word loc("CpmArrays::extMean(R_Vector)");
   Z sm=3;
   R_Vector y=smoothen(x,sm);
   y=accumulate(y,true);
   X2< V<Z>,V<Z> > xvv=locExt(y);
   V<Z> vMin=xvv.c1();
   V<Z> vMax=xvv.c2();
   Z dMin=vMin.dim(), dMax=vMax.dim();
   if (dMin==0 || dMax==0){
```

```
      cpmwarning(loc&": no extrapolation possible, plain mean taken");
      return y.last();
   }
   else{
      R lastMin=y[vMin.last()];
      R lastMax=y[vMax.last()];
      return (lastMin+lastMax)*0.5;
   }
}

void CpmArrays::makePositive(R_Vector& x, R tiny)
{
   Z n=x.dim();
   for (Z i=1;i<=n;i++){
      if (x[i]<tiny) x[i]=tiny;
   }
}

R_Vector CpmArrays::reverse(R_Vector const& x)
{
   Z n=x.dim();
   R_Vector res(n);
   Z n1=n+1;
   for (Z i=1;i<=n;i++){
      res[i]=x[n1-i];
   }
   return res;
}

R_Vector CpmArrays::monotonize(R_Vector const& x, Z pSmooth)
{
   R_Vector temp=smoothen(x,pSmooth);
      // Raw data might not yet be suitable for further processing,
      // thus they will be regularized.
      // Smoothen and derive commute, so it makes no sense to
      // smoothen the derivative got in a later step
   R peak=temp[1];
   Z n=x.dim();
   for (Z i=1;i<=n;i++){
      peak=(peak>temp[i] ? peak : temp[i]);
      temp[i]=peak;
   }
   return temp;
}

R CpmArrays::interpolate(R ic, R_Vector const& x)
{
   Z n=x.dim();
   if (ic<=1.){
      return x[1];
```

```
   }
   else if (ic>=n){
      return x[n];
   }
   else{
      R xiL=floor(ic);
      Z iL=(Z)xiL;
      cpmassert(iL<n,"CpmArrays::interpolate()");
      R p=ic-xiL;
      return x[iL]+p*(x[iL+1]-x[iL]);
   }
}

R CpmArrays::interpolate(R xp, R_Vector const& x, R_Vector const& y)
{
   Word loc("CpmArrays::interpolate(...)");
   Z n=x.dim();
   Z n2=y.dim();
   cpmassert(n==n2,loc);
   if (xp<=x[1]){
      return y[1];
   }
   else if (xp>=x[n]){
      return y[n];
   }
   else{
      Z iL=locate(x,xp);
      R xL=x[iL];
      R xU=x[iL+1];
      R yL=y[iL];
      R yU=y[iL+1];
      R dx=xU-xL;
      cpmassert(dx!=0,loc);
      R p=(xp-xL)/dx;
      return yL+p*(yU-yL);
   }
}

R CpmArrays::interpolateInverse(R xc, R_Vector const& x)
{
   Word loc("CpmArrays::interpolateInverse(...)");
   Z n=x.dim();
   if (xc<=x[1]) return 1.;
   if (xc>=x[n]) return n;
   Z i=locate(x,xc);
   cpmassert(i<n && i>0,loc);
   R nom=(x[i+1]-x[i]);
   cpmassert(nom>0.,loc);
   R p=(xc-x[i])/nom;
   cpmassert(p<1. && p>=0.,loc);
```

```
   return i+p;
}

R_Vector CpmArrays::moments(R_Vector const& data)
{
   Z mL=3;
   static Word loc("R_Vector CpmArrays::moments(R_Vector const& data)");
   cpmmessage(mL,loc&" started");
   Z j,n=data.dim();
   cpmassert(n>0,loc);
   if (n==1){
      R_Vector res(6,0);
      res[1]=data[1];
      return res;
   }
   R s=0.,p;
   for (j=1;j<=n;j++) s += data[j];
   R ave=s/n;
   R adev=0,svar=0,skew=0,curt=0;
   for (j=1;j<=n;j++) {
      adev += cpmabs(s=data[j]-ave);
      svar += (p=s*s);
      skew += (p*=s);
      curt += (p*=s);
   }
   adev /= n;
   svar /= (n-1);
   R sdev=sqrt(svar);
   if (svar) {
      skew /= (n*svar*sdev);
      curt=curt/(n*svar*svar)-3.0;
   }
   else{
      cpmwarning(
         "moment(): variance = 0 found, skew and curt set to 0 ",1);
           // number of warnings should be limited
      skew=0;
      curt=0;
   }
   R_Vector res(6);
   res[1]=ave;
   res[2]=adev;
   res[3]=sdev;
   res[4]=svar;
   res[5]=skew/sqrt(15./n);
   res[6]=curt/sqrt(96./n);
   cpmmessage(mL,loc&" done");
   return res;
}
```

```
//////////////////////////////// module statistics ////////////////////////

namespace{ // anonymous
R probks(R alam)
// Auxiliar function to ksone.
{
   const R EPS1=0.001,EPS2=1.0e-8;
   R fac=2.,sum=0.,term,termbf=0.;
   R a2 = -2.*alam*alam;
   for (Z j=1;j<=100;j++) {
      term=fac*exp(a2*j*j);
      sum += term;
      if (cpmabs(term) <= EPS1*termbf || cpmabs(term) <= EPS2*sum)
         return sum;
      fac = -fac;
      termbf=cpmabs(term);
   }
   return 1.0;
}

void ksone(R_Vector& data, R (*func)(R x), R& d, R& prob)
/*
Given an array data[1..n], and given a user-supplied function of a single
variable func which is a cumulative distribution function ranging from 0
(for smallest values of its argument) to 1 (for largest values of its
argument), this routine returns the 'Komogorov-Smirnov statistic' d, and
the significance level prob. Small values of prob show that the
cumulative function of data is significantly different from func.
The array data is modified by being sorted into ascending order.
*/

{
   Z n=data.dim();
   R fo=0.,fn,ff,en,dt;
   data.order();
   en=n;
   d=0.;
   for (Z j=1;j<=n;j++) {
      fn=j/en;
      ff=(*func)(data[j]);
      dt = CpmRootX::sup(cpmabs(fo-ff),cpmabs(fn-ff));
      if (dt > d) d=dt;
      fo=fn;
   }
   prob=probks(sqrt(en)*d);
}

R erfcc(R x)
{
   R t,z,ans;
```

```
   z=cpmabs(x);
   t=1.0/(1.0+0.5*z);
   ans=t*exp(-z*z-1.26551223+t*(1.00002368+t*(0.37409196+t*(0.09678418+
      t*(-0.18628806+t*(0.27886807+t*(-1.13520398+t*(1.48851587+
      t*(-0.82215223+t*0.17087277)))))))));
   return (x >= 0.0 ? ans : 2.0-ans);
}

R mean_stat, sigma_stat;
R isr2_stat=1./sqrt(2.);
R cum_stat(R x)
// cumulative distribution of the best fitting normal distribution.
// Used for Kolmogorov-Smirnov test
{
   R x_n=(x-mean_stat)/sigma_stat;
   x_n*=isr2_stat;
   return 1-0.5*erfcc(x_n);
}

bool neg(const bool& b){ return !b;}

} // anonymous namespace

R_Vector CpmArrays::statistics(R_Vector const& dat)

{
   Z n=dat.dim();
   R_Vector res1=moments(dat);
   R norm = sqrt(n/24.);
   mean_stat=res1[1];
   sigma_stat=res1[3];
   res1[6]*=norm;
   res1[5]*=(norm*2);
   R d=0,prob=0;
   R_Vector datN=dat;
   ksone(datN,cum_stat,d,prob);
      // now datN are in ascending order
   R_Vector res2(4);
   res2[1]=prob;
   res2[2]=datN[1];
      // minimum
   res2[3]=( n==2*(n/2) ? 0.5*(datN[n/2]+datN[1+n/2]) : datN[(1+n)/2] );
      // median
   res2[4]=datN[n];
      // maximum

   return res1&res2;
      // appending the results (efficient return)
}
```

```
R_Vector CpmArrays::bimodalAnalysis(R_Vector const& dat)
{
   Word loc("R_Vector CpmArrays::bimodalAnalysis(R_Vector)");
   Z i,n=dat.dim();
   R3 ims=dat.infMeanSup();
   R mean0=(ims[1]+ims[3])*0.5; // between min and max
   R mean1=(ims[2]+mean0)*0.5; // taking mean also into account
   V<bool> sel(n);
   for (i=1;i<=n;i++) sel[i]=(dat[i]>=mean1);
   R_Vector upper=dat.select(sel); // the valuea above mean
   sel=sel(F<bool,bool>(neg)); // elegant modification of sel
   R_Vector lower=dat.select(sel);
      // the values below limit
   R_Vector res(2);
   R rL=lower.mean();
   R rU=upper.mean();
   cpmassert(rL<=rU,loc);
   res[1]=rL;
   res[2]=rU;
   return res;
}

//////////////////////////// end of module statistics() /////////////////

R_Vector CpmArrays::gradient( const R_Matrix& x, R_Vector const& y)
{
   cpmmessage("R_Vector gradient() started");
   try{
   Z n=y.dim();
   Z d=x.dim2();
   cpmassert( n==x.dim1()&&n>d,"CpmArrays::gradient(R_Matrix,R_Vector)");
      // since d>=1 as for each dimension in R_Matrix, this implies n>1
   R_Vector dy(n-1);
   R_Matrix dx(n-1,1);
   for (Z i=1;i<n;i++){
      dy[i]=y[i+1]-y[1];
      dx[i]=x[i+1]-x[1];
   }
   return (!dx)*dy;
   }
   catch(...){
      Error("Problem in gradient( const R_Matrix&, R_Vector const&)");
      return R_Vector();
   }
}

R_Matrix CpmArrays::gradient( const R_Matrix& x, const R_Matrix& y)
{
   cpmmessage("R_Matrix gradient() started");
```

```
   try{
   Z n=y.dim();
   Z d=x.dim2();
   cpmassert( n==x.dim1() && n>d,"gradient(R_Matrix,R_Matrix)");
      // since d>=1 as for each dimension in R_Matrix, this implies n>1
   R_Matrix dy(n-1,1);
   R_Matrix dx(n-1,1);
   for (Z i=1;i<n;i++){
      dy[i]=y[i+1]-y[1];   // now the y[i] are vectors of dimension p, dy
      // is of matrix type (n-1,p)
      dx[i]=x[i+1]-x[1];   // dx is of matrix type (n-1,d)
   }
   return (!dx)*dy;     // !dx is of type (d,n-1), thus result of type
   // (d,p), OK
   }
   catch(...){
      Error("Problem in gradient( const R_Matrix&, const R_Matrix&)");
      return R_Matrix();
   }
}

//////////////////////////////////// geometry ////////////////////////////////

void CpmArrays::rotation(R_Vector& x, R_Vector const& a,
   R_Vector const& b)
{

// preliminaries: form two orthonormal vectors out of a,b. x is not used

   Z n1=x.dim(), n2=a.dim(), n3=b.dim();
   if (n1!=n2 || n2!=n3)
   cpmerror("rotation(R_Vector,R_Vector,R_Vector): dimension mismatch");
   R aa=a.absVal(), bb=b.absVal();
   if (aa==0. || bb==0.)
   cpmerror(
      "rotation(R_Vector,R_Vector,R_Vector): zero vector not allowed");
   aa=1./aa; bb=1./bb;
   R_Vector a1=a*aa;
   R_Vector b1=b*bb; // now a1 and b1 are unit vectors
   if (a1==b1) return; // nothing to do
   R c=(b1|a1);
   R s=1-c*c;
   if (s<0.)
   cpmerror("rotation(R_Vector,R_Vector,R_Vector): unexpected error");
   s=sqrt(s);
   R_Vector a2=b1-c*a1;
   aa=a2.absVal();
   if (aa==0.)
   cpmerror("rotation(R_Vector,R_Vector,R_Vector): unexpected error");
   aa=1./aa;
```

```
   a2*=aa; // now a1 and a2 are orthonormal vectors

// action on x

   R x1=(x|a1);
   R x2=(x|a2);
   R y1=c*x1-s*x2;
   R y2=s*x1+c*x2;
      // The vector x1*a1+x2*a2 will be rotated to y1*a1+y2*a2
      // and the orthogonal part x-x1*a1+x2*a2 of x will be left
        // unchanged
   x+=(y1-x1)*a1+(y2-x2)*a2;
}

void CpmArrays::simplex(R_Matrix& s, Z d)
{
   Z i,j,d1=d+1;
   R d1Inv=1./d1;
   R_Matrix s_(d1,d);
   s=s_;
   R_Vector e0(d1); // zero vector
   R_Matrix e(d1); // unit matrix
      // The d1 corners of the simplex in d-dimensional space are
      // formed out of the d1 canonical basis vectors in d1-dimensional
      // space. After 'removing the mean value' they all belong to a
      // hyper plane which finally is rotated into a natural
      // d-dimensional subspace
   for (i=1;i<=d1;i++) e0+=e[i];
   e0*=d1Inv; // mean value of the e[i]
   R_Vector b=e[d1];
   R_Vector a=e0;
   for (i=1;i<=d1;i++) e[i]-=e0; // now the d+1 vectors e[i] have
      // zero sum
   R r=e[1].absVal();
   if (r==0.)
   cpmerror("simplex: unexpected condition r=0");
    r=1./r;
   for (i=1;i<=d1;i++) rotation(e[i],a,b);
   for (i=1;i<=d1;i++){
      for (j=1;j<=d;j++){
         s[i][j]=r*e[i][j];
      }
   }
}

Z2 CpmArrays::linearToMatrix(Z k, Z m, Z n)
{
   if (m<1 || n<1)
      cpmerror("CpmArrays::linearToMatrix(...) : m<1 || n<1");
   Z mn=m*n;
```

```
   k--;
   while (k<0) k+=mn;
   while (k>=mn) k-=mn;
   Z p=k/n; // result varies from 0 to m-1
   Z q=k%n; // result varies from 0 to n-1
   return Z2(p+1,q+1);
}

Z CpmArrays::matrixToLinear(Z i, Z j, Z m, Z n)
{
   if (m<1 || n<1)
      cpmerror("CpmArrays::matrixToLinear(...) : m<1 || n<1");
   i--;
   while (i< 0) i+=m;
   while (i>=m) i-=m; // now i in {0,...,m-1}
   j--;
   while (j< 0) j+=n;
   while (j>=n) i-=n; // now j in {0,...,n-1}
   return i*n+j+1; // runs from 1 to m*n OK !
}

R_Vector CpmArrays::binomialCoefficients(Z n)
{
   Word loc("CpmArrays::binomialCoefficients(Z)");
   cpmassert(n>=-1,loc);
   Z np=n+1;
   if (np==0) return R_Vector(0);
   R_Vector res(np,1.); // initialized with 1 is OK
   if (np<=2){
      return res;
   }
   else if (np==3){
      res[2]=2;
      return res;
   }
   else if (np==4){
      res[2]=3;
      res[3]=3;
      return res;
   }
   else{
      R zaehler=n;
      R nenner=1;
      Z j=2;
      Z k=n;
      while (k>=j){
         res[j]=res[j-1]*zaehler/nenner;
         if (k>j) res[k]=res[j];
         zaehler-=1;
         nenner+=1;
```

```
         k--;
         j++;
      }
      return res;
   }
}

V< X2<Z,R> > CpmArrays::levelCrossing(R_Vector const& y, R frac,
   bool autoLevel, R levelSpezifier)
{
   Z mL=3;
   static Word loc("CpmArrays::levelCrossing(...)");
   cpmmessage(mL,loc&" started");
   Z i,n=y.dim();
   if (n<1) return V< X2<Z,R> >(0);
   R threshold;
   if (!autoLevel) threshold=levelSpezifier;
   else{
      Percentilizer per(y);
      threshold=per(levelSpezifier);
   }
   threshold*=frac;
   vector<Z> vz;
   vector<R> vr;
   R yPrev=y[1]; // we start below threshold
   for (i=1;i<=n;i++){
      R yi=y[i];
      bool b1=(yPrev<threshold);
      bool b2=(yi<threshold);
      if (b1==b2){
         ;
      } // from now on b1!=b2; thus (b1 && !b2) or (!b1 && b2)
      //  yi and yPrev lie on different sides of threshold
      // thus yi!=yPrev ==> (yi-yPrev)!=0
      else if (b1){ // ascending passage of threshold
         vz.push_back(1);
         // (threshold-yPrev):p=(yi-yPrev):1
         // p=(threshold-yPrev)/(yi-yPrev)
         R p=(threshold-yPrev)/(yi-yPrev);
         vr.push_back(i+p);
      }
      else if (!b1){ // descending passage of threshold
         vz.push_back(-1);
         // (yPrev-yi):1=(yPrev-threshold):p ==>
         // p*(yPrev-yi)=(yPrev-threshold) ==>
         // p=(yPrev-threshold)/(yPrev-yi) ==>
         // p=(threshold-yPrev)/(yi-yPrev) (same formula as in
         // previous case
         R p=(threshold-yPrev)/(yi-yPrev);
         vr.push_back(i+p);
```

```
      }
      yPrev=yi;
   }
   Z nres=(Z)vz.size();
   V< X2<Z,R> > res(nres);
   for (i=1;i<=nres;i++) res[i]=X2<Z,R>(vz[i-1],vr[i-1]);
   cpmmessage(mL,loc&" done");
   return res;
}

Iv CpmArrays::pairedLevelCrossing(R_Vector const& y, R frac,
   bool autoLevel, R levelSpezifier)
{
   V< X2<Z,R> > arr=levelCrossing(y,frac,autoLevel,levelSpezifier);
   Z i,n=arr.dim();
   if (n<1) return Iv();
   Z z1=arr[1].c1();
   R x1=arr[1].c2();
   R x2=0;
   bool pairingFound=false;
   for (i=1;i<=n;i++){
      Z zi=arr[i].c1();
      if (zi!=z1){
         x2=arr[i].c2();
         pairingFound=true;
      }
   }
   if (!pairingFound) return Iv();
   else return Iv(x1,x2);
}

//////////////////////////////// function ... ////////////////////////

   V< V<R2> > CpmArrays::homDataRan(V< V<R2> > const& x)
   {
      Word loc("CpmArrays::homDataRan(V< V<R2> >");
      Z d=x.dim();
      Z par=d%2;
      cpmassert(par==0,loc);
      V< V<R2> > y=x;
      Z i,j,dh=d/2;
      for (i=1;i<=dh;++i){
         Z di=x[i].dim();
         Z is=i+dh;
         Z dis=x[is].dim();
         R_Vector vi(di);
         for (j=1;j<=di;++j){
            vi[j]=x[i][j][2];
         }
         R_Vector vis(dis);
```

```
      for (j=1;j<=dis;++j){
         vis[j]=x[is][j][2];
      }
      R mi=vi.supAbs();
      R mis=vis.supAbs();
      mi=CpmRootX::sup<R>(mi,mis);
      if (mi>0.){
         mi=1./mi;
         for (j=1;j<=di;++j){
            y[i][j][2]*=mi;
         }
         Z is=i+dh;
         for (j=x[is].b();j<=x[is].e();++j){
            y[is][j][2]*=mi;
         }
      }
   }
   return y;
}

V< V<R2> > CpmArrays::difOfCorData(V< V<R2> > const& xa, bool homFirst)
{
   Word loc("CpmArrays::homDataRan(V< V<R2> >");
   V< V<R2> > x= homFirst ? homDataRan(xa): xa;
   Z d=x.dim();
   Z par=d%2;
   cpmassert(par==0,loc);
   R tol=1e-6;

   Z i,j,dh=d/2;
   V< V<R2> > res(dh);
   for (i=1;i<=dh;++i){
      Z is=i+dh;
      Z di=x[i].dim();
      Z dis=x[is].dim();
      cpmassert(di==dis,loc);
      V<R2> resi(di);
      for (j=1;j<=di;++j){
         R ti=x[i][j][1];
         R tis=x[is][j][1];
         R di=CpmRoot::disVal(ti,tis);
         cpmassert(di<=tol,loc);
         R tim=(ti+tis)*0.5;
         R yi=x[is][j][2]-x[i][j][2];
         resi[j]=R2(tim,yi);
      }
      res[i]=resi;
   }
   return res;
}
```

```
bool CpmArrays::toGnuPlot(ostream& str, V< V<R2> > const& x )
{
   Z mL=3;
   Word loc("CpmArrays::toGnuPlot(ostream,V< V<R2> >)");
   CPM_MA
   Z n=x.dim(),i,j;
   Z n1=x[1].dim();
   Z err=0;
   for (i=2;i<=n;++i) err+=cpmpos(n1-x[i].dim());
   cpmassert(err==0,loc);
   for (j=1;j<=n1;++j){ // each j writes a line of data to stream
      for (i=1;i<=n;++i){ // each i generates a pair (t,y) in a
         // data line
         R2 val=x[i][j];
         R t=val[1];
         R y=val[2];
         str<<t<<" "<<y<<" ";
      }
      str<<endl;
   }
   CPM_MZ
   return str ? true : false;
}

bool CpmArrays::toGnuPlot(Word const& fileName, V< V<R2> > const& x )
{
   Z mL=3;
   Word loc("CpmArrays::toGnuPlot(Word,V< V<R2> >)");
   CPM_MA
   CpmSystem::OFile ofi(fileName);
   bool res=CpmArrays::toGnuPlot(ofi(),x );
   CPM_MZ
   return res;
}
```

221 cpmviewport.h

```
//? cpmviewport.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_VIEWPORT_H_
#define CPM_VIEWPORT_H_
/*

   class Viewport is the Cpm interface to the graphical
   capabilities of the hardware.

*/
#include <cpmvl.h>

#if !defined(CPM_NOGRAPHICS)
   // otherwise we don't need graphics libraries
   #if defined(WIN32)
   // Microsoft Compiler Linker, together with the
   // Windows glut system in my C:\opt\cppms
      #if defined(MINGW)
         #include <c:/opt/Dev-Cpp/include/GL/openglut.h>
      #else
         #include <glut.h>
      #endif
   #else
   //  'Cygwin' or 'Linux'
      #include <GL/glut.h>
   #endif
#endif

namespace CpmGraphics{

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::L;
   using CpmRoot::Word;
   using CpmArrays::Vl;
   using namespace std;

#if defined(WIN32)
   const Word Platform("Windows");
#else
   const Word Platform("Cygwin or Linux");
#endif

//////////////////////////////////// struct xy ///////////////////////////
// simple data structure for graphical points. As ever in graphics:
// x-axis horizontal from left to right (thus perfectly normal)
```

```
// y-axis downwards.
   struct xy{ // graphical points
      Z x,y;
      xy(Z a=0, Z b=0):x(a),y(b){}
      xy operator+(xy p)const{ return xy(x+p.x,y+p.y);}
      xy operator-(xy p)const{ return xy(x-p.x,y-p.y);}
      xy& operator+=(xy p){ x+=p.x;y+=p.y;return *this;}
      xy& operator-=(xy p){ x-=p.x;y-=p.y;return *this;}
      bool operator==(xy p)const{ return x==p.x && y==p.y;}
   };

// simple data structure for graphical rectangles

   struct xyxy{ // graphical rectangles
      Z x1,y1,x2,y2; // (x1,y1) left upper corner,
         // (x2,y2) right lower corner

      xyxy(Z a=0,Z b=0,Z c=0,Z d=0):x1(a),y1(b),x2(c),y2(d){}
         // construction from corner coordinates

      xyxy(xy p1, xy p2):x1(p1.x),y1(p1.y),x2(p2.x),y2(p2.y){}
         // construction from corners

      xyxy(xy p, Z w, Z h):x1(p.x),y1(p.y),x2(x1+w-1),y2(y1+h-1){}
         // construction from left upper corner, width, and height

      Z w()const{ return x2-x1+1;}

      Z h()const{ return y2-y1+1;}

      bool isVoid()const{ return x1<0;}
         // since the meaningfull graphical rectangles have no
         // negative components, this is an easy way to encode
         // exceeptional behavior

      xyxy operator+(xy p)const
         { return xyxy(x1+p.x,y1+p.y,x2+p.x,y2+p.y);}
      xyxy operator-(xy p)const
         { return xyxy(x1-p.x,y1-p.y,x2-p.x,y2-p.y);}
      xyxy& operator+=(xy p){ return *this=*this+p;}
      xyxy& operator-=(xy p){ return *this=*this-p;}

      xyxy operator|(xyxy r)const;
         // smallest xyxy that contains both *this and r
         // kind of union

      xyxy& operator|=(xyxy r){ return *this=(*this)|r;}
         // | as a mutating operation

      xyxy operator&(xyxy r)const;
```

```
         // returns the set section of *this and r.
         // The result res satisfies res.isVoid()==true
         // if this section is the void set.

      xyxy& operator&=(xyxy r){ return *this=(*this)&r;}
         // & as a mutating operation

      xy operator[](Z i)const{ return i<=1 ? xy(x1,y1) : xy(x2,y2);}
         // For an instance r of xyxy r[1] is the upper left corner
         // and r[2] is the lower right corner.

      xy cor(Z i)const // corner
      // Returns the i-th corner. Here corners are numbered counter
      // clock-wise starting with the left upper corner as 1. All
      // non-matching arguments give corner 1. Thus no exceptions!
      {
         if (i==2) return xy(x1,y2);
         if (i==3) return xy(x2,y2);
         if (i==4) return xy(x2,y1);
         return xy(x1,y1);
      }
   };

   struct rgb;
   class Img24;

//////////////////////// class ColRef ////////////////////////////////

const R i255=1./255;

   class ColRef{ //lean 24-bit color-values
      // light 24-bit representation of color-values
      // that easily converts (back and forth) to the low-level color
      // data

      friend struct rgb;
      friend class Img24;
      static Z lowerLimit;
      static Z upperLimit;
      static Z x(Z z)
      {
         if (z>upperLimit) return upperLimit;
         else if (z<lowerLimit) return lowerLimit;
         else return z;
      }
      typedef ColRef Type;
   public:
      L r,g,b; // all values valid, so public OK
      CPM_IO
      ColRef():r((L)lowerLimit),g((L)lowerLimit),b((L)lowerLimit){}
```

```
      ColRef(Z r_, Z g_, Z b_):r((L)x(r_)),g((L)x(g_)),b((L)x(b_)){}
      ColRef(L r_, L g_, L b_):r(r_),g(g_),b(b_){}
      L getR()const{ return r;}
      L getG()const{ return g;}
      L getB()const{ return b;}
      L operator[](Z i)const
         { if (i==3) return b; else if (i==2) return g; else return r;}
      R glR()const{ return i255*r;}
      R glG()const{ return i255*g;}
      R glB()const{ return i255*b;}
      bool isGray()const{ return r==g && g==b;}
      ColRef add(ColRef const& cr)const
      { return ColRef((Z)r+(Z)cr.r,(Z)g+(Z)cr.g,(Z)b+(Z)cr.b);}
      Word nameOf()const{ return "ColRef";}
      bool operator!=(ColRef cr)const
         // has to do one, two, or three tests
         // depending on the arguments
      {
         if (r!=cr.r) return true;
         if (g!=cr.g) return true;
         if (b!=cr.b) return true;
         return false;
      }

      bool operator==(ColRef cr)const
         // just as with != different ColRefs cause less
         // computational effort than equal ones.
      {
         if (r!=cr.r) return false;
         if (g!=cr.g) return false;
         if (b!=cr.b) return false;
         return true;
      }

    };

//////////////////////////// struct rgb ////////////////////////////////

struct rgb { // Z-valued red green blue
   // a simple struct to provide a convenient interface for
   // functions draw. This class will be a base class for deriving
   // a more functional class Color in namespace CpmImaging.
   Z r,g,b;

   rgb():r(ColRef::lowerLimit),g(ColRef::lowerLimit),
      b(ColRef::lowerLimit){}

   rgb(Z r_, Z g_, Z b_, bool norm=true):r(r_),g(g_),b(b_)
      {if (norm) normalize();}
```

```
   explicit rgb(ColRef cr):r(cr.r),g(cr.g),b(cr.b){}

   virtual Word nameOf()const{ return Word("rgb");}

   operator ColRef()const{ return ColRef(r,g,b);}

   void normalize(){ r=ColRef::x(r); g=ColRef::x(g); b=ColRef::x(b);}
      // ensuring the right value range

   static rgb mix(rgb c1, rgb c2, R w1, R w2, R gamma);
      // mixing colors according to function mixingRule

   R gray()const{return (r+g+b)/3.;}

   bool isGray()const{ return r==g && g==b;}

   rgb operator*(R fac)const
   { return rgb(cpmround(r*fac),cpmround(g*fac),cpmround(b*fac));}

   static Z mixingRule(Z z1, Z z2, R w1, R w2, R gamma);
      // defines mixing of colors according to given weights
      // w1+w2=1 is not assumed
      // ( w1=w2=1 corresponds to addition of light)

   void write(ostream& str)const
   { str<<endl<<"rgb: r="<<r<<" g="<<g<<" b="<<b<<endl;}

   friend void operator<<(ostream& str, rgb const& c)
   { str<<c.r; str<<c.g; str<<c.b;}

   friend void operator>>(istream& str, rgb & c)
   { str>>c.r; str>>c.g; str>>c.b;}
};

////////////////////////////// class Font ////////////////////////////////
// Presently only Helvetica 12 needed from this font system

class Font{
// only Helvetica 12 needed from this font system of GLUT
   void* name;
   Z h;
      // height in pixels
   Z mw;
      // maximum width in pixels (estimation based on height)
   Z d;
      // suitable line separation taking decent ('Unterlaenge') into
      // account
public:
   explicit Font(Z type=4);
   explicit Font(R height);
```

```
      // creates an instance of a font height that comes closest
      // to the value given by the argument. See the simple
      // implementation
      // code for the details

   Z getHeight()const{ return h;}
   Z getWidth()const{ return mw;}
      // maximum width in pixels (typically width of W in proportional
      // fonts
   Z getLineSep()const{ return d;}
   void* getName()const{ return name;}
};

//////////////////////// class Rec ////////////////////////////////////
// Pixel rectangle with rec_ which always fits Viewport::win()
// and linear image buffer mem_ of arbitrary size.
// The display function dis of this class shows mem_
// always whithin rec_. This function is Cpm's means
// to bring pixel-based graphical information to screen.
// Present implementation relies on OpenGL (only if
// CPM_NOGRAPHICS is not set). If CPM_NOGRAPHICS is
// defined we need no graphics libraries from the system
// and can generate all graphics as PPM-images (on an
// pixel rectangle that can be set arbitrarily
// by editing cpmconfig.ini

class Rec{ // pixel rectangle which always fits Viewport::win()
   xyxy rec_;
   Z n_;
      // dimension of mem_
   L* mem_;
      // should be unsigned char = L
      // Data element of this type needed in function dis()
      // by OpenGL. This is linearized data in two-fold manner:
      // The rectangular image area is un-fold to a linear chain,
      // and the r g b pixel data are represented as a list of
      // characters.

public:
   explicit Rec(xyxy r=xyxy(0,0,0,0));
      // Constructor which fills the image data from the one and only
      // initialized Viewport, more precisely from the data
      // within the set section of r and the Viewport rectangle
      // Viewport::win()
      // Present implementation relies on OpenGL
   explicit Rec(Vl< Vl<ColRef> > const& bh);
      // constructs a Rec from the matrix data bh. The corresponding
      // rectangle rec_ is created such that it fits Viewport::win().
   Rec(xyxy r, L c);
      // The pixels within the rectangle r are set equal to c
```

```
      // On display this is always a gray (black to white) frame.
      // Notice that colorful colors would be more difficult to
      // handle here.
      // Rectangle is created such that it fits Viewport::win().
   Rec(Rec const& r);
   ~Rec();
   Rec& operator=(Rec const& r);
   Z dim()const{ return n_;}
      // number of bytes = 3*number of pixels
   Z h()const{ return rec_.h();}
   Z w()const{ return rec_.w();}
   Z x1()const{ return rec_.x1;}
   Z y1()const{ return rec_.y1;}
   Z x2()const{ return rec_.x2;}
   Z y2()const{ return rec_.y2;}
   xy operator[](Z i)const{ return rec_[i];}
   Rec& operator+=(xy shift);
      // result is always within Viewport::win()
   Rec operator+(xy shift)const{ Rec res=*this; res+=shift; return res;}
   Rec& operator-=(xy shift);
      // result is always within Viewport::win()
   Rec operator-(xy shift)const{ Rec res=*this; res-=shift; return res;}
   void dis()const;
      // display; displays the image data in mem_ on the area rec_
      // On this function all bitmap imaging in C+- is based.
      // Present implementation relies on OpenGL. Function glDrawPixels
      // needs a L* as the last argument. So it would not be convenient to
      // replace the mem_ data element of the present class by one of type
      // Vl<L>
   Vl< Vl<ColRef> > fold()const;
      // returns a matrix-like memory version of mem_
};

//////////////////////// class Viewport ////////////////////////////////
/*
class Viewport is a lean interface to the
systems graphical capabilities. It also defines and activates a
statusbar supporting a continuous message stream from a running
program going to a dedicated area of the screen.

In order for the OpenGL version to work (which is the only one that
survived, I had working implementations based on MSFC and on FLTK)
one has to do some initialisation in function main() as is done in
cpmapplication.cpp.

The class as it stands is a 'blind viewport' since it allows  n o t
to perform actions which can be seen on screen.
*/

class Viewport{ // lean interface to the system's graphical capabilities
```

```
   // The Viewport class has no non-static data member, so
   // every member function could be declared static without changing
   // their behavior. However, it is more convenient to say
   //
   // Viewport vp;
   // vp.addText(10,10,"hello world");
   //
   // than
   //
   // Viewport::addText(10,10,"hello world");
   //
   // Notice that there is no way to have more than one different
   // instances of Viewport. So we can control a single graphical area,
   // which in my normal applications is one 'application window'.
   // Classes Frame, Graph, and Frames organize as many rectangular
   // subframes of this basic window as we want.
   static bool initialized;
   static Z applWindowWidth;
   static Z applWindowHeight;
   static Z fullWindowHeight;
   static xyxy window;
   static xyxy fullWindow;
   static R gamma;
      // a gamma-value for the systems video screen
      // presently set to 2.2 (usual in Windows systems)
      // in code. Would not be illogical to read this in from
      // cpmsystemdependencies.h
      // This value of gamma will be used as default for the
      // image class Img24
   static ColRef colText;
   static xyxy seg1;
   static xyxy seg2;
   static xyxy seg3;
   static xyxy seg4;
   static ColRef cs1;
   static ColRef cs2;
   static ColRef cs3;
   static ColRef cs4;
   static Z heightStatusBar;
   static Z size;
   static Z rank;

public:
   static void setColText(Z r, Z g, Z b){ colText=ColRef(r,g,b);}
   static void setColText(ColRef cr){ colText=cr;}
   static void setColText(rgb crgb){ colText=(ColRef)crgb;}
   static ColRef getColText(){ return colText;}

   static xyxy win(){ return window;}
   static xyxy fullWin(){ return fullWindow;}
```

```
static Z getHeightStatusBar(){ return heightStatusBar;}
static void onStatusBar(const Word& w, Z i);
static void clrPan();
   //: clear panes
static void fill(xyxy rec, ColRef cr);
   // Fills rectangle rec with solid color cr
static void placeWord(Z tx, Z ty, Word const& w,
   const Font& font, Z wMax=500);
   // wMax sets the maximum number of pixels for the length of the
   // printed picture of w. If w is longer it gets cut ( e.g. for
   // disciplined writing on status bars.
static xyxy textLine(Z tx, Z ty, Word const& w,
   ColRef crText, R h);
   // writing w in color crText.
   // Only the pixels making up the characters are written so that
   // previous screen content is only minimally shadowed by the text.
   // The text may not be easy to read, however.
   // h is a proposal for the font hight (the actually chosen
   // hight depends on the available fonts). Introduced
   // 2005-04-14
   // The return value is a enclosing box for the text - a bit larger
   // than the minimum enclosing box.

Viewport(Z w, Z h, Word const& title);
Viewport(){}

static R getGamma(){ return gamma;}
static void setGamma(R g){ gamma=g;}
static bool isInitialized(){return initialized!=0;}
   // does not allow to change status !!
static Z pelX(){ return applWindowWidth;}
static Z pelY(){ return applWindowHeight;}

Z getCol()const{return applWindowWidth;}
   // number of columns
   // horizontal extension of the graphically accessible
   // application window in pixels. The range of the first pixel
   // access index (mostely named x or i) is {0,1,...,getCol()-1}
Z getWidth()const{return applWindowWidth;}
   // legalizing another conventional name

Z getLin()const{ return applWindowHeight;}
   // number of lines (rows)
   // vertical extension of the graphically accessible
   // application window in pixels. The range of the second pixel
   // access index (mostely named y or j) is {0,1,...,dim2()-1}
Z getHeight()const{ return applWindowHeight;}
   // legalizing another conventional name
```

```
   Z getFullHeight()const{return fullWindowHeight;}

   Z dim1()const{return applWindowWidth;}
   Z dim2()const{return fullWindowHeight;}
      // The screen area under control of Viewport consists of
      // dim1()*dim2() writable and readable pixels. In the case
      // of OpenGL-implementation a part of the lines are used as
      // forming a statusbar. So these should not be used for
      // displaying images and graphics (but they perfectly
      // can from a syntactic point of view)

   Z getRank()const{ return rank;}
   Z getSize()const{ return size;}

   xyxy addText(Z x1, Z y1, Word const& text, R fontHeight=12, Z font=1);
      // (x1,y1) is the upper left corner of the writing box.
      // The return value is a box which encloses the text and is a bit
      // larger than minimal.
      // Thus the last argument remains passive since we select the font
      //  the height of which comes closest to fontHeigt.

      // Here we are the names of the available fonts together with a
      // number which does not matter here.
      // font = 1 : GLUT_BITMAP_TIMES_ROMAN_10
      // font = 2 : GLUT_BITMAP_HELVETICA_10
      // font = 3 : GLUT_BITMAP_HELVETICA_12
      // font = 4 : GLUT_BITMAP_8_BY_13
      // font = 5 : GLUT_BITMAP_9_BY_15
      // font = 6 : GLUT_BITMAP_HELVETICA_18
      // font = 7 : GLUT_BITMAP_TIMES_ROMAN_24
};

}// namespace

#endif
```

222 cpmviewport.cpp

```
//? cpmviewport.cpp
//? Status of work 2008-10-25.
//?
#include <cpmviewport.h>

using CpmRoot::Z;
using CpmRoot::L;
using CpmRoot::R;
using CpmRoot::Rh;
using CpmRoot::Word;
using CpmRoot::getByte;

using CpmSystem::Message;

using CpmArrays::Vl;

using CpmGraphics::Rec;
using CpmGraphics::xy;
using CpmGraphics::xyxy;
using CpmGraphics::Viewport;
using CpmGraphics::rgb;
using CpmGraphics::ColRef;

using namespace std;

Z CpmGraphics::Viewport::heightStatusBar=16;
//////////////////////// struct xyxy ////////////////////////////////////
namespace {
Z infZ(Z i, Z j){ return i<=j ? i : j;}
Z supZ(Z i, Z j){ return i<=j ? j : i;}
}

xyxy xyxy::operator|(xyxy r)const
{
   Z rx1=infZ(x1,r.x1);
   Z ry1=infZ(y1,r.y1);
   Z rx2=supZ(x2,r.x2);
   Z ry2=supZ(y2,r.y2);
   return xyxy(rx1,ry1,rx2,ry2);
}

xyxy xyxy::operator&(xyxy r)const
{
   if (x2 < r.x1) return xyxy(-1,-1,-1,-1);
   if (r.x2 < x1) return xyxy(-1,-1,-1,-1);
      // if upper end of one is smaller than the lower
      // end of the other, then the projections of
```

```
      // *this and r to the x-axis ared disjoint
   if (y2 < r.y1) return xyxy(-1,-1,-1,-1);
   if (r.y2 < y1) return xyxy(-1,-1,-1,-1);
      // same for x replaced by y
   Z rx1=supZ(x1,r.x1);
   Z ry1=supZ(y1,r.y1);
   Z rx2=infZ(x2,r.x2);
   Z ry2=infZ(y2,r.y2);
   return xyxy(rx1,ry1,rx2,ry2);
}

Rec& Rec::operator+=(xy shift)
{
   (rec_+=shift)&=Viewport::win(); return *this;
}
Rec& Rec::operator-=(xy shift)
{
   (rec_-=shift)&=Viewport::win(); return *this;
}
// the following implementation follows that of Vl<> in cpmvl.h

Rec::~Rec(){ delete[] mem_;}

Rec::Rec(xyxy r): rec_(r&Viewport::win()),
   n_(rec_.w()*rec_.h()*3),mem_(new L[n_])
// together with the usage in Rec::dis, this is all we need from OpenGL
{
#if !defined(CPM_NOGRAPHICS)
   glFlush();
   Z x1=rec_.x1, y1=rec_.y1, w=rec_.w(), h=rec_.h(), wBytes=3*w;
   L* p=mem_;
   Z gly=Viewport::fullWin().h()-y1-h;
   Z j=h;
   while (j--){
     L* buff=new L[wBytes];
     glReadPixels(x1,gly++,w,1,GL_RGB,GL_UNSIGNED_BYTE,buff);
     Z ib=wBytes;
     L* q=buff;
     while (ib--) *p++=*q++;
     delete[] buff;
   }
#endif
}

Rec::Rec(xyxy r, L c): rec_(r&Viewport::win()),
   n_(rec_.w()*rec_.h()*3),mem_(new L[n_])
{
   L* p=mem_;
   Z j=n_;
   while (j--)*p++=c;
```

```
}
      // The pixels within the rectangle r are set equal to c
      // On display this is always a gray (black to white) frame.
      // Notice that colorful colors would be more difficult to
      // handle here

Rec& Rec::operator=(Rec const& r)
{
   if (this==&r) return *this;
   rec_=r.rec_;
   Z nMem=n_;
   n_=r.n_;
   if (n_!=nMem){ //  avoid superfluous delete-new action
      delete[] mem_;
      mem_=new L[n_];
   }
   L* q=r.mem_;
   L* it=mem_;
   Z i=n_;
   while (i--) *it++ = *q++;
   return *this;
}

Rec::Rec(Rec const& r):rec_(r.rec_),n_(r.n_)
{
   L* itp=r.mem_;
   L* q=new L[n_];
   L* it=q;
   Z i=n_;
   while (i--) *it++=*itp++;
   mem_=q;
}

void Rec::dis()const
// screen representation of a linear block of pixel values.
// Basicly this is all we need from OpenGL
{
#if !defined(CPM_NOGRAPHICS)
   Z h1=rec_.h();
   Z w1=rec_.w();
   glRasterPos2i(rec_.x1,rec_.y1+h1);
   glDrawPixels(w1,h1,GL_RGB,GL_UNSIGNED_BYTE,mem_);
   glFlush();
#endif
}

Vl< Vl<ColRef> >  Rec::fold()const
{
   L* p=mem_;
   Z w1=w(),h1=h();
```

```
   Vl<Vl<ColRef> > res(h1,Vl<ColRef>(w1));
   for (Z i=h1-1;i>=0;--i){
      for (Z j=0;j<w1;++j){
         L r=*p++;
         L g=*p++;
         L b=*p++;
         res[i][j]=ColRef(r,g,b);
      }
   }
   return res;
}

Rec::Rec(Vl< Vl<ColRef> > const& bh)
{
   Z h1=bh.dim();
   Z w1=bh[0].dim();
   xyxy r1(xy(0,0),w1,h1);
   rec_=r1&Viewport::win();
   Z height=rec_.h();
   Z width=rec_.w();
   n_=height*width*3;
   mem_=new L[n_];
   L* p=mem_;
   for (Z i=height-1;i>=0;i--){
      for (Z j=0;j<width;j++){
         ColRef val=bh[i][j];
         *p++=val[1];
         *p++=val[2];
         *p++=val[3];
       }
   }
}

//////////////////// class ColRef ////////////////////////

Z CpmGraphics::ColRef::lowerLimit=0;
Z CpmGraphics::ColRef::upperLimit=255;
   // no surprises

bool ColRef::prnOn(ostream& str)const
{
   cpmwt("ColRef");
   cpmp(r);
   cpmp(g);
   cpmp(b);
   return true;
}

bool ColRef::scanFrom(istream& str)
{
```

```
   cpms(r);
   cpms(g);
   cpms(b);
   return true;
}

//////////////////// class rgb ////////////////////////////

Z CpmGraphics::rgb::mixingRule(Z z1, Z z2, R w1, R w2, R gamma)
{
   if (gamma==0)
      return ColRef::upperLimit;
   else if (gamma==1)
      return Z(w1*z1+w2*z2);
   else if (gamma==2)
      return Z(sqrt(w1*z1*z1+w2*z2*z2));
   else if (0<gamma && gamma<10){
      R p1=w1*::pow((R)z1,gamma);
      R p2=w2*::pow((R)z2,gamma);
      R gammaInv=1./gamma;
      return Z(::pow(p1+p2,gammaInv));
   }
   else
      return (z1>=z2 ? z1 : z2);
}

rgb CpmGraphics::rgb::mix(rgb c1, rgb c2, R w1, R w2, R gamma)
{
   Z a=mixingRule(c1.r,c2.r,w1,w2,gamma);
   Z b=mixingRule(c1.g,c2.g,w1,w2,gamma);
   Z c=mixingRule(c1.b,c2.b,w1,w2,gamma);
   rgb res(a,b,c);
   res.normalize();
   return res;
}

////////////////////////// class Viewport ////////////////////////////

// common initializations of static variables

ColRef CpmGraphics::Viewport::colText=ColRef();
R CpmGraphics::Viewport::gamma=2.3;
//R CpmGraphics::Viewport::gamma=1;
// best guess from a Kodak tool GAMMA.TIF for my office monitor
// is 2.3, also for my TFT-Monitor this seems to be the best
xyxy CpmGraphics::Viewport::window;
xyxy CpmGraphics::Viewport::fullWindow;
Z CpmGraphics::Viewport::size=1;
Z CpmGraphics::Viewport::rank=1;
```

```
#if !defined(CPM_NOGRAPHICS)
using CpmGraphics::Font;
bool CpmGraphics::Viewport::initialized=false;
Z CpmGraphics::Viewport::applWindowWidth=0;
Z CpmGraphics::Viewport::applWindowHeight=0;
Z CpmGraphics::Viewport::fullWindowHeight=0;

namespace{

Z getX()
{
   GLint x4[4];
   glGetIntegerv(GL_CURRENT_RASTER_POSITION,x4);
   return x4[0];
}

} // namespace

CpmGraphics::Font::Font(Z type)
{
   if (type==1){
      name=GLUT_BITMAP_TIMES_ROMAN_10;
      h=10;
   }
   else if (type==2){
      name=GLUT_BITMAP_HELVETICA_10;
      h=10;
   }
   else if (type==3){
      name=GLUT_BITMAP_HELVETICA_12;
      h=12;
   }
   else if (type==4){
      name=GLUT_BITMAP_8_BY_13;
      h=13;
   }
   else if (type==5){
      name=GLUT_BITMAP_9_BY_15;
      h=15;
   }
   else if (type==6){ // GLUT_BITMAP_HELVETICA_18 is very poorly designed
      // or implemented and should not be used
      // name=GLUT_BITMAP_HELVETICA_18;
      // h=18;
      name=GLUT_BITMAP_9_BY_15;
      h=15;
   }
   else {
      name=GLUT_BITMAP_TIMES_ROMAN_24;
      h=24;
```

```
   }
   d=CpmRoot::round(h*0.2);
   mw=CpmRoot::round(h*0.6); // as in 9 by 15 font
      // only an approximation - needed since glutBitmapWidth(...)
      // couldn't be made working so far
}

CpmGraphics::Font::Font(R fh)
{
   if (fh<=10.0){
      name=GLUT_BITMAP_HELVETICA_10;
      h=10;
   }
   else if (fh<=11.0){
      name=GLUT_BITMAP_TIMES_ROMAN_10;
      h=10;
   }
   else if (fh<=12.5){
      name=GLUT_BITMAP_HELVETICA_12;
      h=12;
   }
   else if (fh<=14.0){
      name=GLUT_BITMAP_8_BY_13;
      h=13;
   }
   else if (fh<=16.5){
      name=GLUT_BITMAP_9_BY_15;
      h=15;
   }
   else if (fh<=21.0){ // see comment to Font(6)
   //   name=GLUT_BITMAP_HELVETICA_18;
    //  h=18;
      name=GLUT_BITMAP_9_BY_15; // experiment due to L'
      h=15;
   }
   else if (fh<=24.5) {
      name=GLUT_BITMAP_TIMES_ROMAN_24;
      h=24;
   }
   else{
      name=GLUT_BITMAP_TIMES_ROMAN_24;
      h=24;
   }
   d=CpmRoot::round(h*0.2);
   mw=CpmRoot::round(h*0.6); // as in 9 by 15 font
      // only an approximation - needed since glutBitmapWidth(...)
      // couldn't be made working so far
}

// basic Cpm graphics library should be independent from CpmSystem and
```

```
// thus not to have to include cpmsystem.h

xyxy CpmGraphics::Viewport::seg1;
xyxy CpmGraphics::Viewport::seg2;
xyxy CpmGraphics::Viewport::seg3;
xyxy CpmGraphics::Viewport::seg4;

// background colors of the panes on the status bar
ColRef CpmGraphics::Viewport::cs1(L(255),L(255),L(50));
ColRef CpmGraphics::Viewport::cs2(L(210),L(255),L(110));
ColRef CpmGraphics::Viewport::cs3(L(180),L(180),L(190));
ColRef CpmGraphics::Viewport::cs4(L(165),L(165),L(255));

void Viewport::fill(xyxy r, ColRef cr)
{
   glColor3f(cr.glR(),cr.glG(),cr.glB());
   glRecti(r.x1,r.y1,r.x2+1,r.y2+1);
   glFlush();
}

Viewport::Viewport(Z w, Z h, const Word& title)
{
      if (initialized) return;
      Word loc("Viewport::Viewport(Y,Y,Word)");
      size=CpmMPI::Cpm_com.getSize();
      rank=CpmMPI::Cpm_com.getRank();
      ColRef crBac(L(230),L(255),L(180));
      applWindowWidth=w;
      fullWindowHeight=h;
      fullWindow=xyxy(xy(0,0),applWindowWidth,fullWindowHeight);
      applWindowHeight=fullWindowHeight-heightStatusBar;
      window=xyxy(xy(0,0),applWindowWidth,applWindowHeight);
      if (rank==1){
         Word mes="OpenGL window got for applWindowWidth = ";
         mes&=cpm(applWindowWidth);
         mes&=" , applWindowHeight= ";
         mes&=cpm(applWindowHeight);
         Word mes2="Platform is "&Platform;
         cout<<mes<<endl;
         cout<<mes2<<endl;
      }
      R cp1=Message::pane1(); // from cpmsystemdependencies
      R cp2=Message::pane2();
      R cp3=Message::pane3();
      R cp4=Message::pane4();
      R sumPanes=cp1+cp2+cp3+cp4;
      if (sumPanes==0){
         cp1=2;
         cp2=2;
         cp3=1;
```

```
         cp4=1;
         sumPanes=cp1+cp2+cp3+cp4;
      }
      R awsp=applWindowWidth/sumPanes;
      Z w1=cpmround(awsp*cp1);
      Z w2=cpmround(awsp*cp2);
      Z w3=cpmround(awsp*cp3);
      Z w4=applWindowWidth-w1-w2-w3;
      Z yL=applWindowHeight;
      Z xL=0;
      seg1=xyxy(xy(xL,yL),w1,heightStatusBar);
      xL+=w1;
      seg2=xyxy(xy(xL,yL),w2,heightStatusBar);
      xL+=w2;
      seg3=xyxy(xy(xL,yL),w3,heightStatusBar);
      xL+=w3;
      seg4=xyxy(xy(xL,yL),w4,heightStatusBar);
      initialized=true;
      xy p0;
      xyxy all(p0,applWindowWidth,applWindowHeight);
      fill(all,crBac);
      R fontHeight=14;
      Font font(fontHeight);
      Z tx=font.getWidth();
      Z ty=font.getHeight();
      ColRef crText(L(0),L(0),L(255));
      textLine(tx,ty,title,crText,fontHeight);
}

void Viewport::onStatusBar(const Word& w, Z i)
{
   static Z th=cpmround(0.75*heightStatusBar);
   R fH=12.0;
   static Font font(fH); // Helvetica 12
   if (!isInitialized() ) return;
   ColRef cr;
   xyxy seg;
   if (i==1){
      cr=cs1;
      seg=seg1;
   }
   else if (i==2){
      cr=cs2;
      seg=seg2;
   }
   else if (i==3){
      cr=cs3;
      seg=seg3;
   }
    else if (i==4){
```

```
      cr=cs4;
      seg=seg4;
   }
   fill(seg,cr);
   Z shift=2;
   placeWord(seg.x1+shift, seg.y1+th, w, font, seg.w()-shift);
 }

void Viewport::clrPan()
{
   fill(seg1,cs1);
   fill(seg2,cs2);
   fill(seg3,cs3);
   fill(seg4,cs4);
}

void Viewport::placeWord(Z tx, Z ty, const Word& w, const Font& f,
   Z wMax)
{
   glColor3f(colText.glR(),colText.glG(),colText.glB());
   void* FONT=f.getName();
   Z x=tx,y=ty;
   Z xLim=tx+wMax;
   Z xM=f.getWidth();
   xLim-=xM;
   Z n=w.dim();
   glRasterPos2i(x,y);
   for (Z i=1;i<=n;++i){
      glutBitmapCharacter(FONT,w[i]);
      x=getX();
      if (x>xLim) break;
   }
   glFlush();
}

xyxy Viewport::textLine(Z tx, Z ty, const Word& w,
      ColRef crText, R h)
      // writing w on a line previously filledwith crBac
{
   glColor3f(colText.glR(),colText.glG(),colText.glB());
   Font f(h);
   void* FONT=f.getName();
   Z x=tx,y=ty;
   Z xM=f.getWidth();
   Z yM=f.getHeight();
   Z decender=cpmround(0.25*yM);
   Z i,n=w.dim();
   Z more=2; // makes the background rectangle a bit larger than the text
   glColor3f(crText.glR(),crText.glG(),crText.glB());
   glRasterPos2i(x,y);
```

```
   for (i=1;i<=n;++i){
       glutBitmapCharacter(FONT,w[i]);
   }
   Z txF=getX();
   glFlush();
   return xyxy(tx-more,ty-yM+decender-more,txF+more,ty+decender+more);
 }

xyxy Viewport::addText(Z x, Z y, Word const& line,
   R fontHeight, Z fontNumber)
{
   fontNumber; // not used
   return textLine(x,y,line,colText,fontHeight);
}

#else
// e.g. if CPM_NOGRAPHICS is defined
// trivial implementation; since then graphics
// works through a RAM-based data structure Img24

bool CpmGraphics::Viewport::initialized=true;

Viewport::Viewport(Z w, Z h, const Word& title){}
void Viewport::onStatusBar(const Word& w, Z i){}

Z CpmGraphics::Viewport::applWindowWidth=10;
Z CpmGraphics::Viewport::applWindowHeight=10;
Z CpmGraphics::Viewport::fullWindowHeight=10;

void CpmGraphics::Viewport::placeWord(Z tx, Z ty, const Word& w,
   const Font& f, Z wMax){tx;ty;w;f;wMax;}

void CpmGraphics::Viewport::fill(xyxy r, ColRef cr){r;cr;}

xyxy Viewport::textLine(Z tx, Z ty, const Word& w,
      ColRef crText, R h)
{
   tx;ty;w;crText;h;
   return xyxy();
}

xyxy Viewport::addText(Z x, Z y, Word const& text, R size, Z fontNumber)
{
   x;y;text;size;fontNumber;
   return xyxy();
}

void Viewport::clrPan()
{}
```

```
#endif
```

223 cpmvl.h

```
//? cpmvl.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_VL_H_
#define CPM_VL_H_
/*

Purpose: Basic array class with valid indexing starting at 0.

   Indexes restricted to C-convention for present class Vl<T>.
   Class V<T> will have my standard convention of starting with 1. This
   was motivated by the difficulties in transforming my old sli.cpp to
   modern style. There I used native C arrays ('for efficiency' ) and it
   turned out to be very tedious and error prone to change the index
   range everywhere in this large program. Same situation might come
   again and again when trying to learn from code written for STL vectors
   or native C arrays. So it is helpful to have a safe and efficient
   replacement for native C arrays.

Credit: Modified and extended from Andrew Koenig: Ruminations on C++,
AT&T Ch. 7. Authoritative and enlightening treatment. However,
compared to the present version, some essentials are missing in
Koenig's book. Also two misprints caused trouble.

*/
#include <cpmfl.h>
   // small and efficient function template with minimum
   // infra-structure requirements
#include <cpmzinterval.h>
   // includes cpmsystem.h
   // With using arrays something may go wrong and so messaging
   // capability is indispensible
#include <cpmmacros.h>
#include <vector> // needed for the additions of 2006-11-06

//////////////////////////////// class Vl<> ////////////////////////////////
// l stands for 'lean'.
// Array with index check, reference counting, and copy on write, and
// 'value semantics' (as opposed to 'pointer semantics')
// Generalized from Koenig's class Handle p. 72-73.
// Indexes run from 0 to dim-1. This helps to transform native C-code
// (not following the convention of the Numerical Receipes) into the
// present framework. Also helps to experiment with replacing 'Vl' by
// 'vector' of the STL. Vl< Vl<ColRef> > is the type of bitmap data
// in my graphical workhorse class Img24. This can be considered a proof
// for good performance of the class. Efficient conversion functions from
// and to std::vector<> are now provided.
```

```
// There is a nice way to iterate over the whole index range without
// mentioning the bounds as 0 and dim-1
//    Vl<T> v=....;
//    for (Z i=v.b();i<=v.e();++i) v[i]=...;
// for V<T> v=...; this would be the same.
// This allows new C+- code be written that is the same independent
// of the decision whether to use Vl or V. This could once lead to
// having a single vector template. I'm already on this track by having
// implemented the IO-interface and the order-interface already here.
// Notice that this is done by assuming the functions prnOn, scanFrom,
// and com are defined in CpmRoot for type T. Nevertheless Vl<T> can be
// instantiated for any T if order operations and IO operations are
// never called for any instance.

namespace CpmArrays{

   using namespace CpmStd;

   using CpmRoot::Word;
   using CpmSystem::Error;
   using CpmFunctions::F;
   using CpmFunctions::F1;

// some infrastructure

   enum Outside { DEFAULT, CYCLIC, CONSTANT};
      //: outside
      // Controls the meaning of 'out of range indexes'.
      // DEFAULT: definition as the default value associated with
      // the type under consideration
      // CYCLIC: setting the meaning of out of range indexes by
      // cyclic repetition of value
      // CONSTANT: continuation as constant from the nearest value

   extern Z dimMax;
      // If the dimension of an array was either the result of a
      // calculation or of reading from a file, then the result may be
      // off the programmer's intent by orders of magnitude if something
      // went wrong. So it is helpful to exclude unnaturally large arrays
      // from becoming allocated.

   extern bool ranChc;
      //: range check
      // Initialized as true.

   extern bool ranChcAlw;
      //: range check always
      // If this is true, all access operators even those the name
      // of which suggests the opposite get checked --- with poor
      // diagnostics, though.
```

```
      // Initialized as true, since access to non-allocated memory,
      // as a rule, causes disaster. I had to discover in 2008-03-03
      // that such a case happened in the workhorse function
      // CpmGraphics::Graph::mark(Vl< Vl<C> >,...)
      // on a regular basis, due to an seemingly safe but actually
      // un-safe usage of V<>::cui().

   void setDimMax(Z n);
      // sets dimMax=n unless n<0. In this case error with message

   Z safeDim(Z n);
      // returns n for 0<=n<=dimMax, otherwise error with message

template <class T>
   // We assume that T provides (explicitely or implicitely)
   // copy constructor, and assignement or that T is a built-in type.
   // If the index operator [] is to be used and the variable ranChc
   // (which is initialized as 'true') was not set to 'false' an out
   // of range error will result in a runtime error which will be
   // documented on cpmcerr.txt. See ranChcAlw for an additional control.
   // The error message is particularly explicit (indicating the type of
   // T) when also the macro CPM_NAMEOF is defined. If this is the case,
   // the type T needs to define the member function
   // CpmRoot::Word nameOf()const. For all Cpm-classes this is the case
   // and for user classes, there should be no difficulty in adding such
   // a function. If one wants to make use of the function declared
   // by the declaration macros CPM_ORDER type T needs to define the
   // member function  CpmRoot::Z com(T const&)const;
   // If one wants to make use of the functions declared by the
   // declaration macro CPM_IO, type T needs to define the member
   // functions bool prnOn(ostream&)const and bool scanFrom(istream&).
   // These requirements do not apply to basic types: for T=Z,R,Rh,L one
   // may use all functions of Vl<T> without further requirements.
   // This template is not intended to serve as a base class for
   // derivations: It has no virtual functions and no protected members.

class Vl{ // vector template, indexing starts with 0
   // Although implementation details are inspired from handle classes,
   // the Vl class is a value array and not a handle class.
   // All allocations made with new T[] all de-allocations are  delete[]
   // No 16 bit memory model features integrated, although this would not
   // be difficult. All data are private, so the only access to data in
   // derived classes is over the public functions of the class. All
   // these incorporate reference counting internally where needed (only
   // if the preprocessing directive CPM_USECOUNT is active). The user of
   // the class can't see this and has not to be aware of this.
   // In building on this class, I decided not to choose inheritence but
   // using Vl as a data type of components.

   typedef Vl<T> Type;
```

```
public:
   CPM_IO
   CPM_ORDER

explicit   Vl(Z n=0) ;
      // Has n components. Gives an error for n<0 and n>dimMax.
      // The components are initialized by the default constructor
      // of T if T is a class for which such a constructor is defined
      // (explicitely or implicitely).
      // If T is a built-in type, initialization is done as 0.
      // See BS3, p. 131 for initialization of built-in types via
      // formal constructor calls.

explicit Vl(std::vector<T> const& v);
      // Construction from a standard library vector. This is useful
      // for interaction with the standard containers (which don't
      // implement reference counting)

   Vl(Z n, T const& t) ;
      // Has n components all initialized as t

   Vl(Z n, F<Z,T> const& f);
      // construction from a function. Of course,
      // Vl<T> v(n,f);
      // implies v[i]==f(i) for all valid indexes i of v.

   Vl(Vl<T> const& h) ;
      // copy constructor

   ~Vl();
      // destructor

   Vl<T>& operator=(Vl<T> const& h);
      // assignment

   Z dim()const { return sz_;}
      //: dimension
      // Returns the number of components of the vector *this

   Z size()const { return sz_;}
      //: size
      // for uniformity with STL

   std::vector<T> std()const;
      //: standard
      // returns a STL-vector

   Word nameOf()const;
      //: name of
```

```
      // returns a name of the type

   IvZ dom()const { return IvZ(0,sz_-1);}
      //: domain
      // Notice that with the array *this there is the
      // function f: {0,...,sz_-1}-->T, i|-->*(p_+i)
      // associated in a natural manner.
      // For this function, dom() is just the domain.
      // The understanding of arrays as functions with domains of type
      // IvZ seems to be a good guide for defining some of the member
      // functions in a more natural manner by using arguments of type
      // IvZ. Present examples are the functions fa_ and valOn.

   bool isVoid()const { return sz_==0;}
      //: is void
      //  short answer on whether the dimension is zero

   bool valInd(Z i)const{ return i>=0 && i<sz_;}
      //: valid index
      // returns the validity of i as an index of *this

// constant access functions

   T const& operator[](Z i)const;
      // getting to the value of a component for 'read', e.g.
      // T t=(*this)[i];

   T const& cui(Z i)const;
      // component (with) unchecked index
      // getting to the value of a component for 'read', e.g.
      // T t=cui(i);
      // It helps to write efficient code in classes which use Vl<>-typed
      // data members.
      // Notice that writing loops by using b() and e() to define
      // the range is much safer than using limits like 0 and dim().
      // index range check is missing only if variable ranChcAlw was
      // changed to 'false'.

   T const& cyc(Z i)const;
      //: cyclic
      // getting to the value of a component for 'read', e.g.
      // T t=cyc(i);
      // Here i is understood as cyclic (i.e. i modulo sz_). So no value
      // of the index has to be considered 'out of range' and for i
      // 'in range' cyc(i)==(*this)[i]

   T const& con(Z i)const;
      //: constant
      // Same logic as cyc() but with constant continuation
      // instead of cyclic.
```

```
   T operator()(Z i, Outside mode=DEFAULT)const;
      // Read access defined for all i.
      // By this definition, a vector becomes a mapping from Z to T.
      // For i's outside the proper range,
      // the default T is returned for mode==DEFAULT or if
      // sz_==0. If mode==CYCLIC cyc(i) gets returned.
      // For mode==CONSTANT we return p_[0] for i<=0 and p_[sz_-1]
      // for i>=sz_.
      // Notice that the return value is not a reference in
      // accordance with function behavior.

   T const& read(Z i, Outside mode=DEFAULT)const;
      //: read
      // Reading components in an efficient (as &'s), safe and flexible
      // manner.
      // See T operator()(Z i, Outside mode=DEFAULT)const; for the
      // meaning of the second argument.

   F<Z,T> fnc(Outside meth=DEFAULT)const;
      // returns the function Z --> T, i |--> (*this)(i,meth)

// non-constant access functions
   T& operator[](Z i);

   T& cui(Z i);
      // setting to the value of a component
      // T t=...;
      // Vl<T> x=...;
      // x.cui(i)=t;

   T& cyc(Z i);
      // setting to the value of a component
      // T t=...;
      // Vl<T> x=...;
      // x.cyc(i)=t; meaning of i is modulo sz_. So i is never out of
      // range
   T& con(Z i);
      // Same logic as cyc() but with constant continuation
      // instead of cyclic.

// accessing the first and the last element
   T& first(){ return fir();}
      // old name
   T const& first()const{ return fir();}
      // old name

   T& fir();
      //: first
   T const& fir()const;
```

```
      //: first
   T& last();
      //: last
   T const& last()const;
      //: last

// getting the first and the last index
   Z b()const { return 0;}
      //: begin ; returns the first valid index, here 0

   Z e()const { return sz_-1;}
      //: end ; returns the last valid index, here sz_-1
      // allows to write loops over all components as
      // for (Z i=v.b();i<=v.e();i++) ... v[i] ...;
      // Note that this is safe also v.dim()==0, since then
      // v[e()] will never be called. Here one could replace
      // v[i] by v.cui(i) without real danger.

   void set_(T const& t);
      // non-constant function which sets all components of (*this)
      // equal to t

   Vl<T> set(Z i, T const& t)const;
      // returns a vector that originates from *this by setting the
      // value of component i.
      // Regular behaviour:
      // if we say
      // Z i=...;
      // T t= ...;
      // Vl<T> v=...;
      // v=v.set(i,t);
      // then the i-th component (see eli() ) of v will get the value t
      // unless i<1.
      // If i is a value for which (*this)[i-1] is not yet defined,
      // the vector becomes enlarged to the necessary size and all
      // not specified components initialized with the default
      // constructor of T

   T in_(T const& t, bool reversed=false);
      // This operations treats *this as a shift register:
      // All components get shifted by one position 'to the right' and
      // the total length (dim) of the vector remains the same. So what
      // was the last component prior to the operation has to be removed
      // from the vector, in order to not wasting information, this
      // removed component will shine up as the return value of the
      // function. After the operation, the first component of the vector
      // will be t.
      // If reversed==true, t gets inputted at the end, and all shift
      // operations go 'to the left'.
```

```
// building new objects from given ones (generative methods)

// concatenating vectors and appending components. These operations have
// always O(sz_) complexity. The corresponding combined assignements are
// less direct to implement are not considered useful in the present
// context (they would not be more efficient than the friend versions,
// since new memory has to be allocated anyway).

   Vl<T> append(T const& t)const;
      // returns a vector which is obtained from *this by appending t
      // as the last component
      // notice also the prepend functions to be introduced
      // after the insert-functions (in order to have the inline
      // definition available).

   Vl<T> append(Vl<T> const& h)const;
      // returns a vector which is obtained from *this by appending h
      // at the back end

   void push_back(T const& t) { *this=append(t);}
      // for uniformity with STL

   void operator<<(T const& t) { *this=append(t);}
      // appending an element in Ruby-style

   void operator<<(Vl<T> const& h) { *this=append(h);}
      // appending an array in Ruby-style

   Vl<T>& operator&=(T const& t) { return *this=append(t);}
   Vl<T>& operator&=(Vl<T> const& h) { return *this=append(h);}
   Vl<T> operator&(T const& t)const { return append(t);}
   Vl<T> operator&(Vl<T> const& h)const { return append(h);}
      // appending in my favorite style

   Vl<IvZ> valOn(F<T,bool> const& f)const;
      //: valid on
      // retuns the array of those sub-intervals of the
      // whole indexing interval dom() on which the function
      // dom()-->bool, i|-->f((*this)[i]) yields true.
      // Notice that the result res \in Vl<IvZ> is a convenient
      // representation of a subset of dom(). Considering
      // *this as a function g: dom()-->T, then the function
      // h:=g&f is of type dom()-->bool and res, as a subset
      // of dom(), is h^-1({true}).

// resizing

   Vl<T> resize(Z newDim)const;
      // Returns a vector res such that res.dim()=newDim. If newDim is
      // smaller than dim(), the end of *this will be cut away. If newDim
```

```
      // is larger, T()'s will be added.
      // For newDim<0 the action is as if newDim==0.

   Vl<T> cut(Z i)const{ return resize(sz_-i);}
      // returned is a V<T> which results from *this by removing i
      // components from the end. For exotic values of i, see code
      // and explanation of resize.

// elimination and insertion

   Vl<T> eli(Z i, Z nEli=1)const;
      // eli stands for eliminate
      // returned is a vector which is obtained from *this by
      // eliminating the i-th component and the nEli-1 following ones
      // (thus nEli components are removed) and shifting all later
      // components (if there are such components left) 'to the left' to
      // close the gap. The phrase 'i-th component' refers to natural
      // counting (thus p_[i-1] is the i-th component of *this).

   Vl<T> eli(IvZ const& iv)const;
      // eliminating all components which belong to iv. No exceptions
      // can happen!

   Vl<T> eliFirst(Z nEli=1)const { return eli(1,nEli);}
      // returned is a vector which is obtained from *this by
      // eliminating the nEli first components. If no
      // argument is provided, actually the first component
      // gets eliminated

   Vl<T> eliLast()const { return eli(sz_,1);}
      // returned is a vector which is obtained from *this by
      // eliminating the last component.

   Vl<T> eli1(Vl<T>& h, Z i)const;
      // We return a vector which results from *this by eliminating
      // h.dim() components starting at the i'th component (for i<1,
      // i==1 is understood). After the call h will hold the
      // eliminated components in due order (and no more - even if h was
      // longer before).
      // The function name ends in '1' to indicate that the first
      // argument is a non-constant reference, used for communication
      // of a part of the result.

   Vl<T> select(const Vl<bool>& s)const;
      // returned is a list which is obtained from *this by eliminating
      // all components (*this)[i] for which s[i] is defined and
      // statisfies s[i]==false. Beside of this removal, the order of the
      // components in *this is retained. So if s is defined
      // by a condition s[i]=Condition((*this)[i]), for the vector
      // component, this condition has to express a property we like to
```

```
      // have fulfilled for the result-vector of the select-operation.
      // 's expresses the favorable condition' and  n o t  the one to be
      // eliminated. Notice, that the select operation makes sense for
      // all values of s.dim().

   Vl<T> ins(Z i, T const& t)const;
      // ins stands for insert
      // returned is a vector which is obtained from *this by
      // inserting t as the i-th component and shifting all later
      // components 'to the right' to give room for t.
      // The phrase 'i-th component' refers to natural counting (thus
      // p_[i-1] is the i-th component of *this).

   Vl<T> ins(Z i, Vl<T> const& h)const ;
      // returned is a vector which is obtained from *this by
      // inserting h as the i-th and following component,
      // and shifting all later components of *this
      // 'to the right' to give room for h.
      // The phrase 'i-th component' refers to natural counting (thus
      // p_[i-1] is the i-th component of *this).

   Vl<T> prepend(T const& t)const
      // returns a vector which is obtained from *this by appending t
      // as the first component
      { return ins(1,t);}

   Vl<T> prepend(Vl<T> const& h)const
      // returns a vector which is obtained from *this by appending h
      // at the front end
      { return ins(1,h);}

   Vl<T> rev()const;
      // returned is the reversed vector (indexing in the opposite
      // direction).

   template <class Y>
   Vl<Y> operator()(F<T,Y> const& f)const { return Vl<Y>(sz_,fnc()&f);}
      // generating arrays of different type by a type changing function

// diagnostics
   void show(ostream& out)const ;
      // diagnostic messages;

// defining generative laws and mutating laws for various expressions by
// function pointers. The idea behind is, that for T which allows
// particular operations, we can define those without using iteration via
// a operator[] since these all are defined directly in terms of
// pointers.
// See implementation of +=, -=, ... in Va<T> how this works
```

```
T fAcc1(T (*f)(T const&), void (*acc)(T&, T const&))const;
   // returns an accumulated value of all values f(c), where c
   // are the components of *this. Accumulation is defined by
   // the argument acc:
   // T res=T(); 'for all components c' acc(res,c)

T fAcc2(T (*f)(T const&, T const&), void (*acc)(T&, T const&),
   Vl<T> const& h )const ;
   // returns an accumulated value of all values f(c,ch), where c and
   // ch are the components of *this and h respectively. Accumulation
   // is defined by the argument acc:
   // T res; 'for all components c' acc(res,c)
   // useful in defining scalar products

Vl<T> fa(T (*f)(T const&))const ;
   // returns a vector defined by replacing the components c of *this
   // by f(c)

Vl<T> fb(T (*f)(T const&, T const&), T const& t)const;
   // returns a vector defined by replacing the components c of *this
   // by f(c,t)

void fc_(T (*f)(T const&, T const&), T const& t) ;
   // replaces *this by a vector defined by replacing the components c
   // of *this by f(c,t)

void fd(T (*f)(T const&, T const&), T& t)const;
   // replaces t by f(c,t) for each component c
   // If, for instance, f(c,t)=t+c*c we replace t by t+sum of c*c

Vl<T> fe(T (*f)(T const&, T const&), Vl<T> const& h)const;
   // returns a vector defined by replacing the components c of *this
   // by f(c,c') where c' are the components of h.
   // Error if dim()!=h.dim()

Vl<T> fe2(T (*f)(T const&, T const&), Vl<T> const& h)const;
   // Very similar to fe. However, the dimension of the result
   // is the maximum of dim() and h.dim(). Non-existing components
   // of any of the operands are replaced by T().

void ff_(T (*f)(T const&, T const&), Vl<T> const& h, Z i=0);
   // Replaces the components of *this starting from (*this)[i]
   // by f(c,c') where c' are the components of h. Thus for i=0
   // and h.dim()>=sz_ the whole array *this is affected.

void fg_(T (*f)(T const&, T const&, T const&),
   Vl<T> const& h, T const& t);
   // replaces the components c of *this
   // by f(c,c',t) where c' are the components of h
```

```
   void fa_(F<T,T> const& f, IvZ iv);
      // replaces the components c of *this with index in iv
      // by f(c); modern form of fa. Notice that IvZ is a class
      // that describes contiguous sets of integers.

   void fh_(T const& w1, T const& w2, T const& w3, Outside mode);
      // replaces the component c[i] of *this by
      // w1*read(i-1,mode)+w2*read(i,mode)+w3*read(i+1,mode)

private:
// member functions
   Vl(Z n, T* p1):sz_(n),p_(p1){}
      // p1 has to be created with new; since this is prone to misuse
      // this function has to be private.

   void cow(void);
      //: copy on write ; The first '*this-changing' statement in the
      // body of a non-constant member function has to be
      //   cow();

   T* copy()const;
      // returns a pointer to a storage area that contains
      // a copy of those of *this

// data members

   Z sz_;
      // number of components; sz_>=0

   T* p_;
      // pointer to beginning of the array. Should be accessible to
      // derived classes for enabling fast component operations.

#if defined(CPM_USECOUNT)
   UseCount u_;
#endif

// static data
   static const T def_;
      // allows read function to return references
// static functions
   static T fCyc(Z const& i, Vl<T> const& v) { return v(i,CYCLIC);}
   static T fCon(Z const& i, Vl<T> const& v) { return v(i,CONSTANT);}
   static T fDef(Z const& i, Vl<T> const& v) { return v(i,DEFAULT);}
   static size_t sit(Z i) { return i<0 ? (size_t)(-i) : (size_t)i;}
      // instead of new T[i], I now write new T[sit(i)] so that
      // allocation is always fed with a parameter which fits the system
      // (in 64 bit systems, size_t may be 64 bit wide). I assume that
      // size_t is known due to inclusion of <vector> und takes the
      // bit-width of the machine properly into account.
```

```
};

//////// using Vl for some special template arguments/////////////////

Vl<Z> IvZtoVlofZ(IvZ const& iv);
   //: Iv to Vl of Z
   // returns the Z's that make up the interval iv
   // as the components of an ordered array (increasing
   // order, of course)

Vl<Z> VlofIvZtoVlofZ(Vl<IvZ> const& viv);
   //: Vl of IvZ to Vl of Z
   // appends the results from applying the previous functions
   // to the viv[i] according to the obvious code
   //  { Vl<Z> res(0); Z n=viv.dim();
   //  for (Z i=0;i<n;++i) res&=IvZtoVlofZ(viv[i]); return res;}

Vl<Word> comLineLean(int argc, char* argv[]);
   //: command line lean
   // converts the C-traditional command-line argument into a C+- array.
   // Tool for implementing the more canonical function
   // V<Word> comLine(int argc, char* argv[])

//////////////////////// Implementation ////////////////////////////////
template <class T>
const T Vl<T>::def_=T();

template <class T>
inline   Vl<T>::~Vl(){
#if defined(CPM_USECOUNT)
   if (u_.only()){ delete[] p_;}
#else
   delete[] p_;
#endif
}

template <class T>
Word Vl<T>::nameOf()const{
   T t;
   Word nt=CpmRoot::Name<T>()(t);
   Word wi="Vl<";
   return wi&nt&">";
}

template <class T>
inline T const& Vl<T>::operator[](Z i)const
{
   if (ranChc){
      if (i<0 || i>=sz_){
         cpmerror(nameOf()&"::operator[]: read index out of  range: i= "&
```

```
         cpm(i)&" dim= "&cpm(sz_));
      }
   }
   return p_[i];
}

template <class T>
inline T& Vl<T>::operator[](Z i)
{
   if (ranChc){
      if (i<0 || i>=sz_){
         cpmerror(nameOf()&"::operator[]: write index out of  range: i= "&
         cpm(i)&" dim= "&cpm(sz_));
      }
   }
#if defined(CPM_USECOUNT)
   cow(); // copy on write since operator is not const
#endif
   return p_[i];
}

// using intentionally previously defined []-indexing
// for common messaging and safeness.
template <class T>
T const& Vl<T>::fir()const{ return (*this)[0];}

template <class T>
T& Vl<T>::fir(){ return (*this)[0];}

template <class T>
T const& Vl<T>::last()const{ return (*this)[sz_-1];}

template <class T>
T& Vl<T>::last(){ return (*this)[sz_-1];}

template <class T>
inline T const& Vl<T>::cui(Z i)const
{
   if (ranChcAlw){
      if (i<0 || i>=sz_) cpmerror("index out of range");
   }
   return p_[i];
}

template <class T>
inline T& Vl<T>::cui(Z i)
{
   if (ranChcAlw){
      if (i<0 || i>=sz_) cpmerror("index out of range");
   }
```

```
#if defined(CPM_USECOUNT) // don't call a function if not needed
   // even if its implementation is trivial
   cow();
#endif
   return p_[i];
}

template <class T>
T const& Vl<T>::cyc(Z i)const
{
   if (i<0){
      while (i<0){ i+=sz_;}
      return p_[i];
   }
   else return p_[i%sz_];
}

template <class T>
T& Vl<T>::cyc(Z i)
{
#if defined(CPM_USECOUNT)
   cow();
#endif
   if (sz_==0){
      cpmerror("Vl<T>::cyc(Z i): i="&cpm(i)&" is no valid index in void\
       array");
      return p_[0]; // never reached
   }
   if (i<0){
      while (i<0){ i+=sz_;}
      return p_[i];
   }
   else return p_[i%sz_];
}

template <class T>
T const& Vl<T>::con(Z i)const
{
   if (sz_==0) return def_;
   if (i<0) return p_[0];
   else if (i>=sz_) return p_[sz_-1];
   else return p_[i];
}

template <class T>
T& Vl<T>::con(Z i)
{
   if (sz_==0){
      cpmerror("Vl<T>::con(Z i): i="&cpm(i)&
         " is no valid index in void array");
```

```
      return p_[0]; // never happens
   }
   cow();
   if (i<0) return p_[0];
   else if (i>=sz_) return p_[sz_-1];
   else return p_[i];
}

template <class T>
T const& Vl<T>::read(Z i, Outside mode)const
{
   if (i>=0 && i<sz_) return p_[i];
   else{
      if (mode==DEFAULT) return def_; // reference to it can be returned
      else if (mode==CYCLIC) return cyc(i);
      else return i<0 ? p_[0] : p_[sz_-1];
   }
}

template <class T>
T* Vl<T>::copy()const
{
   T* itp=p_;
   T* q=new T[sit(sz_)];
   T* it=q;
   Z i=sz_;
   while (i--) *it++=*itp++;
   return q;
}

template <class T>
T Vl<T>::operator()(Z i, Outside mode)const
{
   if (sz_==0) return T();
   if (i>=0 && i<sz_) return p_[i];
   if (mode==DEFAULT) return T();
   else if (mode==CYCLIC) return cyc(i);
   else return con(i);
}

template <class T>
F<Z,T> Vl<T>::fnc(Outside mode)const
{
   if (mode==DEFAULT) return F1<Z,Vl<T>,T>(*this)(fDef);
   else if (mode==CYCLIC) return F1<Z,Vl<T>,T>(*this)(fCyc);
   else return F1<Z,Vl<T>,T>(*this)(fCon);
}

template <class T>
void Vl<T>::cow(void)
```

```
   // body is void if CPM_USECOUNT is not defined
{
#if defined(CPM_USECOUNT)
   if (u_.makeonly()){
      p_= sz_==0 ? 0 : copy();
      u_.startNew();
    //  cpmcerr<<endl<<"Vl<T>::cow() called"<<endl;
   }
#endif
}

template <class T>
Vl<T>::Vl(Z n):sz_(safeDim(n)),p_(new T[sit(sz_)])
{
   T* q=p_;
   for (Z i=0;i<sz_;++i) *q++ = T();
}

template <class T>
Vl<T>::Vl(Z n, T const& t):sz_(safeDim(n)),p_(new T[sit(sz_)])
{
   T* q=p_;
   for (Z i=0;i<sz_;++i) *q++ = t;
}

template <class T>
Vl<T>::Vl(Z n, F<Z,T> const& f):sz_(safeDim(n)),p_(new T[sit(sz_)])
{
   T* q=p_;
   for (Z i=0;i<sz_;++i) *q++ = f(i);
}

template <class T>
Vl<T>::Vl(std::vector<T> const& v):sz_((Z)v.size()),p_(new T[sit(sz_)])
{
   typename std::vector<T>::const_iterator i;
   T* q=p_;
   for (i=v.begin();i!=v.end();++i) *q++=*i;
}

template <class T>
std::vector<T> Vl<T>::std()const
{
   std::vector<T> res(sz_);
   typename std::vector<T>::iterator i;
   T* q=p_;
   for (i=res.begin();i!=res.end();++i) *i=*q++;
   return res;
}
```

```
template <class T>
#if defined(CPM_USECOUNT)
   Vl<T>::Vl(Vl<T> const& h) :sz_(h.sz_),u_(h.u_),p_(h.p_){}
#else
   Vl<T>::Vl(Vl<T> const& h) :sz_(h.sz_),p_(h.copy()){}
#endif

template <class T>
Vl<T>& Vl<T>::operator=(Vl<T> const& h)
{
   if (sz_==0 && h.sz_==0) return *this;
      // added 2002-03-07, should be OK
   if (this==&h) return *this;
#if !defined(CPM_USECOUNT)
   Z szMem=sz_;  // added 2000-11-17
   // definition restricted to the case !defined(CPM_USECOUNT)
   // 2005-06-22 to avoid waring on non use initialized
   // variable
#endif
   sz_=h.sz_;
#if defined(CPM_USECOUNT)
   if (u_.reattach(h.u_)) delete[] p_;
   p_=h.p_;
#else
   if (sz_!=szMem){ // added 2000-11-17 in order to avoid
      // superfluous delete, new action
      delete[] p_;
      p_=new T[sit(sz_)];
   }
   T* q=h.p_;
   T* it=p_;
   Z i=sz_;
   while (i--) *it++ = *q++;
#endif
   return *this;
}

template <class T>
Vl<T> Vl<T>::append(Vl<T> const& h)const
{
   Z i,sz2=sz_+h.sz_;
   T* p2=new T[sit(sz2)];
   T* it=p2;
   T* q1=p_;
   i=sz_;
   while(i--) *it++ = *q1++;
   T* q2=h.p_;
   i=h.sz_;
   while (i--) *it++ = *q2++;
   return Vl<T>(sz2,p2);
```

```
}

template <class T>
Vl<T> Vl<T>::append(T const& t)const
{
   Z i,sz2=sz_+1;
   T* p2=new T[sit(sz2)];
   T* it=p2;
   T* q=p_;
   i=sz_;
   while(i--) *it++ = *q++;
   *it++ = t;
   return Vl<T>(sz2,p2);
}

template <class T>
Vl<T> Vl<T>::rev()const
{
   if (sz_<2){
      return *this;
      // nothing to do if we have no or one component
   }
   else{
      T* p2=new T[sit(sz_)];
      T* it=p2;
      T* itp=p_;
      itp+=(sz_-1);
      Z i=sz_;
      while (i--)  *it++ = *itp--;
      return Vl<T>(sz_,p2);
   }
}

template <class T>
Vl<T> Vl<T>::resize(Z newDim)const
{
   Z n2= newDim<0 ? 0 : newDim;
   Z nCopy=(n2<=sz_ ? n2 : sz_);
   Vl<T> res(n2); // correctly initialized even for n2>sz_
   T* it=res.p_;
   T* itp=p_;
   Z i=nCopy;
   while (i--) *it++ = *itp++;
   return res;
}

template <class T>
Vl<T> Vl<T>::eli(IvZ const& iv)const
{
   IvZ i=iv&dom();
```

```
   if (i.isVoid()) return *this ;
   else return eli(i[1],i.card());
}

template <class T>
Vl<T> Vl<T>::eli1(Vl<T>& h, Z i)const
{
// Here we use natural counting of components so that
// the j-th component of h is h[j-1].
   Z nEli=h.dim();
   if (i<1) i=1;
   if (sz_==0){ // nothing eliminated, nothing left
      h=Vl<T>(0);
      return Vl<T>(0);
   }
   else if (i>sz_){ // nothing eliminated
      h=Vl<T>(0);
      return *this;
   }
   else{ // now i>=1 and i<=sz_ and sz_>=1
      // if true, we have sz_>=1
   // the i-th component is the first to be eliminated
   // so we have i-1 components of *this which are on the left-hand
   // side of the elimination area. These have to shine up in
   // the result vector to be returned
      Z nRes1=i-1;
   // So the maximum number of
   // components of *this that run the risk to get eliminated is
   // sz_-nRes1
      Z nEliRisk=sz_-nRes1;
      if (nEli>nEliRisk) nEli=nEliRisk;
      Z nRes2=sz_-nRes1-nEli;
      Z nRes=nRes1+nRes2;
      T* pRes=new T[sit(nRes)];
      T* pEli=new T[sit(nEli)];
      T* itRes=pRes;
      T* itEli=pEli;
      T* itp=p_;
      Z j=nRes1;
      while (j--) *itRes++=*itp++;
      j=nEli;
      while (j--) *itEli++=*itp++;
      j=nRes2;
      while (j--) *itRes++=*itp++; // for j==0 nothing done
      h=Vl<T>(nEli,pEli);
      return Vl<T>(nRes,pRes);
   }
}

template <class T>
```

```
Vl<T> Vl<T>::eli(Z i, Z nEli)const
{
   if (nEli<0) Error("Vl<T>::eli() : nEli<0");
   if (!(i>0 && i<=sz_)) Error("Vl<T>::eli() : !(i>0 && i<=sz_)" );
      // if true, we have sz_>=1
   Z remaining=1+sz_-i;   // number of component i and  followers
      // is >=1 due to assert()
   Z nEliActual=( nEli<=remaining ? nEli : remaining);
      // this is the number of components which we actually will be
      // removing
   Z j, sz1=sz_-nEliActual;
   if (sz1==0) return Vl<T>(0);
   else {
      T* p1=new T[sit(sz1)];
      T* it=p1;
      T* q1=p_;
      for (j=1;j<=i-1;j++) *it++ = *q1++; // the j is a counting device
         // only, no components are taken, so starting from 1 is no
         // mistake; it is a convenience.
      for (j=1;j<=nEliActual;j++) q1++;   // advancing over the elements
      // to be removed
      for (j=i;j<=sz1;j++) *it++ = *q1++;
      return Vl<T>(sz1,p1);
   }
}

template <class T>
Vl<T> Vl<T>::ins(Z i, Vl<T> const& h)const
{
   if (i<1 || i>(sz_+1)) cpmerror(
      "Vl<T>::ins(Z i,Vl<T>): unvalid argument i="&cpmwrite(i));
   Z j, sz1=sz_+h.sz_;
   T* p1=new T[sit(sz1)];
   T* it=p1;
   T* q1=p_;
   T* q2=h.p_;
   for (j=1;j<=i-1;j++)   *it++ = *q1++;
   for (j=1;j<=h.sz_;j++)  *it++ = *q2++; // h.sz_ terms
   for (j=i;j<=sz_;j++)    *it++ = *q1++;
   return Vl<T>(sz1,p1);
}

template <class T>
Vl<T> Vl<T>::ins(Z i, T const& t)const
{
   if (i<1 || i>(sz_+1))
      cpmerror("Vl<T>::ins(Z i,T): unvalid argument i="&cpmwrite(i));
   Z j, sz1=sz_+1;
   T* p1=new T[sit(sz1)];
   T* itp=p_;
```

```
   T* it=p1;
   for (j=1;j<=i-1;j++) *it++ = *itp++;
   *it++ =t;
   for (j=i;j<=sz_;j++)  *it++ = *itp++;
   return Vl<T>(sz1,p1);
}

template <class T>
T Vl<T>::in_(T const& t, bool reversed)
// safe logic by indexing, propably not utmost performance
{
   if (sz_==0) return T();
   cow();
   Z i;
   T res;
   if (!reversed){
      res=p_[sz_-1]; // last component of array
      for (i=sz_-1;i>0;i--){
         p_[i]=p_[i-1]; // causal processing: on the right-hand side we
         // evaluate only expressions which were not yet changed during
         // the loop
      }
      p_[0]=t;
   }
   else{
      res=p_[0]; // first component of array
      for (i=0;i<sz_-1;i++){
         p_[i]=p_[i+1]; // causal processing: on the right-hand side we
         // evaluate only expressions which were not yet changed during
         // the loop
      }
      p_[sz_-1]=t;
   }
   return res;
}

template <class T>
void Vl<T>::set_(T const& t)
{
   cow();
   Z i=sz_;
   T* it=p_;
   while(i--) *it++ = t;
}

template <class T>
void Vl<T>::show(ostream& out)const
{
   out<<endl<<"Vl<T>::show()";
#if defined(CPM_USECOUNT)
```

```
   out<<endl<<" usecount = "<<u_.getCount();
#endif
   out<<endl<<" start address = "<<p_;
   out<<endl<<" end address = "<<(p_+(sz_-1));
   long b=1+(long)((p_+(sz_-1))-p_);
   out<<endl<<" bytes="<<b;
}

template <class T>
Vl<T> Vl<T>::set(Z i, T const& t)const
{
   Z iC=i-1;
   // now iC is a 'C-pointer index'

   if (iC<0) return *this;   // nothing changed
   else if (iC<sz_){
      // vector can already hold the new value
      Vl<T> res(*this);
      res[iC]=t;
      return res;
   }
   else{
      // now we have to return an enlarged vector
      Z j, sz2=iC+1;
      Vl<T> res(sz2);
      res[iC]=t;
      T* it=res.p_;
      T* itp=p_;
      Z i=sz_;
      while (i--) *it++ = *itp++;
      return res;
   }
}

template <class T>
T Vl<T>::fAcc1(T (*f)(T const&), void (*acc)(T&, T const&))const
{
   T* p1=p_;
   T res=T();
   for (Z i=0;i<sz_;i++) acc(res,f(*p1++));
   return res;
}

template <class T>
T Vl<T>::fAcc2(T (*f)(T const&, T const&), void (*acc)(T&, T const&),
   Vl<T> const& h)const
{
   T* p1=p_;
   T* p2=h.p_;
   T res=T();
```

```
   for (Z i=0;i<sz_;i++) acc(res,f(*p1++,*p2++));
   return res;
}

template <class T>
Vl<T> Vl<T>::fa(T (*f)(T const&))const
{
   T* p1=new T[sit(sz_)];
   T* it=p1;
   T* itp=p_;
   Z i=sz_;
   while(i--) *it++ = f(*itp++);
   return Vl<T>(sz_,p1);
}

template <class T>
Vl<T> Vl<T>::fb(T (*f)(T const&, T const&), T const& t)const
{
   T* p1=new T[sit(sz_)];
   T* it=p1;
   T* itp=p_;
   Z i=sz_;
   while(i--) *it++ = f(*itp++,t);
   return Vl<T>(sz_,p1);
}

template <class T>
void Vl<T>::fc_(T (*f)(T const&, T const&), T const& t)
{
   cow();
   T val;
   T* itp=p_;
   Z i=sz_;
   while(i--){
      val=*itp;
      *itp++=f(val,t);
   }
}

template <class T>
void Vl<T>::fd(T (*f)(T const&, T const&), T& t)const
{
   T* itp=p_;
   Z i=sz_;
   while (i--) t=f(*itp++,t);
}

template <class T>
Vl<T> Vl<T>::fe(T (*f)(T const&, T const&), Vl<T> const& h)const
{
```

```
   if (sz_!=h.sz_) throw Error("Vl<T>::fe(): dimension mismatch");
   T* p1=new T[sit(sz_)];
   T* it=p1;
   T* p2=p_;
   T* p3=h.p_;
   Z i=sz_;
   while (i--) *it++ = f(*p2++,*p3++);
   return Vl<T>(sz_,p1);
}

template <class T>
Vl<T> Vl<T>::fe2(T (*f)(T const&, T const&), Vl<T> const& h)const
{
   if (sz_==h.sz_) return fe(f,h);
   else if (sz_>h.sz_){
      T t0=T();
      T* p1=new T[sit(sz_)];
      T* it=p1;
      T* p2=p_;
      T* p3=h.p_;
      Z i=sz_;
      Z ih=h.sz_;
      while (ih--){
         i--;
         *it++ = f(*p2++,*p3++);
      }
      while(i--){
         *it++ = f(*p2++,t0);
      }
      return Vl<T>(sz_,p1);
   }
   else{
      T t0=T();
      T* p1=new T[sit(h.sz_)];
      T* it=p1;
      T* p2=p_;
      T* p3=h.p_;
      Z i=sz_;
      Z ih=h.sz_;
      while (i--){
         ih--;
         *it++ = f(*p2++,*p3++);
      }
      while(ih--){
         *it++ = f(t0,*p3++);
      }
      return Vl<T>(h.sz_,p1);
   }
}
```

```
template <class T>
void Vl<T>::ff_(T (*f)(T const&, T const&), Vl<T> const& h, Z is)
{
   if (is<0) is=0;
   Z sz1=sz_-is;
   Z sz2=h.sz_;
   Z szMin=( sz1<sz2 ? sz1 : sz2);
   if (szMin<1) return; // nothing to do
   cow();
   T val;
   T* p2=h.p_;
   T* itp=p_+is;
   Z i=szMin;
   while (i--){
      val=*itp;
      *itp++=f(val,*p2++);
   }
}

template <class T>
void Vl<T>::fa_(F<T,T> const& f, IvZ iv)
{
   IvZ iva=dom()&iv;
   if (iva.isVoid()) return; // nothing to be done
   cow();
   T val;
   T* itp=p_;
   itp+=iva.inf();
      // setting the iterator pointer to the right place
   Z i=iva.card(); // this is the number of requested loop
      // actions, logic OK as the simple case i=1 tells
   while(i--){
      val=*itp;
      *itp++=f(val);
   }
}

template <class T>
void Vl<T>::fg_(T (*f)(T const&, T const&, T const&),
      Vl<T> const& h, T const& t)
{
   Z szMin=( sz_<=h.sz_ ? sz_ : h.sz_);
   cow();
   T val;
   T* p2=h.p_;
   T* itp=p_;
   Z i=szMin;
   while (i--){
      val=*itp;
      *itp++=f(val,*p2++,t);
```

```
   }
}

template <class T >
void Vl<T>::fh_(T const& w_1, T const& w0, T const& w1, Outside mode)
{
   if (sz_<1) return;
   cow();
   T v_1=read(-1,mode);
   T v0=p_[0];
   T v1=read(1,mode);
   Z k=0;
   while (k<sz_){
      p_[k]=v_1*w_1+v0*w0+v1*w1;
      ++k;
      v_1=v0;
      v0=v1;
      v1=read(k+1,mode);
   }
}

template <class T >
Vl<T> Vl<T>::select(const Vl<bool>& s)const
{
   Z n=dim(),ns=s.dim();
   Vl<T> resPrel=*this; // the actual result may have less components
   T* itPrel=resPrel.p_;
   T* itp=p_;
   Z iAct=0;
   Z i=n;
   Z is=0;
   while (i--){
      bool keepIt= (is<ns ? s[is++] : true);
      T tAct=*itp++;
      if (keepIt){
         *itPrel++=tAct;
         iAct++;
      }
   }
   if (iAct==n) return resPrel; // no element was purged
   else return resPrel.resize(iAct);
}

template <class T >
Vl<IvZ> Vl<T>::valOn(F<T,bool> const& f)const
{
   Z mL=3;
   static Word loc("Vl<T>::valOn(F<T,bool>)");
   CPM_MA
   Vl<IvZ> res(0);
```

```
   T* q=p_;
   bool yetFoundTrue=false;
   Z firstTrue=0, firstFalseAfterTrue=0;
   for (Z i=0;i<sz_;++i){
      bool val=f(*q++);
      if (val){ // we found true
         if (!yetFoundTrue){ // then start a interval of validity
            yetFoundTrue=true;
            firstTrue=i;
         }
         else{ // normally nothing to do
            // but if we are at the end of the array the
            // last pending truth interval has to be
            // considered as finished and has to be added
            if (i==sz_-1){
               firstFalseAfterTrue=sz_;
               IvZ ivAct("",firstTrue,firstFalseAfterTrue);
               res&=ivAct; // appending
               // nothing else to do since we are finished
               CPM_MZ
               return res;
            }
         }
      }
      else{ // we found false
         if (yetFoundTrue){ // then i is the terminator
            // of a truth interval
            firstFalseAfterTrue=i;
            IvZ ivAct("",firstTrue,firstFalseAfterTrue);
            res&=ivAct;
            yetFoundTrue=false;
         }
      }
   }
   CPM_MZ
   return res;
}

template <class T>
bool Vl<T>::prnOn(ostream& str)const
{
   Z mL=3;
   Word loc=nameOf()&"::prnOn(...)";
   CPM_MA
   cpmwt((nameOf()&" begin").str());
   CpmRoot::IO<Z>().o(sz_,str);
   T* q=p_;
   for (Z i=0;i<sz_;i++){
      if (!CpmRoot::IO<T>().o(*q++,str)){
         cpmwarning("failed to write component "&cpm(i+1)&" of "
```

```
            &cpm(sz_));
         CPM_MZ
         return false;
      }
   }
   cpmwt((nameOf()&" end").str());
      // for large sz_ it would be difficult to find the
      // end of the vector data if these are written to a file
      // and a human reader wants to inspect them
   CPM_MZ
   return true;
}

template <class T>
bool Vl<T>::scanFrom(istream& str)
{
   Z mL=3;
   Word loc=nameOf()&"::scanFrom(...)";
   CPM_MA
   Z n;
   bool suc=CpmRoot::IO<Z>().i(n,str);
   if (!suc){
      cpmwarning(loc&": can’t read dimension");
      CPM_MZ
      return false;
   }
   cpmmessage(mL,"dimension read as "&cpm(n));
   if (n>dimMax) cpmwarning(loc&": n>dimMax");
   Vl<T> res(n);
   for (Z i=res.b();i<=res.e();i++){
      suc=CpmRoot::IO<T>().i(res.cui(i),str);
      if (!suc){
         cpmwarning("failed to read component "&cpm(i+1)&" of "
            &cpm(sz_));
         CPM_MZ
         return false;
      }
   }
   *this=res;
   CPM_MZ
   return true;
}

template <class T >
Z Vl<T>::com(Vl<T> const& s)const
// short vectors < longer vectors
{
   Z d1=dim(), d2=s.dim();
   if (d1<d2) return 1;
   else if (d1>d2) return -1;
```

```
   else{
      T* q=p_;
      T* qs=s.p_;
      for (Z i=0;i<d1;i++){
         Z ci=CpmRoot::Comp<T>()(*q++,*qs++);
         if (ci!=0) return ci;
      }
      return 0;
   }
}

}  // namespace CpmArrays

namespace CpmRoot{
// This is a pattern for partial specializations --- to which
// the specializations to class templates belong --- of the
// IO class template started in cpmnumbers.h and the
// Name class template started in cpmword.h.
// A corresponding definition is needed for all non-C+- classes which
// shall be used as template arguments of C+- containers and
// are expected to provide then the same functionality as
// corresponding C+- classes.

   template<class T>
   class IO< std::vector<T> >{ // partial specialization
    public:
      IO(){}
      bool o(std::vector<T> const& v, ostream& str)const
      { return CpmArrays::Vl<T>(v).prnOn(str);}
      bool i(std::vector<T>& v, istream& str)const
      {
         CpmArrays::Vl<T> vl;
         bool res=vl.scanFrom(str);
         v=vl.std();
         return res;
      }
   };

#if defined(CPM_NAMEOF)
   template<class T>
   class Name< std::vector<T> >{ // partial specialization
   public:
      Name(){}
      Word operator()(std::vector<T> const& vt )const
      { return Word("std::vector<"&CpmRoot::Name<T>()(T())&">");}
   };
#endif
}
#endif
```

224 cpmvl.cpp

```
//? cpmvl.cpp
//? Status of work 2008-10-25.
//?
#include <cpmvl.h>

using CpmRoot::Z;
using CpmRoot::Word;
using CpmArrays::Vl;
using CpmArrays::IvZ;

Z CpmArrays::dimMax=4000000;
bool CpmArrays::ranChc=true;
bool CpmArrays::ranChcAlw=true;

void CpmArrays::setDimMax(Z n)
{
   if (n<0)
      cpmerror(cpmwrite(n)&" not OK as value for CpmArrays::dimMax");
   dimMax=n;
}

Z CpmArrays::safeDim(Z n)
{
   if (n<0 || n>dimMax)
      cpmerror(cpmwrite(n)&" not OK as dimension of CpmArrays::Vl<>");
   return n;
}

Vl<Z> CpmArrays::IvZtoVlofZ(IvZ const& iv)
{
   Z n=iv.card();
   Vl<Z> res(n);
   if (n==0) return res;
   Z i,val=iv.inf();
   for (i=res.b();i<=res.e();++i) res[i]=val++;
   return res;
}

Vl<Z> CpmArrays::VlofIvZtoVlofZ(Vl<IvZ> const& viv)
{
   Vl<Z> res(0);
   for (Z i=viv.b();i<=viv.e();++i) res&=IvZtoVlofZ(viv[i]);
   return res;
}

Vl<Word> CpmArrays::comLineLean(int argc, char* argv[])
{
```

```
   Vl<Word> res(argc);
   for (Z i=res.b();i<=res.e();i++) res[i]=argv[i];
   return res;
}
```

225 cpmvo.h

```
//? cpmvo.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_VO_H_
#define CPM_VO_H_
/*
   Purpose: Defining a array class Vo<T> which exploits the assumtion that
   T supports ordering. See cpmv.h

*/
#include <cpmv.h>

namespace CpmArrays{

//////////////////////// class Vo ////////////////////////////////////

template <class T>
   // We assume that T provides (explicitely or implicitely)
   // copy constructor, and assignement or that T is a built-in type.
   // T < T and T > T assumed.

class Vo : public V<T>{ // version of V with order-related operations
   // derivation which imlements the order interface
public:
   typedef Vo<T> Type;
   typedef V<T> Base;

explicit Vo(Z n=0):V<T>(n){}

   Vo(Z n, T const& t):V<T>(n,t){}

   Vo(const V<T>& h):V<T>(h){}
         // 'down-cast'-constructor

// access to the functionality of Vo in a framework in which
// Vl is considered basic. Notice that the conversion
// from Vo back to Vl gets performed by the function c()
// which is defined in V
   explicit Vo(Vl<T> const& vl):V<T>(vl){}

   CPM_ORDER

   Vl<T> toVl()const{ return Base::vl_;}

   virtual V<T>* clone(void)const{ return new Vo(*this);}

   virtual Word nameOf()const
```

```
   //: name of
{
   Word wi="Vo<";
   return wi&CpmRoot::Name<T>()(T())&">";
}

V<Z> permutationForIncreasingOrder(void)const;
   // Returned is an array per such that i<j enforces
   //  (*this)[per[i]]<(*this)[per[j]]
   //       or
   // (*this)[per[i]]==(*this)[per[j]]

V<Z> permutationForDecreasingOrder(void)const
   // analogous to previous function
{ return permutationForIncreasingOrder().rev();}

void order(void);
   // Permutes the indexes such that
   // (*this)[i]<(*this)[j] || (*this)[i]==(*this)[j]
   // whenever i<j. The components (instances of T) are allowed to
   // be large objects since excessive copying is avoided.

void sort_(void){ order();}
   //: sort

V<Z> perForDecOrd(void)const
   //: permutation for decreasing order
{ return permutationForDecreasingOrder();}

V<Z> perForIncOrd(void)const
   //: permutation for increasing order
{ return permutationForIncreasingOrder();}

Vo<T> increasingCopy(void)const;
   // returns a increasingly ordered version of (*this) without
   // changing *this

Vo<T> incCopy(void)const{ return increasingCopy();}
   //: increasing copy

Vo<T> strictlyIncreasingVersion(void)const;
   // returns a stricly increasingly ordered version of (*this)
   // which leaves out multiply appearing components
   // (again, without changing *this)

Vo<T> strIncVer(void)const{ return strictlyIncreasingVersion();}
   //: strictly increasing version

Vo<T> decreasingCopy(void)const;
   // returns a decreasingly ordered version of (*this) without
```

```
   // changing *this

Vo<T> decCopy(void)const{ return decreasingCopy();}
   //: decreasing copy

Z find(T const&, bool ordered=true)const;
   //: find
   // if the return value is 0, there is no i=1,...n
   // such that (*this)[i]==t. Otherwise, the return value i is
   // such that (*this)[i]==t.
   // If ordered==true, this assumes that (*this) is ordered in the
   // way achieved by function order(), else no assumptions on
   // ordering. The components are compared against t starting from 1
   // (to dim()). The first occurrence of t gets reported.

Vo<T> purge(const V<T>& s)const;
   // returned is a list which is obtained from *this by eliminating
   // all components which equal to one of the components of s. Beside
   // of this removal, the order of the components in *this is
   // retained.

Vo<T> pur(const V<T>& s)const{ return purge(s);}
   //: purge

Z locate(T const&)const;
   // returned is a value j such that t is between (*this)[j] and
   // (*this)[j+1]. (*this) must be monotonic, either increasing
   // (ascending) or decreasing.
   // j=0 or j=n is returned to indicate that t is out of range.

Z loc(T const& t)const{ return locate(t);}
   //: locate

Vo<T> enqueue(T const&,bool asc=true)const;
   // assumes that (*this) is monotonic. Then t will be inserted in
   // the orderly correct position into of a copy of (*this). This copy
   // remains monotonic and gets returned.
   // The second argument comes only in action for dim()==1, since
   // then no direction is provided by *this and a direction is needed
   // for the result (for dim()==0, the result has dimm()==1 and no
   // direction).

Vo<T> enq(T const& t,bool asc=true)const{ return enqueue(t,asc);}
   //: enqueue

T sup()const;
   //: supremum
   // returns the largest component of *this; faster than getting the
   // largest element by ordering
```

```
   T inf()const;
      //: infimum
      // returns the smallest component of *this; faster than getting the
      // largest element by ordering

   Z indSup()const;
      //: index (of) supremum
      // returns the smallest i for which (*this)[i]=(*this).sup()

   Z indInf()const;
      //: index (of) infimum
      // returns the smallest i for which (*this)[i]=(*this).inf()

   pair<Z,Z> indInfSup()const;
      //: returns indInf(), indSup() as a pair in a single pass

   pair<T,T> infSup()const;
      //: infimum supremum
      // Let res be the return value, then res.first==inf(),
      // res.second==sup(). Fast single-pass operation
};

// primary function of the order interface
template <class T >
Z Vo<T>::com(const Vo<T>& s)const
// short vectors < longer vectors
{
   return Base::vl_.com(s.c());
}

// Function indexx from Press et al. changed into a function template
// p. 339  (role of M: see p. 333; algorithm is based on Quicksort, see
// sort(), anyway the choice of M may influence performance but never
// puts a limitation on the size of the vector to be ordered.

namespace{
   inline void swapZ(Z& i, Z& j){ Z ii=i; i=j; j=ii;}
}

template <class T>
V<Z> Vo<T>::permutationForIncreasingOrder(void)const
// Indexes an array v[1..n], i.e. outputs the array indx[1..n] such that
// v[indx[j]] is in ascending order for j=1,2,...,n. The input
// quantities n and v are not changed. Especially, v-elements are
// not copied back and forth which is essential if T is a large class
{
   const Z mL=6;
   cpmmessage(mL,"Vo<T>::permutation...(): started");
   const Z M=7;
   const Z NSTACK=50;
```

```
Z nVec=Base::dim();
Vo<Z> indx(nVec);
Z i,indxt,ir=nVec,j,k,l=1,jstack=0;
T a;

Vo<Z> istack(NSTACK);

for (j=1;j<=nVec;j++) indx[j]=j;
for (;;) {
   if (ir-l < M) {
      for (j=l+1;j<=ir;j++) {
         indxt=indx[j];
         a=(*this)[indxt];
         for (i=j-1;i>=1;i--) {
            if ((*this)[indx[i]] < a) break;
            if ((*this)[indx[i]] == a) break;
            indx[i+1]=indx[i];
         }
         indx[i+1]=indxt;
      }
      if (jstack == 0) break;
      ir=istack[jstack--];
      l=istack[jstack--];
   } else {
      k=(l+ir)/2; // was >> 1 instead of /2 before
      swapZ(indx[k],indx[l+1]);
      if ((*this)[indx[l+1]] > (*this)[indx[ir]]) {
         swapZ(indx[l+1],indx[ir]);
      }
      if ((*this)[indx[l]] > (*this)[indx[ir]]) {
         swapZ(indx[l],indx[ir]);
      }
      if ((*this)[indx[l+1]] > (*this)[indx[l]]) {
         swapZ(indx[l+1],indx[l]);
      }
      i=l+1;
      j=ir;
      indxt=indx[l];
      a=(*this)[indxt];
      for (;;) {  // the i<nVec and j>1 test added by UM, otherwise in
            // patological cases of < and > relations we get a range
            // error
         do i++; while (i<nVec && (*this)[indx[i]] < a);
         do j--; while (j>1 && (*this)[indx[j]] > a);
         if (j < i) break;
         swapZ(indx[i],indx[j]);
      }
      indx[l]=indx[j];
      indx[j]=indxt;
```

```
         jstack += 2;
         if (jstack > NSTACK) cpmerror("NSTACK too small in ordering()");
         if (ir-i+1 >= j-l) {
            istack[jstack]=ir;
            istack[jstack-1]=i;
            ir=j-1;
         } else {
            istack[jstack]=j-1;
            istack[jstack-1]=l;
            l=i;
         }
      }
   }
   cpmmessage(mL,"Vo<T>::permutation...(): done");
   return indx;
}

// Function locate from Press et al. changed into a function template

template <class T>
Z Vo<T>::locate(const T& x)const
{
   Z ju,jm,jl,n=Base::dim();
   jl=0;
   ju=n+1;
   bool ascnd=((*this)[n] > (*this)[1]);
   while (ju-jl > 1) {
      jm=(ju+jl)/2;// Press et al. have >>1 istead of /2
      if ((x > (*this)[jm]) == ascnd)
         jl=jm;
      else
         ju=jm;
   }
   return jl;
}

// functions based on order relations in T

template <class T>
Vo<T> Vo<T>::increasingCopy(void)const
{
   Vo<Z> per=permutationForIncreasingOrder();
   Z i, n=Base::dim();
   Vo<T> y(n);
   for (i=1;i<=n;i++) y[i]=(*this)[per[i]];
   return y;
}

namespace{
   template <class T>
```

```
   bool fEqual(T const& t1, T const& t2){ return t1==t2;}
}

template <class T>
Vo<T> Vo<T>::strictlyIncreasingVersion(void)const
{
   Vo<T> res=increasingCopy();
   return res.condense(fEqual).first;
}

template <class T>
Vo<T> Vo<T>::decreasingCopy(void)const
{
   Vo<Z> per=permutationForIncreasingOrder();
   Z i, n=Base::dim();
   Vo<T> y(n);
   for (i=1;i<=n;i++) y[1+n-i]=(*this)[per[i]];
   return y;
}

template <class T>
void Vo<T>::order(void)
{
   V<Z> per=permutationForIncreasingOrder();
   Z i, n=Base::dim();
   Vo<T> y(n);
   for (i=1;i<=n;i++) y[i]=(*this)[per[i]];
   for (i=1;i<=n;i++) (*this)[i]=y[i];
}

template <class T >
Z Vo<T>::find(T const& t, bool ordered)const
{
   Z n=Base::dim();
   if (n==0) return 0;
   if (ordered){ // fast method
      Z jl=0, jm, ju=n+1;
      while (ju-jl > 1) {
         jm=(ju+jl) >> 1;
         if (t < (*this)[jm])
            ju=jm;
         else
            jl=jm;
      }
      if (jl<=n){
         if (jl<n && t==(*this)[jl+1]) return jl+1;
         if (jl>0 && t==(*this)[jl]) return jl;
      }
      return 0;
   }
```

```
   else{
      Z iRes=0; // correct result if the next loop doesn't find t
      for (Z i=1;i<=n;i++){
         if ((*this)[i]==t){ iRes=i; break;}
      }
      return iRes;
   }
}

template <class T >
Vo<T> Vo<T>::purge(const V<T>& s)const
{
   Z i,j,n=Base::dim(),ns=s.dim();
   V<bool> vs(n,true);
   for (i=1;i<=n;i++){
      T t=(*this)[i];
      for (j=1;j<=ns;j++){
         if( t==s[j] ){ vs[i]=false; break; }
      }
   }
   return Base::select(vs);
}

template <class T >
T Vo<T>::sup()const
{
   Z n=Base::dim();
   cpmassert(n>0,"T Vo<T>::sup()const");
   T t,res=(*this)[1];
   for (Z i=2;i<=n;i++){
      t=(*this)[i];
      if (t>res) res=t;
   }
   return res;
}

template <class T >
Z Vo<T>::indSup()const
{
   Z n=Base::dim();
   cpmassert(n>0,"Vo<T>::indSup()const");
   T t,valMax=(*this)[1];
   Z res=1;
   for (Z i=2;i<=n;i++){
      t=(*this)[i];
      if (t>valMax){ valMax=t; res=i;}
   }
   return res;
}
```

```
template <class T >
T Vo<T>::inf()const
{
   Z n=Base::dim();
   cpmassert(n>0,"T Vo<T>::inf()const");
   T t,res=(*this)[1];
   for (Z i=2;i<=n;i++){
      t=(*this)[i];
      if (t<res) res=t;
   }
   return res;
}

template <class T >
Z Vo<T>::indInf()const
// needed as fast as possible
{
   Z n=Base::dim();
   cpmassert(n>0,"Z Vo<T>::indInf()const");
   Z res=0;
   T t,valMin=Base::vl_.cui(res);

   for (Z i=1;i<n;i++){
      t=Base::vl_.cui(i);
      if (t<valMin){ valMin=t; res=i;}
   }
   return ++res;
}

template <class T >
pair<Z,Z> Vo<T>::indInfSup()const
{
   Z n=Base::dim();
   if (n<=0) cpmerror("Vo<T>::indInfSup(): dim()<=0");
   T t,valMin=(*this)[1],valMax=valMin;
   Z resMin=1;
   Z resMax=1;
   for (Z i=2;i<=n;i++){
      t=(*this)[i];
      if (t<valMin){ valMin=t; resMin=i;}
      if (t>valMax){ valMax=t; resMax=i;}
   }
   return pair<Z,Z>(resMin,resMax);
}

template <class T >
pair<T,T> Vo<T>::infSup()const
{
   Z n=Base::dim();
   if (n<=0) cpmerror("Vo<T>::infSup(): dim()<=0");
```

```
   T t,valMin=(*this)[1],valMax=valMin;
   for (Z i=2;i<=n;i++){
      t=(*this)[i];
      if (t<valMin){ valMin=t; }
      if (t>valMax){ valMax=t; }
   }
   return pair<T,T>(valMin,valMax);
}

template <class T >
Vo<T> Vo<T>::enqueue(T const& t, bool asc)const
{

   Z d=Base::dim();
   if (d==0) return Vo<T>(1,t);
   if (d==1){
      Vo<T> res(2);
      T t1=Base::vl_[0];
      if (asc){
         if (t1<t){
            res[1]=t1;
            res[2]=t;
         }
         else{
            res[1]=t;
            res[2]=t1;
         }
      }
      else{
         if (t1<t){
            res[1]=t;
            res[2]=t1;
         }
         else{
            res[1]=t1;
            res[2]=t;
         }
      }
      return res;
   }
   Z i=locate(t);
   return ins(i+1,t);
}
//////////////////////////// iterated templates ////////////////////////////
// describing multi-indexed quantities
//////////////////////////////// class VVo ////////////////////////////////
// matrices

template <class T>
class VVo: public VV<T>{// version of VV with order-related operations
```

```
   typedef VV<T> Base;
   typedef VVo<T> Type;

public:
   CPM_ORDER
// constructors
   VVo(Z d1, Z d2):Base(d1,d2){}
   VVo( Z d1, Z d2, T const& t):Base(d1,d2,t){}
      // all components are equal to t
   VVo(void):Base(){}
   VVo(VV<T> const& x):Base(x){}
   VVo( V< V<T> > const& x):Base(x){}
};

template <class T >
Z VVo<T>::com(VVo<T> const& s)const
{
   Z d1=Base::dim(), d2=s.dim();
   if (d1<d2) return 1;
   else if (d1>d2) return -1;
   else{
      for (Z i=1;i<=d1;i++){
         Vo<T> ti=(*this)[i];
         Vo<T> si=s[i];
         Z ci=CpmRoot::com<T>(ti,si);
         if (ci!=0) return ci;
      }
      return 0;
   }
}

//////////////////////// class VVVo ////////////////////////////////////
// tensors of rank 3

template <class T>
class VVVo: public VVV<T>{// version of VVV with order-related operations

   typedef VVV<T> Base;
   typedef VVVo<T> Type;

public:
   CPM_ORDER
// constructors
   VVVo(Z d1,Z d2,Z d3):Base(d1,d2,d3){}
   VVVo(Z d1,Z d2,Z d3,T const& t):Base(d1,d2,d3,t){}
      // all components are equal to t
   VVVo(void):Base(){}
   VVVo(VVV<T> const& x):Base(x){}
   VVVo( V< V< V<T> > > const& x):Base(x){}
```

```
};

template <class T >
Z VVVo<T>::com(VVVo<T> const& s)const
{
   Z d1=Base::dim(), d2=s.dim();
   if (d1<d2) return 1;
   else if (d1>d2) return -1;
   else{
      for (Z i=1;i<=d1;i++){
         VVo<T> ti=(*this)[i];
         VVo<T> si=s[i];
         Z ci=CpmRoot::com<T>(ti,si);
         if (ci!=0) return ci;
      }
      return 0;
   }
}

//////////////////////// class VVVVo ////////////////////////////////
// tensors of rank 4

template <class T>
class VVVVo:public VVVV<T>{//version of VVVV with order-related operations

   typedef VVVV<T> Base;
   typedef VVVVo<T> Type;

public:
   CPM_ORDER
// constructors
   VVVVo(Z d1,Z d2,Z d3,Z d4):Base(d1,d2,d3,d4){}
   VVVVo(Z d1,Z d2,Z d3,Z d4,T const& t):Base(d1,d2,d3,d4,t){}
      // all components are equal to t
   VVVVo(void):Base(){}
   VVVVo(VVVV<T> const& x):Base(x){}
   VVVVo( V< V< V< V<T> > > > const& x):Base(x){}
};

template <class T >
Z VVVVo<T>::com(VVVVo<T> const& s)const
{
   Z d1=Base::dim(), d2=s.dim();
   if (d1<d2) return 1;
   else if (d1>d2) return -1;
   else{
      for (Z i=1;i<=d1;i++){
         VVVo<T> ti=(*this)[i];
         VVVo<T> si=s[i];
         Z ci=CpmRoot::com<T>(ti,si);
```

```
         if (ci!=0) return ci;
      }
      return 0;
   }
}

} // namespace

#endif
```

226 cpmvr.h

```
//? cpmvr.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_VR_H_
#define CPM_VR_H_
/*

Purpose: Defining a vector template providing the full
functionality of the rich interface

*/
#include <cpmva.h>

//////////////////////////// class Vr<> ////////////////////////////////

namespace CpmArrays{

   using CpmRoot::R;

template <class T>
   // We assume that T provides (explicitely or implicitely)
   // copy constructor, and assignement
   // or that T is a built-in type.

class Vr: public Va<T>{ // version of Va with a rich interface

public:
   typedef Vr<T> Type;
   typedef Va<T> Base;
   typedef T ScalarType; // was missing till 2001-10-12
      // detected by the gnu compiler

explicit   Vr(Z n=0):Va<T>(n){}
      // note that Vr(0) is defined and is different from Vr()

   Vr(Z n,const T& t):Va<T>(n,t){}
      // initializes all n components with t

   Vr(const V<T>& h):Va<T>(h){}
      // 'down-cast'-constructor

   Vr(const Vo<T>& h):Va<T>(h){}
      // 'down-cast'-constructor

   Vr(const Va<T>& h):Va<T>(h){}
      // 'down-cast'-constructor
```

```
// access to the functionality of Vr in a framework in which
// Vl is considered basic. Notice that the conversion
// from Vr back to Vl gets performed gy the function c()
// which is defined in V
   explicit Vr(Vl<T> const& vl):Va<T>(V<T>(vl)){}

   virtual V<T>* clone(void)const{ return new Vr<T>(*this);}

   CPM_TEST
   CPM_DESCRIPTORS
      // disVal from Va
};

// member functions

template <class T>
Word Vr<T>::nameOf(void)const
{
   Word w1="Vr<";
   Word w2=CpmRoot::Name<T>()(T());
   return w1&w2&">";
}

template <class T>
Word Vr<T>::toWord(void)const
{
   Z d=Base::dim();
   Word res="( ";
   for (Z i=1;i<d;i++){
      Word wi=CpmRoot::toWord<T>((*this)[i]);
      wi&=", ";
      res&=wi;
   }
   if (d>0) res&=CpmRoot::toWord<T>((*this)[d]);
   res&=" )";
   return res;
}

template <class T>
Z Vr<T>::hashVal(void)const
{
   Z hash=0;
   for (Z i=1; i<=Base::dim(); i++){
      hash+=CpmRoot::hashVal<T>((*this)[i]);
   }
   return hash;
}

template <class T>
Vr<T> Vr<T>::testVal(Z cpl)const
```

```
{
   Z it=0;
   it=CpmRoot::testVal<Z>(it,cpl);
   T xt;
   xt=CpmRoot::testVal<T>(xt,cpl);
   return Vr<T>(it,xt);
}

template <class T>
Vr<T> Vr<T>::ranVal(Z j)const
{
   Vr<T> res(Base::dim());
   T tempX;
   if (j==0){
      for (Z i=1; i<=Base::dim(); i++){
         tempX=(*this)[i];
         res[i]=CpmRoot::ranVal<T>(tempX,j);
      }
   }
   else{
      Z j0=1000; // this is done to avoid an obvious correlation
         // among the coordinates of x.ranVal(i)
         // and x.ranVal(i+1)
      Z jIncr=CpmRoot::ranVal<Z>(j0,j);
      Z jc=j;
      for (Z i=1; i<=Base::dim(); i++){
         tempX=(*this)[i];
         jc+=jIncr;
         res[i]=CpmRoot::ranVal<T>(tempX,jc);
      }
   }
   return res;
}

// We do not implement the iterated templates since they are now
// endowed with sufficient functionality already as Va, VVa, VVVa,
// and VVVVa

} // namespace

#endif
```

227 cpmwaves.h

```
//? cpmwaves.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_WAVES_H_
#define CPM_WAVES_H_

/*
   Description : wave equation in one dimension.
   Introducing a subdivision of the 1-dimensional discrete
   chain that is intended to help parallelization
*/

#include <cpmrfunc.h>
#include <cpmrmatrix.h>
#include <cpmlinalg.h>
#include <cpmv.h>

namespace CpmWaves{

using CpmRoot::Z;
using CpmRoot::R;
using CpmRoot::Word;
using CpmRootX::Z1;
using CpmRootX::R1;
using CpmRootX::B;
using CpmLinAlg::R2;
using CpmArrays::R_Vector;
using CpmArrays::R_Matrix;
using CpmArrays::V;
using CpmFunctions::R_Func;
using CpmGeo::Iv;

//////////////////// class Wave ////////////////////////////////////////////

class Wave{ // states of a discrete string
   Z1 d;
      // number of discrete points in representing the state of the string
   R1 c;
      // velocity in the wave equation (could be made position
      // dependent for more interesting applications)
   R1 dx;
      // spatial separation of neighbor points.
   Z1 method;
      // method to compute a time step
      // presently 0 or 1
      // 0 Euler method, 1 explicit mid point method (see
      // implementation of Wave::step(R) in file cpmwaves.cpp)
```

```
   R_Vector x;
      // x[i] is the wave quantity (e.g. elongation of the elastic
      // string) at point i; i=1,2,...d. We treat the periodic problem in
      // which the right neighbor of i=d is i=1
   R_Vector v;
      // dx[i]/dt. Such a representation is superior to considering x
      // alone and let x[i,t],x[i,t-dt] be a representation of the
      // velocity

   R1 fac;
      // dependent quantity (c/dx)^2; needed for efficient coding of
      // function acc

   R_Vector acc(const R_Vector& ra); // acceleration
      // function which determines the dynamics; in our case it is
      // c^2 * second spatial derivation. The discrete version of
      // the spatial derivatives are written in a way that the
      // periodicity mentioned above is taken into account.

   void finish(){ fac=c/dx; fac*=fac;}

   typedef Wave Type;
      // this typedef is expected by the following declaration
      // macro CPM_IO, which includes a unified definition of send
      // and receive for Wave objects
public:
   Word nameOf()const{ return "Wave";}
   CPM_IO
      // declaration and partial implementation of I/O by a macro
explicit   Wave(Z d0=1, R c0=1, R dx0=1 );
      // a basic constructor
   void step(R dt);
      // time stepping algorithm, depends on value of method
   R get_c()const{ return c;}
      // evident
   R get_dx()const{ return dx;}
      // evident
   R2 getLeft()const{ return R2(x[2],v[2]);}
      // returns the data that a 'left neighbor' needs to replace
      // his own unreliably computed boundary values
   R2 getRight()const{ return R2(x[d-1],v[d-1]);}
      // returns the data that a 'right neighbor' needs to replace
      // his own unreliably computed boundary values

   R2 to(Z j)const
  { if (j==-1) return getRight(); else return getLeft(); }

   void from(Z j, R2 xv)
  //    { if (j==1) setRight(xv); else setLeft(xv);}
  { if (j==1) setRight(xv); else setLeft(xv); }
```

```
   void setRight(R2 xv){ x[d]=xv[1]; v[d]=xv[2];}
      // replaces own unreliable boundary data by the
      // values given by the argument (these argument values have to
      // be obtained by calling getLeft() (sic!) for the right neighbor
   void setLeft(R2 xv){ x[1]=xv[1]; v[1]=xv[2];}
      // replaces own unreliable boundary data by the
      // values given by the argument (these argument values have to
      // be obtained by calling getRight() for the ledt neighbor
   void setLocalState(Z j, R ax, R vx)
      {x.cyc(j)=ax; v.cyc(j)=vx;}
      // evident
   R2 getLocalState(Z j)const{ return R2(x.cyc(j),v.cyc(j));}
      // evident
   Z dim()const{ return d;}
      // evident
   void setMethod(Z m){ method=m;}
      // evident
   R_Matrix rep()const;
     // graphical object representing the amplitude distribution
};

//////////////////// class Waves ////////////////////////////////////////

class Waves{ // array of waves on strata with one-point overlap

   Z1 d;
   R1 c;
   R1 dx;
   Z1 method;
   V<Wave> vw;
   typedef Waves Type;
public:
   CPM_IO
   Waves(){}
   Waves(const V<Z>& vz, R c0, R dx0);
   Waves(const V<Wave>& vw0);
   void step(R dt);
   void setState(const R_Func& ampl, const R_Func& vel);
      // setting amplitude and velocity from functions
      // defined on 1-dimensional position space
   void adjust();
      // sets the boundary values in a consistent manner
   R_Matrix rep()const;
      // graphical object representing the amplitude distribution
   Iv getSpace()const;
      // getting the lower and the upper end of the x-space covered
   void setMethod(Z m)
      { method=m; for (Z i=1;i<=d;i++) vw[i].setMethod(m);}
   Wave& operator[](Z i){ return vw[i];}
```

```
   const Wave& operator[](Z i)const{ return vw[i];}
   V<Wave> get_vw()const{ return vw;}
};

} // namespace

#endif
```

228 cpmwaves.cpp

```
//? cpmwaves.cpp
//? Status of work 2008-10-25.
//?
// cpmwaves.cpp
// see cpmwaves.h

#include <cpmwaves.h>

using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmArrays;
using namespace CpmFunctions;

using CpmWaves::Wave;
using CpmWaves::Waves;

////////////////////// class Wave ////////////////////////////////

Wave::Wave(Z d0, R c0, R dx0):d(d0),x(d),v(d),c(c0),dx(dx0)
{ finish();}

R_Vector Wave::acc(const R_Vector& x)
{
   Z d=x.dim();
   R_Vector res(d);
   for (Z i=2;i<d;i++){ // i.e. we let the boundary values i=1 and i=n
      // equal to 0. Thus the positions of the boundary points will
      // not be computed correctly. This is no problem since the
      // boundary points will be correctly computed by the
      // neighbor-systems
      res[i]=fac*(x[i-1]-2*x[i]+x[i+1]);
   }
   return res;
}

void Wave::step(R dt)
// terse code (following a proposal by Thomas Dera)
{
   R dth=dt*0.5; // dt half
   if (method==0){ // Euler method; direct, simple, but unstable
      // also 4 loops over the chain (2 calls to add, 1 call to +,
      // 1 copy
      R_Vector vOld=v;
      v.add(acc(x),dt); // v now changed, vNew=vOld+acc*dt
      x=x.add(v+vOld,dth);
         // efficient way of saying xNew=xOld+vOlt*dt+acc*(dt^2)/2
   }
```

```
   else if(method==1){ // explicit midpoint method; well known for
      // ordinary differential equations. Mostly interpreted in a tricky
      // way as a 2-step method which then goes under the name Leap Frog.
      // Since discretized partial differential equations are time
      // discrete
      // ordinary differential equations in an obvious way, there is
      // nothing
      // fundamentally new involved here. Nevertheless, it seems not to
      // be
      // usual to treat partial differential equations this way. Why ?
      // I tested the method also for the Schrödinger equation and found
      // norm and energy conserved to a very impressive degree (also
      // long term !)
      //
      // Three loops over the chain (3 calles to add)
      x.add(v,dth);
         // midpoint configuration
      x.add(v.add(acc(x),dt),dth);
   }
   else{ // ment as terse form of Euler, obsolete!
      // Not working; when + is being evaluated v already was changed and
      // the
      // original value is no longer available (both arguments of
      // + are references!) so the result is the same as
      // x.add(2*v.add(acc(x),dt),dth); Although the behavior for my
      // present
      // test case is not bad, this method has no value since it gives an
      // error proportional to t^2 for constant acceleration. This was a
      // hard lesson;
      // thanks to Thomas Dera for helping to understand the situation.
      x.add(v+v.add(acc(x),dt),dth);
   }
}

/*********
void Wave::prnOn(ostream& str)const
{
   d.prnOn(str);
   c.prnOn(str);
   dx.prnOn(str);
   fac.prnOn(str);
   method.prnOn(str);
   x.prnOn(str);
   v.prnOn(str);
}
****************/

bool Wave::prnOn(ostream& str)const
{
   cpmp(d);
```

```
   cpmp(c);
   cpmp(dx);
   cpmp(fac);
   cpmp(method);
   cpmp(x);
   cpmp(v);
   return true;
}

bool Wave::scanFrom(istream& str)
{
   cpms(d);
   cpms(c);
   cpms(dx);
   cpms(fac);
   cpms(method);
   cpms(x);
   cpms(v);
   return true;
}

R_Matrix Wave::rep()const
{
   Z i;
   R_Matrix res(2,d);
   R pos=0;
   for (i=1;i<=d;i++){
      res[1][i]=pos;
      res[2][i]=x[i];
      pos+=dx;
   }
   return res;
}

////////////////////// class Waves ////////////////////////////////

Waves::Waves(const V<Z>& vz, R c0, R dx0):d(vz.dim()),vw(d),c(c0),dx(dx0)
{
  for (Z i=1;i<=d;i++) vw[i]=Wave(vz[i],c0,dx0);
}

Waves::Waves(const V<Wave>& vw0):d(vw0.dim()),vw(vw0),
   c(vw0[1].get_c()),dx(vw0[1].get_dx())
{}

void Waves::adjust()
{
/********
   Z mesLim=3;
   if (cpmverbose>=mesLim)
```

```
      cpmmessage("Waves::adjust() started");
   for (Z i=1;i<=d;i++){
      Z iL=vw.cyclic(i-1);
      Z iR=vw.cyclic(i+1);
      R2 left=vw[iL].getRight();
      R2 right=vw[iR].getLeft();
      vw[i].setRight(right);
      vw[i].setLeft(left);
   }
   if (cpmverbose>=mesLim)
      cpmmessage("Waves::adjust() done");
***********/
   // old function cyclic does no longer exist, i
   // if needed it could easily be reconstructed
}

void Waves::step(R dt)
{
   for (Z i=1;i<=d;i++) vw[i].step(dt);
   adjust();
}

void Waves::setState(const R_Func& ampl, const R_Func& vel)
{
   Z mesLim=3;
   if (cpmverbose>=mesLim)
      cpmmessage("Waves::setState(...) started");
   R x=0;
   for (Z i=1;i<=d;i++){
      Z di_=vw[i].dim();
      for (Z j=2; j<di_; j++){ // set only the
         // propriatary points of each component
         x+=dx;
         R ax=ampl(x);
         R vx=vel(x);
         vw[i].setLocalState(j,ax,vx);

      }
   }
   adjust();
   if (cpmverbose>=mesLim)
      cpmmessage("Waves::setState(...) done");
}

Iv Waves::getSpace()const
{
   Z i,ic=0;
   for (i=1;i<=d;i++){
      ic+=vw[i].dim()-2;
   }
```

```
   return Iv(0,ic*dx);
}

R_Matrix Waves::rep()const
      // graphical object representing the amplitude distribution
{
   Z i,j,ic=0;
   for (i=1;i<=d;i++){
      ic+=vw[i].dim()-2;
   }
   Z dRep=ic;
   R_Matrix res(2,dRep);
   ic=0;
   R x=0;
   for (i=1;i<=d;i++){
      Z di_=vw[i].dim();
      for (j=2; j<di_; j++){ // set only the
         // propriatary points of each component
         R2 ls=vw[i].getLocalState(j);
         ic++;
         x+=dx;
         res[1][ic]=x;
         res[2][ic]=ls[1];
      }
   }
   return res;
}

bool Waves::prnOn(ostream& str)const
{
   cpmp(d);
   cpmp(c);
   cpmp(dx);
   cpmp(method);
   cpmp(vw);
   return true;
}

bool Waves::scanFrom(istream& str)
{
   cpms(d);
   cpms(c);
   cpms(dx);
   cpms(method);
   cpms(vw);
   return true;
}
```

229 cpmword.h

```
//? cpmword.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_WORD_H_
#define CPM_WORD_H_
/*
   Description: Declares class Word which is now essentially identical
      to string of the standard library. It is formed from it according
      to the advice in BS3, end of Chapter 20.3.

      Word is a minimal string class to which Cpm classes refer for
         type identification and for messaging
*/
#include <cpminterfaces.h>
#if !defined(CPM_NAMEOF)
   #include <typeinfo>
#endif

namespace CpmRoot{

   using namespace CpmStd;

   class Word;

   Word toWord(R x, Z prc);
   Word toWord(Rh x, Z prc);
   Word toWord(unsigned int i, char* ff);
      // ff standard format string
   Word toWord(unsigned long int i, char* ff);
   Word toWord(int i, char* ff);
   Word toWord(long int i, char* ff);
   Word toWord(double r, char* ff);
   Word toWord(long double r, char* ff);

   ofstream& to_ofstream(Word const& w);
   ifstream& to_ifstream(Word const& w);
   string toString(Word const& w);

// concatenation of words described by operator &
   Word operator &(Word const&, Word const&);
   Word operator &(const char* cp, Word const& w);
   Word operator &(Word const& w, const char* cp);
   Word operator &(char cp[], Word const& w);
   Word operator &(Word const& w, char cp[]);

// for convenience same is allowed to be denoted +
   Word operator +(Word const&, Word const&);
```

```
   Word operator +(const char* cp, Word const& w);
   Word operator +(Word const& w, const char* cp);
   Word operator +(char cp[], Word const& w);
   Word operator +(Word const& w, char cp[]);

//////////////////////// class Word ////////////////////////////////////

   class Word{ //  wrapping charcter strings, rich functionaliy
      // Instances of Word are dynamically allocated strings of
      // characters like those of string (on which class the present
      // implementation is based). Input and Output works as expected only
      // if we have a word in the narrower sense that the character
      // string does not contain 'whitespace'.
      // Due to functions toBool(), toZ(), toR() a Word is a universal
      // carrier of numerical information. For instance the content
      // of a command line may be represented as a list of Words (an
      // instance of V<Word> in C+- )

   string rep_;

   static bool scnLin;
      // modifies the function scanFrom() in a way that also character
      // strings containing blanks can be read in to yield a single Word.
      // Such a facility is needed if Word is to be used as the data type
      // of names of more complex entities like organizations, theories,
      // etc.: Certainly 'set theory' and 'Eastman Kodak Company' should
      // be allowed names.

   static Z endOfWord; // end of Word
      // 0 nothing, 1 blank, else newline after w.prnOn()

   typedef Word Type;
   typedef Word CloneBase;

public:

   static void setScanLine(bool sl){ scnLin=sl;}
      //: set scan line
      // see private static data member scnLin
   static bool getScanLine(){ return scnLin;}
      //: get scan line
      // see private static data member scnLin

   static void setEndOfWord(Z i){ endOfWord=i;}
      //: set end of word
      // see private static data member endOfWord
   static Z getEndOfWord(){ return endOfWord;}
      //: get end of word
      // see private static data member endOfWord
```

```
explicit Word(char c):rep_(1,c){}
// a Word consisting of just one character c

explicit Word(Z i, char c='a'):rep_(i,c){}
// a Word consisting of i copies of character c
// useful for getting new words by changing letters at given positions

Word(const char* charPtr):rep_(charPtr){}
   // construction of a Word from char*, and thus from literals
   // such as Word w("convenient programming")
   // Although this defines a conversion from const char* to
   // Word, in a statement  cout<<"This is ..."<<endl;  the
   // interpretation of "This is ..." as const char* applies,
   // since exact matches are preferred over matches which employ
   // user-defined conversions. This is important, since
   // cout<<Word("This is ...")<<endl; may append some separator
   // to the input string depending on the value of the
   // static data member endOfWord.

explicit Word(string const& s):rep_(s){}
   // construction of a Word from string

explicit Word(ostringstream const& ost):rep_(ost.str()){}
   // construction of a Word from a string stream
   // This is the most convenient way to create detailed texts held as
   // a Word, as in the following code fragment:
   // ostringstream ost;
   // ost<<...<<...endl<<...;
   // Word text(ost);

Word():rep_(){}
   // default constructor, creates the void word which consists of
   // 0 characters

const char* operator-(void)const{ return rep_.c_str();}
   // explicit conversion from Word to const char*
   // The result is owned by the Word and the user should not try to
   // delete it.

const char* c_str(void)const{ return rep_.c_str();}
   // to make Word and string as much alike as possible

char* c_string(void)const;
   // the user has to delete the result before it goes out of scope
   // modifies from BS3, p. 590

string str()const{ return rep_;}
   //: string

string toStr()const{ return rep_;}
```

```
      //: to string

   Word skipChr(char c)const;
      //: skip character
      // Returns a Word which results from *this by eliminating c
      // everywhere.

   void skipSpc_();
      //: skip space, mutating function
      // eliminates all characters which by the function isspace(char)
      // of <cctype> is classified as space

   void skipSurSpc_();
      //: skip surrounding space
      // eliminates all space characters (whitespace) as above which
      // surround a chain of space separated traditional words.
      // So the result is a robust representation for a textual content
      // of a line of text, since unvisible parts at the beginning and
      // the end get eliminated
      // added 2004-10-12

   Word firstWord()const;
      // returns the first whitespace separated part of the string rep_
      // (which may contain whitespace). It would be nice to
      // return a complete list of Words here, but we prefer not to
      // build on any array class in the present basic class.
      // The result may be the void Word.

   Word firstWord_();
      // returns the first whitespace separated part of the string rep_
      // (which may contain whitespace) and changes *this into
      // the part of the word which remains.
      // The ending '_' is reminescent of Ruby's '!' as a indicator
      // for mutating functions. Calling this function in succession
      // allows to create a complete list of words in a line without
      // having a list- or array-class available at this point.

// we have to define later a general template function
// template <class T1, class T2>
// operator & (const T1&, const T2&)
// This implies that in (non-template !) functions operator & only exact
// matches will be accepted (see Stroustrup p. 589), certainly not the
// ones that can be achieved by userdefined conversions. So, we have to
// take care of char* and char[] in this place

// concatenation of words described by operator &
   Word& operator &=(Word const& w);
   Word& operator &=(const char* cp);
   friend Word operator &(Word const&, Word const&);
   friend Word operator &(const char* cp, Word const& w);
```

```
   friend Word operator &(Word const& w, const char* cp);
   friend Word operator &(char cp[], Word const& w);
   friend Word operator &(Word const& w, char cp[]);

// for convenience same is allowed to be denoted +. This is convenient
// since + binds stronger than & and especially stronger than <<
   Word& operator +=(Word const& w);
   Word& operator +=(const char* cp);
   friend Word operator +(Word const&, Word const&);
   friend Word operator +(const char* cp, Word const& w);
   friend Word operator +(Word const& w, const char* cp);
   friend Word operator +(char cp[], Word const& w);
   friend Word operator +(Word const& w, char cp[]);

// access functions
   Z dim(void)const{ return Z(rep_.length());}
      //: dimension
      // is 0 for the void Word
   Z b(void)const{ return 1;}
      // b for begin, first index of a letter
   Z e(void)const{ return dim();}
      // e for end, last index of a letter
      // A loop over the letters of Word w takes the form
      // for (Z i=w.b();i<=w.e();++i){ ... = w[i];}
      // This is correct also for the void word.
      // This looping style is also valid for all C+- array classes.
   Z size(void)const{ return dim();}
      //: size
   Z length(void)const{ return dim();}
      // for uniformity with std library types

   bool isVoid()const { return dim()==0;}
      //: is void
      //  short answer on whether the dimension is zero

   friend Z length(Word const& w){ return w.dim();}
      // number of characters (no addition for a terminator)

   char operator[](Z i)const;
      // save reading of characters. Valid indexes are 1...dim()

   char& operator[](Z i);
      // save overwriting of characters in a word.

   bool valInd(Z i)const{ return i>0 && i<=dim();}
      //: valid index
      // returns the validity of i as an index of *this

   Word cut(Z i)const;
      //: cut
```

```
      // returned is a Word which results from *this by removing i
      // components from the end. If i<0 we remove -i components from
      // the beginning.

// functions dealing Words as filenames
   Word resize(Z newDim)const;
      // returns a Word res such that res.dim()=newDim. If newDim is
      // smaller than dim(), the end of *this will be cut away.
      // If newDim is larger, blanks will be added.

   Word remExt(bool extended=false)const;
      //: remove extension
      // Intention of the function: returns what is left if any
      // file name extension (if there is one) is removed from
      // *this. So for *this=="myprog.ini" we return "myprog".
      // Actually we read and copy *this from the beginning until we
      // find a character which signifies the beginning of the extension.
      // In nomal mode this is a dot, in extended mode this is any capital
      // letter. The extended mode arrose from the need to distinguish
      // executables made by different compilers by a suffix such as
      // palaMS.exe, palaMinGW.exe, pala_ms.exe. Since the application
      // framework in cpmapplication.cpp deduces the program name,
      // and thus the name of the ini-file to search, from
      // the name of the executable, the above-mentioned executables would
      // try to read ini-files palaMS.ini, palaMinGW.ini, and
      // pala_ms.ini. They should, however, read simply pala.ini.
      // This is achieved by removing the extension from the executables
      // name for a true value of the argument.
      // The presently implemented notion of extension in 'extended mode'
      // is, that it is everything whatis not a lower-case letter or a
      // digit. This reflects

   Word appBefExt(Word const& app)const;
      //: append before extension
      // Word("myprog.ini").appBefExt("2") yields "myprog2.ini"

   Word baseName()const;
      //: base name
      // Only the characters right to the right-most "\" or "/"
      // are returned as a Word. If *this is a file name including
      // a directory path, then the actual file name gets extracted and
      // returned. (corresponds do File:basename of Ruby)

   bool readable()const;
      // returns true if *this is the name of a file that can be
      // opened and gives a valid istream to read from.
      // The stream will be closed after the test and not returned.

   Word makeFileName()const;
      //: make file name
```

```
      // returns a version of *this in which all characters which may not
      // be acceptable in file names are replaced by "_"
// end of functions dealing Words as filenames

   Word rev()const;
      //: reverse
      // returns the reversed version of *this, i.e. the one read from
      // the end

   Word& rev_(){ return *this=rev();}
      //: reverse
      // mutating form of reversal

   Word slc_(char c, bool firstSep=true);
      //: slice
      // If c does not occur in *this, we return the original *this
      // change *this to Word("").
      // One could do it the other way round, then the implementation
      // of remExt in terms of the present function would need to handle
      // the case that no dot is present (as for executables in Linux)
      // would need special treatment.
      // Now the interesting case:
      // If c appears in *this, we fix the position where it
      // appears in the leftmost position for firstSep==true
      // and in the rightmost position else.
      // we return the part of the word which is left of c
      // and change *his to the part right of c. Then we have for
      // Word w=...;
      // Word wMem=w;
      // char c=...; // such that c occurs in w !
      // Word wl=w.slc_(c);
      // the property
      // wl&Word(c)&w==wMem;

   Word tail(Z i)const;
      //: tail
      // if i>=dim() we return *this. Else we remove as many components
      // from the beginning that only i remain (' the last i components of
      // the original Word') and return the result.

   Z find(char c)const;
      //: find
      // returns the first position (1.....dim()) of a c in the
      // Word *this. If the character c isn't there we get 0

    Z find(Word const& w)const;
      //: find
      // returns the first position (1.....dim()) of a
      // subword w in the Word *this. If the Word w isn't there
      // we get 0
```

```
   static Word write(R r, Z prec=0);
      // implemented as { return Word(toStr(r,prec));}
      // see CpmRoot::toStr(R,Z) for the exact meaning

   static Word write(int n, Z fieldLength=0);
   static Word write(long int n, Z fieldLength=0);
   static Word write(unsigned int n, Z fieldLength=0);
   static Word write(unsigned long int n, Z fieldLength=0);
   static Word write(string h);
   static Word write(bool b);

   R toR()const;
      // outputs a R obtained by converting *this
   Z toZ()const;
      // outputs a Z obtained by converting *this
   bool toBool()const;
      // outputs a bool obtained by converting *this

// these declarations are repeated outside the class
// and have only influence on the implementation

   friend Word toWord(unsigned int i, char* ff);
   friend Word toWord(unsigned long int i, char* ff);
   friend Word toWord(int i, char* ff);
   friend Word toWord(long int i, char* ff);
   friend Word toWord(double r, char* ff);
   friend Word toWord(long double r, char* ff);

   friend ofstream& to_ofstream(Word const& w);
   friend ifstream& to_ifstream(Word const& w);
   friend string toString(Word const& w);

   bool readLine(istream& in){return CpmRoot::readLine(rep_,in);}
      // making *this equal to the content of the next non-comment
      // line in stream in

   CPM_IO
      // scanFrom(in);  makes *this equal to the next noncomment
      // whitespace separated sequence of characters in stream in.
      // Thus the code
      //
      //    Word w("This is a word");
      //    w.prnOn(aStream);
      //    Word w2;
      //    w2.scanFrom(aStream)
      //
      // will yield w2=="This"
      // and not w2=="This is a word"! This is sometimes what one wants,
      // sometimes not.
```

```
   CPM_ORDER
   CPM_TEST_ALL // implementation borrowed from string
// Declarations of CPM_DESCRIPTORS together with implementation
   Word toWord()const{ return *this;}
   Word nameOf()const{ return "Word";}
};

inline string toString(Word const& w){ return w.rep_;}

#define cpmwrite CpmRoot::Word::write
#define cpm CpmRoot::Word::write

// conversion of selected basic types to string
// as a basis for doing the conversion to Word in a consistent
// manner
// prc<=0 means that no precision gets set in the code, so that we then
// rely on the default precision set by the compiler.
   string toStr(Z x);
   string toStr(N x);
   string toStr(R x, Z prc=0);
   string toStr(Rh x, Z prc=0);
   string toStr(L x);
   string toStr(bool x);

//////////////////// Name<> ////////////////////////////////////////////
// W need a template class istead of a template function since we need
// specializations also for templates such as std::vector<> (see cpmvl.h).
// Whether one has CPM_NAMEOF defined or not --- the C+- basic
// built-in types always have the C+- name (i.e 'R', not 'double')
// For the more complex types, the names created by typeinfo
// (which look acceptable) are used for CPM_NAMEOF not defined.

   template<class T>
   class Name{ // tool class template for defining the nameOf function
   public:
      Name(){}
      Word operator()(T const& t)const
      {
#if defined(CPM_NAMEOF)
         return t.nameOf();
#else
         return Word(typeid(t).name());
#endif
      }
   };

   template<>
   class Name<Z>{ // Z: integers
   public:
      Name(){}
```

```
   Word operator()(Z const& t)const
      { t; return Word("Z");}
};

template<>
class Name<N>{ // N: natural numbers
public:
   Name(){}
   Word operator()(N const& t)const
      { t; return Word("N");}
};

template<>
class Name<R>{ // R: real numbers
public:
   Name(){}
   Word operator()(R const& t)const
      { t; return Word("R");}
};

template<>
class Name<Rh>{ // Rh: real number of reduced accuracy
public:
   Name(){}
   Word operator()(Rh const& t)const
      { t; return Word("Rh");}
};

template<>
class Name<L>{ // L: letter
public:
   Name(){}
   Word operator()(L const& t)const
      { t; return Word("L");}
};

template<>
class Name<bool>{ // specialization
public:
   Name(){}
   Word operator()(bool const& t)const
      { t; return Word("bool");}
};

template<>
class Name<char>{ // specialization
public:
   Name(){}
   Word operator()(char const& t)const
      { t; return Word("char");}
```

```
   };

   template<>
   class Name<string>{ // specialization
   public:
      Name(){}
      Word operator()(string const& t)const
         { t; return Word("string");}
   };

   template<class T>
   Word nameOf(T const& t){ return Name<T>()(t); }
      //: name of
   // convenient syntax for getting a name of the type of
   // a quantity. Definition is such that one may also give
   // Cpm-names to class templates. See beginning of cpmvl.h
   // for treating template <class T> std::vector<T> in this way.

//////////////////////// class ToWord ////////////////////////////////

   template<class T>
   class ToWord{
   public:
      ToWord(){}
      Word operator()(T const& t)const{ return t.toWord();}
   };

   template<>
   class ToWord<Z>{
   public:
      ToWord(){}
      Word operator()(Z const& t)const{ return Word(toStr(t));}
   };

   template<>
   class ToWord<N>{
   public:
      ToWord(){}
      Word operator()(N const& t)const{ return Word(toStr(t));}
   };

   template<>
   class ToWord<L>{
   public:
      ToWord(){}
      Word operator()(L const& t)const{ return Word(toStr(t));}
   };

   template<>
   class ToWord<bool>{
```

```
   public:
      ToWord(){}
      Word operator()(bool const& t)const{ return Word(toStr(t));}
   };

   template<>
   class ToWord<string>{
   public:
      ToWord(){}
      Word operator()(string const& t)const{ return Word(t);}
   };

   template<>
   class ToWord<R>{
   public:
      ToWord(){}
      Word operator()(R const& t)const{ return Word(toStr(t));}
   };

   template<>
   class ToWord<Rh>{
   public:
      ToWord(){}
      Word operator()(Rh const& t)const{ return Word(toStr(t));}
   };

   template<class T>
   Word toWord(T const& t){ return ToWord<T>()(t); }
      //: to word
      // Presently not important, MPI-functionalty relies on
      // prnOn and scanFrom

} // namespace

#define cpmnam CpmRoot::nameOf
#define cpmtow CpmRoot::toWord

#endif
```

230 cpmword.cpp

```
//? cpmword.cpp
//? Status of work 2008-10-25.
//?
#include <cpmtypes.h>
#include <stdio.h>
#include <cctype>
#include <iomanip>

using namespace CpmStd;
using CpmRoot::Word;
using CpmRoot::Z;
using CpmRoot::N;
using CpmRoot::R;
using CpmRoot::Rh;
using CpmRoot::L;
using CpmRoot::skipLeadingWhitespace;
using CpmRoot::skipTrailingWhitespace;

bool Word::scnLin=true;
   // true is dangerous for the presenly implemented movie file
   // production. Don't change without careful analysis of the
   // consequences

Z Word::endOfWord=2; // was 1
   // 0 nothing, 1 blank, else endl

bool Word::prnOn(ostream& str)const
{
   if (endOfWord==0) str<<rep_;
   else if (endOfWord==1) str<<rep_<<" ";
   else              str<<rep_<<endl;
   return (str!=0);
}

bool Word::scanFrom(istream& str)
{
   if (scnLin) CpmRoot::readLine(rep_,str);
   else        CpmRoot::read(rep_,str);
   return (str!=0);
}

char* Word::c_string(void)const
{
   char* p=new char[rep_.length()+1];
   rep_.copy(p,string::npos);
   p[rep_.length()]='\0';
   return p;
```

```
}

Z Word::find(char c)const
{
   Z res=0;
   Z n=dim();
   for (Z i=0;i<n;++i){
      if (rep_[i]==c){
         res=i+1;
         break;
      }
   }
   return res;
}

Z Word::find(const Word& w)const
{
   string::size_type res=rep_.find(w.rep_);
   if (res==string::npos) return 0;
   else return 1+(Z)res;
}

Word Word::resize(Z newDim)const
{
   Z n1=dim();
   Z n2= newDim<0 ? 0 : newDim;
   char blank=' ';
   Word res(n2,blank);
   Z nMin = n1<=n2 ? n1 : n2;
   for (Z i=1;i<=nMin;++i) res[i]=(*this)[i];
   return res;
}

Word Word::slc_(char c, bool firstSep)
{
   Z d=dim();
   if (d==0) return *this;
   Word temp= firstSep ? *this : rev();
   Z ic=temp.find(c);
   if (ic==0){ Word res=*this; *this=""; return res;}
   if (d==1 && ic==1){ // Then *this consists of a single c
      *this="";
      return "";
   }
   Word wl(ic-1);
   Word wr(d-ic);
   Z i,j;
   for (i=1;i<ic;++i) wl[i]=temp[i];
   for (i=ic+1,j=1;i<=d;++i,++j) wr[j]=temp[i];
   if (firstSep){
```

```
      *this=wr;
      return wl;
   }
   else{
      *this=wl.rev();
      return wr.rev();
   }
}

Word Word::appBefExt(Word const& app)const
{
   Word dot(".");
   Z posDot=find(dot);
   if (posDot==0) return *this&app;
   else{
      Z nameL=posDot-1;
      Z extL=dim()-posDot+1;
      Word name=resize(nameL);
      Word ext=tail(extL);
      name&=app;
      name&=ext;
      return name;
   }
}

Word Word::baseName()const
{
   Word res;
   string::size_type i1=rep_.find_last_of("/");
   string::size_type i2=rep_.find_last_of("\\");
   string repx=rep_;
   if (i1==string::npos && i2==string::npos){
      ; // nothing to do
   }
   else if (i1==string::npos){ // then i2 is a valid string position that
      // holds a backslash
      repx.erase(0,i2+1);
   }
   else if (i2==string::npos){// then i2 is a valid string position that
      // holds a slash
      repx.erase(0,i1+1);
   }
   else{
      string::size_type imax=i1;
      if (i2>i1) imax=i2;
         // then right of imax there is np slash or backslash
      repx.erase(0,imax+1);
   }
   return Word(repx);
}
```

```
char Word::operator[](Z i)const
{
   return rep_.at(i-1); // checked access accorcing to the error handling
      // facilities of the standard library
}

char& Word::operator[](Z i)
{
   return rep_.at(i-1);// checked access accorcing to the error handling
      // facilities of the standard library
}

Word Word::cut(Z i)const
{
   string res=rep_;
   if (i>=0){
      res.erase(res.length()-i,i);
   }
   else{
      res.erase(0,-i);
   }
   return Word(res);
}

Word Word::tail(Z n)const
{
   Z d=dim();
   if (d<=n) return *this;
   Z diff=d-n; // notice diff>=1
   Word res(n);
   for (Z i=0;i<n;i++) res.rep_[i]=rep_[i+diff];
      // notice that for i==n-1, we have i+diff=n-1+d-n=d-1
      // so that we just stop with the last letter of rep_.
      // No essential efficiency compromises. Should be fast.
      // We do not use the substring functionality of the standard
      // library since my compiler does not understand Stroustrup's name
      // Basic_substring<char> even after using namespace std;
      // Also forming a new template class for this minor purpose
      // seems not be justified; also the floppy description of the
      // substring constructors on p.596 of BSC3 would require some
      // testing for being sure what the function do in case of
      // non-matching arguments.
   return res;
}

Word CpmRoot::operator &(const Word& w1, const Word& w2)
{
   return Word(w1.rep_+w2.rep_);
}
```

```
Word CpmRoot::operator +(const Word& w1, const Word& w2)
{
     return Word(w1.rep_+w2.rep_);
}

Word Word::write(bool x)
{
   return CpmRoot::toWord(x);
}

Word Word::write(string x)
{
   return Word(x);
}

Word Word::write(R x, Z prec)
{
   return Word(CpmRoot::toStr(x,prec));
}

Word Word::write(int n, Z length)
{
   ostringstream ost;
   if (length>0) ost<<std::setfill('0')<<std::setw(length)<<n;
   else ost<<n;
   return Word(ost.str());
}

Word Word::write(long int n, Z length)
{
   ostringstream ost;
   if (length>0) ost<<std::setfill('0')<<std::setw(length)<<n;
   else ost<<n;
   return Word(ost.str());
}

Word Word::write(unsigned int n, Z length)
{
   ostringstream ost;
   if (length>0) ost<<std::setfill('0')<<std::setw(length)<<n;
   else ost<<n;
   return Word(ost.str());
}

Word Word::write(unsigned long int n, Z length)
{
   ostringstream ost;
   if (length>0) ost<<std::setfill('0')<<std::setw(length)<<n;
   else ost<<n;
```

```
   return Word(ost.str());
}

R Word::toR()const
{
   stringstream sstr;
   sstr<<rep_;
   R res;
   sstr>>res;
   return res;
}

Z Word::toZ()const
{
   stringstream sstr;
   sstr<<rep_;
   Z res;
   sstr>>res;
   return res;
}

bool Word::toBool()const
{
   if (rep_=="true" || rep_=="TRUE" ||
      rep_=="wahr" || rep_=="1" || rep_=="W")
      return true;
   else
      return false;
}

namespace{
   const int fieldLength=32;
   // privat name to file
}

Word CpmRoot::toWord(unsigned i, char* ff)
   // ff =standard format string
{
   char p[fieldLength];
   sprintf(p,ff,i);
   return Word(p);
}

Word CpmRoot::toWord(unsigned long i, char* ff)
{
   char p[fieldLength];
   sprintf(p,ff,i);
   return Word(p);
}
```

```
Word CpmRoot::toWord(int i, char* ff)
{
   char p[fieldLength];
   sprintf(p,ff,i);
   return Word(p);
}

Word CpmRoot::toWord(long i, char* ff)
{
   char p[fieldLength];
   sprintf(p,ff,i);
   return Word(p);
}

Word CpmRoot::toWord(double r, char* ff)
{
   char p[fieldLength];
   sprintf(p,ff,r);
   return Word(p);
}

Word CpmRoot::toWord(long double r, char* ff)
{
   char p[fieldLength];
   sprintf(p,ff,r);
   return Word(p);
}

Word CpmRoot::operator &(const char* cp, const Word& w)
{ return Word(cp)&w;}
Word CpmRoot::operator &(const Word& w, const char* cp)
{ return w&Word(cp);}
Word CpmRoot::operator &(char cp[], const Word& w)
{ return Word(cp)&w;}
Word CpmRoot::operator &(const Word& w, char cp[])
{ return w&Word(cp);}
Word& Word::operator &=(const Word& w)
{ return *this=(*this)&w;}
Word& Word::operator &=(const char* cp)
{ return *this=(*this)&Word(cp);}

// for convenience same is allowed to be denoted +

Word CpmRoot::operator +(const char* cp, const Word& w)
{ return Word(cp)&w;}
Word CpmRoot::operator +(const Word& w, const char* cp)
{ return w&Word(cp);}
Word CpmRoot::operator +(char cp[], const Word& w)
{ return Word(cp)&w;}
Word CpmRoot::operator +(const Word& w, char cp[])
```

```
{ return w&Word(cp);}
Word& Word::operator +=(const Word& w)
{ return *this=(*this)&w;}
Word& Word::operator +=(const char* cp)
{ return *this=(*this)&Word(cp);}

ofstream& CpmRoot::to_ofstream(const Word& w)
{
    ofstream*  outptr=new ofstream(w.rep_.c_str());
    return *outptr;
}

ifstream& CpmRoot::to_ifstream(const Word& w)
{
    ifstream*  inptr=new ifstream(w.rep_.c_str());
    return *inptr;
}

bool Word::readable()const
{
   ifstream ifs(rep_.c_str());
   if (ifs) return true;
   else return false;
}

Word Word::firstWord()const
{
   istringstream ist(rep_);
   string st;
   ist>>st;
   return Word(st);
}

Word Word::firstWord_()
{
   istringstream ist(rep_);
   string st;
   ist>>st;
   rep_=string();
   char c;
   while (ist>>c) rep_+=c;
   return Word(st);
}

Z Word::com(const Word& w)const
{
   if (rep_<w.rep_) return 1;
   else if (rep_>w.rep_) return -1;
   else return 0;
}
```

```
Word Word::makeFileName()const
{
   string::const_iterator i;
   ostringstream res;
   for (i=rep_.begin(); i!=rep_.end(); ++i){
      if (isalnum(*i)) res<<*i;  else res<<'_';
   }
   return Word(res);
}

void Word::skipSpc_()
{
   string::const_iterator i;
   ostringstream res;
   for (i=rep_.begin(); i!=rep_.end(); ++i){
      if (!isspace(*i)) res<<*i;
   }
   rep_=res.str();
}

Word Word::skipChr(char c)const
{
   string::const_iterator i;
   ostringstream res;
   for (i=rep_.begin(); i!=rep_.end(); ++i){
      if (*i!=c) res<<*i;
   }
   return Word(res);
}

namespace{
   bool isOK(char c, bool extended)
   {
      if (!extended) return c!='.';
      else return islower(c)||isdigit(c);
   }
}

Word Word::remExt(bool extended)const
{
   string::const_iterator i;
   ostringstream res;
   for (i=rep_.begin(); i!=rep_.end(); ++i){
      if (isOK(*i,extended)) res<<*i;
      else break;
   }
   return Word(res);
}
```

```
void Word::skipSurSpc_()
{
   string line=skipLeadingWhitespace(rep_);
   rep_=skipTrailingWhitespace(line);
}

R Word::absVal(void)const{ return CpmRoot::absVal<string>(rep_);}
R Word::absSqr(void)const{ return CpmRoot::absSqr<string>(rep_);}
Word Word::con(void)const{ return Word(CpmRoot::con<string>(rep_));}
Word Word::inv(void)const{ return Word(CpmRoot::inv<string>(rep_));}
Word Word::rev(void)const{ return Word(CpmRoot::inv<string>(rep_));}

Word Word::testVal(Z complexity)const
{
   string s;
   string res=CpmRoot::testVal<string>(s,complexity);
   return Word(res);
}

Word Word::ranVal(Z j)const
{
   string res=CpmRoot::ranVal<string>(res,j);
   return Word(res);
}
Z Word::hashVal()const
{
   return CpmRoot::hashVal<string>(rep_);
}

R Word::disVal(Word const& x)const
{
   return CpmRoot::disVal<string>(rep_,x.rep_);
}

Word Word::net(Z i)const
{
   return Word(CpmRoot::net<string>(rep_,i));
}

// functions CpmRoot::toStr
// The parameter prc (precision) makes no sense for integer types
// All conversions from built-in types to Word, which have not
// a char*-typed formatting string as one argument, are based on these
// toStr-functions.

string CpmRoot::toStr(R x, Z prc)
{
   ostringstream ost;
   if (prc>0) ost.precision(prc);
   ost<<x;
```

```
   return ost.str();
}

string CpmRoot::toStr(Rh x, Z prc)
{
   ostringstream ost;
   if (prc>0) ost.precision(prc);
   ost<<x;
   return ost.str();
}

#define CPM_FB ostringstream ost; ost<<x; return ost.str();
string CpmRoot::toStr(Z x){ CPM_FB }
string CpmRoot::toStr(N x){ CPM_FB }
string CpmRoot::toStr(L x){ CPM_FB }
string CpmRoot::toStr(bool x){ CPM_FB }
#undef CPM_FB
```

231 cpmx.h

```
//? cpmx.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_X_H_
#define CPM_X_H_
/*

Purpose: Defining lean cartesian products of 2-8 factors
   to be used similar to pair<> of the standard library.
   Needed for enabling functions to return longer value
   lists than just pairs.
   Non-constant reference arguments as a work-around are considered
   bad style here.
   If one needs cartesian products very often as function arguments or as
   function return values, this might be an indication that it would
   natural to define one or more classes which aggregate these components
   into a meaningful object.

   All these cartesian producs define IO and order operations in the
   same manner as Vl and V do this. This is a considerable simplification
   compared to my old approach to have templates Xno, Xnio for those
   template arguments which implement order operations and IO operations.

   Notice that access to components can be done via references and
   values. Notice also that, other than for pair, first,second,...
   are functions and not data. So the following is correct:
   X2<R,Z> a(4,6);
   Z i=a.second(); // not i=a.second;
      // i.e. i=6
   However,
   Z i=a.c2();
   looks nicer to me



*/
#include <cpmword.h>

namespace CpmArrays{

   using CpmRoot::Word;
   using CpmRoot::Z;
   using namespace CpmStd;

// Classes introduced by this file with fixed names:

//       X2, X3, X4, X5, X6, X7, X8
```

```
//

// All template arguments are assumed to have (not necessarily to define
// explicitely) default constructor, copy constructor, and operator =

// Lists of 2,3,4,5,6,7 or 8 components (hetero-typic lists)

///////////////////////////// class X2<> ///////////////////////////

template <class Y1, class Y2>
class X2{ // heterotypic pairs
   // Lean Cartesian product of Y1 with Y2
protected:
   Y1 x1;
   Y2 x2;
   typedef X2<Y1,Y2> Type;

public:
   CPM_IO
   CPM_ORDER
 // constructors
   X2(Y1 const& x1_, Y2 const& x2_):x1(x1_),x2(x2_){}
   X2(void):x1(),x2(){}
   X2(const X2<Y1,Y2>& z): x1(z.x1), x2(z.x2){}

// 'modern' access functions. Similar to operator[]
// getting the second component of object a is done like
//
// ...=a.c2();
//
// setting is done like
//
// a.c2()=...;
//
   Y1& c1(void){ return x1;}
   Y2& c2(void){ return x2;}
   Y1 const& c1(void)const{ return x1;}
   Y2 const& c2(void)const{ return x2;}
   Y1 first()const{ return x1;}
   Y2 second()const{ return x2;}
   Word nameOf()const
   {
       return Word("X2< "&
          CpmRoot::Name<Y1>()(Y1())&
          ","&
          CpmRoot::Name<Y2>()(Y2())&
          " >");
   }
};
```

```
template <class Y1, class Y2>
Z X2<Y1,Y2>::com(X2<Y1,Y2> const& s)const
{
   if (c1()<s.c1()) return 1;
   if (c1()>s.c1()) return -1;
   if (c2()<s.c2()) return 1;
   if (c2()>s.c2()) return -1;
   return 0;
}

template <class Y1, class Y2>
bool X2<Y1,Y2>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   return true;
}

template <class Y1, class Y2>
bool X2<Y1,Y2>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   return true;
}

//////////////////////////// class X3<> ////////////////////////////

// for more factors:

template <class Y1, class Y2, class Y3>
class  X3{ // heterotypic triplets
protected:
   Y1 x1;
   Y2 x2;
   Y3 x3;
   typedef X3<Y1,Y2,Y3> Type;
public:
   CPM_IO
   CPM_ORDER
   X3(Y1 const& x1_, Y2 const& x2_, Y3 const& x3_):x1(x1_),x2(x2_),
      x3(x3_){}
   X3(void):x1(),x2(),x3(){}
   X3(const X3<Y1,Y2,Y3>& z): x1(z.x1), x2(z.x2), x3(z.x3){}

   Y1& c1(void){ return x1;}
   Y2& c2(void){ return x2;}
   Y3& c3(void){ return x3;}

   Y1 const& c1(void)const{ return x1;}
```

```
   Y2 const& c2(void)const{ return x2;}
   Y3 const& c3(void)const{ return x3;}

   Y1 first()const{ return x1;}
   Y2 second()const{ return x2;}
   Y3 third()const{ return x3;}
   Word nameOf()const
   {
       return Word("X3< "&
          CpmRoot::Name<Y1>()(Y1())&
          ","&
          CpmRoot::Name<Y2>()(Y2())&
          ","&
          CpmRoot::Name<Y3>()(Y3())&
          " >");
   }
};

template <class Y1, class Y2, class Y3>
Z X3<Y1,Y2,Y3>::com(X3<Y1,Y2,Y3> const& s)const
{
   if (c1()<s.c1()) return 1;
   if (c1()>s.c1()) return -1;
   if (c2()<s.c2()) return 1;
   if (c2()>s.c2()) return -1;
   if (c3()<s.c3()) return 1;
   if (c3()>s.c3()) return -1;
   return 0;
}

template <class Y1, class Y2, class Y3>
bool X3<Y1,Y2,Y3>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   cpmp(x3);
   return true;
}

template <class Y1, class Y2, class Y3>
bool X3<Y1,Y2,Y3>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   cpms(x3);
   return true;
}

//////////////////////////// class X4<> ////////////////////////////
```

```
template <class Y1, class Y2, class Y3, class Y4>
class  X4{ // heterotypic 4-tuples
protected:
   Y1 x1;
   Y2 x2;
   Y3 x3;
   Y4 x4;
   typedef X4<Y1,Y2,Y3,Y4> Type;

public:
   CPM_IO
   CPM_ORDER
   X4(Y1 const& x1_, Y2 const& x2_, Y3 const& x3_, Y4 const& x4_):
      x1(x1_),x2(x2_),x3(x3_),x4(x4_){}
   X4(void):x1(),x2(),x3(),x4(){}
   X4(const X4<Y1,Y2,Y3,Y4>& z): x1(z.x1), x2(z.x2), x3(z.x3),x4(z.x4){}

   Y1& c1(void){ return x1;}
   Y2& c2(void){ return x2;}
   Y3& c3(void){ return x3;}
   Y4& c4(void){ return x4;}

   Y1 const& c1(void)const{ return x1;}
   Y2 const& c2(void)const{ return x2;}
   Y3 const& c3(void)const{ return x3;}
   Y4 const& c4(void)const{ return x4;}

   Y1 first()const{ return x1;}
   Y2 second()const{ return x2;}
   Y3 third()const{ return x3;}
   Y4 fourth()const{ return x4;}
   Word nameOf()const
   {
       return Word("X4< "&
          CpmRoot::Name<Y1>()(Y1())&
          ","&
          CpmRoot::Name<Y2>()(Y2())&
          ","&
          CpmRoot::Name<Y3>()(Y3())&
          ","&
          CpmRoot::Name<Y4>()(Y4())&
          " >");
   }
};

template <class Y1, class Y2, class Y3, class Y4>
Z X4<Y1,Y2,Y3,Y4>::com(X4<Y1,Y2,Y3,Y4> const& s)const
{
   if (c1()<s.c1()) return 1;
   if (c1()>s.c1()) return -1;
```

```
   if (c2()<s.c2()) return 1;
   if (c2()>s.c2()) return -1;
   if (c3()<s.c3()) return 1;
   if (c3()>s.c3()) return -1;
   if (c4()<s.c4()) return 1;
   if (c4()>s.c4()) return -1;
   return 0;
}

template <class Y1, class Y2, class Y3, class Y4>
bool X4<Y1,Y2,Y3,Y4>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   cpmp(x3);
   cpmp(x4);
   return true;
}

template <class Y1, class Y2, class Y3, class Y4>
bool X4<Y1,Y2,Y3,Y4>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   cpms(x3);
   cpms(x4);
   return true;
}

//////////////////////////// class X5<> ////////////////////////////

template <class Y1, class Y2, class Y3, class Y4, class Y5>
class  X5{ // heterotypic 5-tuples
protected:
   Y1 x1;
   Y2 x2;
   Y3 x3;
   Y4 x4;
   Y5 x5;
   typedef X5<Y1,Y2,Y3,Y4,Y5> Type;
public:
   CPM_IO
   CPM_ORDER
   X5(Y1 const& x1_, Y2 const& x2_, Y3 const& x3_,
      Y4 const& x4_, Y5 const& x5_):x1(x1_),x2(x2_),x3(x3_),
      x4(x4_),x5(x5_){}
   X5(void):x1(),x2(),x3(),x4(),x5(){}
   X5(const X5<Y1,Y2,Y3,Y4,Y5>& z): x1(z.x1), x2(z.x2),
      x3(z.x3),x4(z.x4),x5(z.x5){}
```

```
   Y1& c1(void){ return x1;}
   Y2& c2(void){ return x2;}
   Y3& c3(void){ return x3;}
   Y4& c4(void){ return x4;}
   Y5& c5(void){ return x5;}

   Y1 const& c1(void)const{ return x1;}
   Y2 const& c2(void)const{ return x2;}
   Y3 const& c3(void)const{ return x3;}
   Y4 const& c4(void)const{ return x4;}
   Y5 const& c5(void)const{ return x5;}

   Y1 first()const{ return x1;}
   Y2 second()const{ return x2;}
   Y3 third()const{ return x3;}
   Y4 fourth()const{ return x4;}
   Y5 fifth()const{ return x5;}
   Word nameOf()const
   {
       return Word("X5< "&
          CpmRoot::Name<Y1>()(Y1())&
          ","&
          CpmRoot::Name<Y2>()(Y2())&
          ","&
          CpmRoot::Name<Y3>()(Y3())&
          ","&
          CpmRoot::Name<Y4>()(Y4())&
          ","&
          CpmRoot::Name<Y5>()(Y5())&
          " >");
   }
};

template <class Y1, class Y2, class Y3, class Y4, class Y5>
Z X5<Y1,Y2,Y3,Y4,Y5>::com(X5<Y1,Y2,Y3,Y4,Y5> const& s)const
{
   if (c1()<s.c1()) return 1;
   if (c1()>s.c1()) return -1;
   if (c2()<s.c2()) return 1;
   if (c2()>s.c2()) return -1;
   if (c3()<s.c3()) return 1;
   if (c3()>s.c3()) return -1;
   if (c4()<s.c4()) return 1;
   if (c4()>s.c4()) return -1;
   if (c5()<s.c5()) return 1;
   if (c5()>s.c5()) return -1;
   return 0;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5>
```

```
bool X5<Y1,Y2,Y3,Y4,Y5>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   cpmp(x3);
   cpmp(x4);
   cpmp(x5);
   return true;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5>
bool X5<Y1,Y2,Y3,Y4,Y5>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   cpms(x3);
   cpms(x4);
   cpms(x5);
   return true;
}

//////////////////////////// class X6<> ////////////////////////////

template <class Y1, class Y2, class Y3, class Y4, class Y5, class Y6>
class  X6{ // heterotypic 6-tuples
protected:
   Y1 x1;
   Y2 x2;
   Y3 x3;
   Y4 x4;
   Y5 x5;
   Y6 x6;
   typedef X6<Y1,Y2,Y3,Y4,Y5,Y6> Type;
public:
   CPM_IO
   CPM_ORDER
   X6(Y1 const& x1_, Y2 const& x2_, Y3 const& x3_,
      Y4 const& x4_, Y5 const& x5_, Y6 const& x6_):x1(x1_),x2(x2_),
      x3(x3_), x4(x4_),x5(x5_),x6(x6_){}
   X6(void):x1(),x2(),x3(),x4(),x5(),x6(){}
   X6(const X6<Y1,Y2,Y3,Y4,Y5,Y6>& z): x1(z.x1), x2(z.x2),
      x3(z.x3),x4(z.x4),x5(z.x5),x6(z.x6){}

   Y1& c1(void){ return x1;}
   Y2& c2(void){ return x2;}
   Y3& c3(void){ return x3;}
   Y4& c4(void){ return x4;}
   Y5& c5(void){ return x5;}
   Y6& c6(void){ return x6;}
```

```
   Y1 const& c1(void)const{ return x1;}
   Y2 const& c2(void)const{ return x2;}
   Y3 const& c3(void)const{ return x3;}
   Y4 const& c4(void)const{ return x4;}
   Y5 const& c5(void)const{ return x5;}
   Y6 const& c6(void)const{ return x6;}

   Y1 first()const{ return x1;}
   Y2 second()const{ return x2;}
   Y3 third()const{ return x3;}
   Y4 fourth()const{ return x4;}
   Y5 fifth()const{ return x5;}
   Y6 sixth()const{ return x6;}
   Word nameOf()const
   {
       return Word("X6< "&
          CpmRoot::Name<Y1>()(Y1())&
          ","&
          CpmRoot::Name<Y2>()(Y2())&
          ","&
          CpmRoot::Name<Y3>()(Y3())&
          ","&
          CpmRoot::Name<Y4>()(Y4())&
          ","&
          CpmRoot::Name<Y5>()(Y5())&
          ","&
          CpmRoot::Name<Y6>()(Y6())&
          " >");
   }
};

template <class Y1, class Y2, class Y3, class Y4, class Y5, class Y6>
Z X6<Y1,Y2,Y3,Y4,Y5,Y6>::com(X6<Y1,Y2,Y3,Y4,Y5,Y6> const& s)const
{
   if (c1()<s.c1()) return 1;
   if (c1()>s.c1()) return -1;
   if (c2()<s.c2()) return 1;
   if (c2()>s.c2()) return -1;
   if (c3()<s.c3()) return 1;
   if (c3()>s.c3()) return -1;
   if (c4()<s.c4()) return 1;
   if (c4()>s.c4()) return -1;
   if (c5()<s.c5()) return 1;
   if (c5()>s.c5()) return -1;
   if (c6()<s.c6()) return 1;
   if (c6()>s.c6()) return -1;
   return 0;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5, class Y6>
```

```
bool X6<Y1,Y2,Y3,Y4,Y5,Y6>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   cpmp(x3);
   cpmp(x4);
   cpmp(x5);
   cpmp(x6);
   return true;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5, class Y6>
bool X6<Y1,Y2,Y3,Y4,Y5,Y6>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   cpms(x3);
   cpms(x4);
   cpms(x5);
   cpms(x6);
   return true;
}

//////////////////////////// class X7<> ////////////////////////////

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7>
class  X7{ // heterotypic 7-tuples
protected:
   Y1 x1;
   Y2 x2;
   Y3 x3;
   Y4 x4;
   Y5 x5;
   Y6 x6;
   Y7 x7;
   typedef X7<Y1,Y2,Y3,Y4,Y5,Y6,Y7> Type;
public:
   CPM_IO
   CPM_ORDER
   X7(Y1 const& x1_, Y2 const& x2_, Y3 const& x3_,
      Y4 const& x4_, Y5 const& x5_, Y6 const& x6_, Y7 const& x7_):
      x1(x1_),x2(x2_),x3(x3_),x4(x4_),x5(x5_),x6(x6_),x7(x7_){}
   X7(void):x1(),x2(),x3(),x4(),x5(),x6(),x7(){}
   X7(const X7<Y1,Y2,Y3,Y4,Y5,Y6,Y7>& z): x1(z.x1), x2(z.x2),
      x3(z.x3),x4(z.x4),x5(z.x5),x6(z.x6),x7(z.x7){}

   Y1& c1(void){ return x1;}
   Y2& c2(void){ return x2;}
   Y3& c3(void){ return x3;}
```

```
   Y4& c4(void){ return x4;}
   Y5& c5(void){ return x5;}
   Y6& c6(void){ return x6;}
   Y7& c7(void){ return x7;}

   Y1 const& c1(void)const{ return x1;}
   Y2 const& c2(void)const{ return x2;}
   Y3 const& c3(void)const{ return x3;}
   Y4 const& c4(void)const{ return x4;}
   Y5 const& c5(void)const{ return x5;}
   Y6 const& c6(void)const{ return x6;}
   Y7 const& c7(void)const{ return x7;}

   Y1 first()const{ return x1;}
   Y2 second()const{ return x2;}
   Y3 third()const{ return x3;}
   Y4 fourth()const{ return x4;}
   Y5 fifth()const{ return x5;}
   Y6 sixth()const{ return x6;}
   Y7 seventh()const{ return x7;}
   Word nameOf()const
   {
       return Word("X7< "&
          CpmRoot::Name<Y1>()(Y1())&
          ","&
          CpmRoot::Name<Y2>()(Y2())&
          ","&
          CpmRoot::Name<Y3>()(Y3())&
          ","&
          CpmRoot::Name<Y4>()(Y4())&
          ","&
          CpmRoot::Name<Y5>()(Y5())&
          ","&
          CpmRoot::Name<Y6>()(Y6())&
          ","&
          CpmRoot::Name<Y7>()(Y7())&
          " >");
   }
};

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7>
Z X7<Y1,Y2,Y3,Y4,Y5,Y6,Y7>::com(X7<Y1,Y2,Y3,Y4,Y5,Y6,Y7> const& s)const
{
   if (c1()<s.c1()) return 1;
   if (c1()>s.c1()) return -1;
   if (c2()<s.c2()) return 1;
   if (c2()>s.c2()) return -1;
   if (c3()<s.c3()) return 1;
   if (c3()>s.c3()) return -1;
```

```
   if (c4()<s.c4()) return 1;
   if (c4()>s.c4()) return -1;
   if (c5()<s.c5()) return 1;
   if (c5()>s.c5()) return -1;
   if (c6()<s.c6()) return 1;
   if (c6()>s.c6()) return -1;
   if (c7()<s.c7()) return 1;
   if (c7()>s.c7()) return -1;
   return 0;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7>
bool X7<Y1,Y2,Y3,Y4,Y5,Y6,Y7>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   cpmp(x3);
   cpmp(x4);
   cpmp(x5);
   cpmp(x6);
   cpmp(x7);
   return true;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7>
bool X7<Y1,Y2,Y3,Y4,Y5,Y6,Y7>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   cpms(x3);
   cpms(x4);
   cpms(x5);
   cpms(x6);
   cpms(x7);
   return true;
}

//////////////////////////// class X8<> ////////////////////////////

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7, class Y8>
class  X8{ // heterotypic 8-tuples
protected:
   Y1 x1;
   Y2 x2;
   Y3 x3;
   Y4 x4;
   Y5 x5;
```

```
   Y6 x6;
   Y7 x7;
   Y8 x8;
   typedef X8<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8> Type;
public:
   CPM_IO
   CPM_ORDER
   X8(Y1 const& x1_, Y2 const& x2_, Y3 const& x3_,
      Y4 const& x4_, Y5 const& x5_, Y6 const& x6_, Y7 const& x7_,
      Y8 const& x8_):
      x1(x1_),x2(x2_),x3(x3_),x4(x4_),x5(x5_),x6(x6_),x7(x7_),x8(x8_){}
   X8(void):x1(),x2(),x3(),x4(),x5(),x6(),x7(),x8(){}
   X8(const X8<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8>& z): x1(z.x1), x2(z.x2),
      x3(z.x3),x4(z.x4),x5(z.x5),x6(z.x6),x7(z.x7),x8(z.x8){}

   Y1& c1(void){ return x1;}
   Y2& c2(void){ return x2;}
   Y3& c3(void){ return x3;}
   Y4& c4(void){ return x4;}
   Y5& c5(void){ return x5;}
   Y6& c6(void){ return x6;}
   Y7& c7(void){ return x7;}
   Y8& c8(void){ return x8;}

   Y1 const& c1(void)const{ return x1;}
   Y2 const& c2(void)const{ return x2;}
   Y3 const& c3(void)const{ return x3;}
   Y4 const& c4(void)const{ return x4;}
   Y5 const& c5(void)const{ return x5;}
   Y6 const& c6(void)const{ return x6;}
   Y7 const& c7(void)const{ return x7;}
   Y8 const& c8(void)const{ return x8;}

   Y1 first()const{ return x1;}
   Y2 second()const{ return x2;}
   Y3 third()const{ return x3;}
   Y4 fourth()const{ return x4;}
   Y5 fifth()const{ return x5;}
   Y6 sixth()const{ return x6;}
   Y7 seventh()const{ return x7;}
   Y8 eighth()const{ return x8;}
   Word nameOf()const
   {
       return Word("X8< "&
          CpmRoot::Name<Y1>()(Y1())&
          ","&
          CpmRoot::Name<Y2>()(Y2())&
          ","&
          CpmRoot::Name<Y3>()(Y3())&
          ","&
```

```
          CpmRoot::Name<Y4>()(Y4())&
          ","&
          CpmRoot::Name<Y5>()(Y5())&
          ","&
          CpmRoot::Name<Y6>()(Y6())&
          ","&
          CpmRoot::Name<Y7>()(Y7())&
          ","&
          CpmRoot::Name<Y8>()(Y8())&
          " >");
   }
};

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7, class Y8>
Z X8<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8>::com(
   X8<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8> const& s)const
{
   if (c1()<s.c1()) return 1;
   if (c1()>s.c1()) return -1;
   if (c2()<s.c2()) return 1;
   if (c2()>s.c2()) return -1;
   if (c3()<s.c3()) return 1;
   if (c3()>s.c3()) return -1;
   if (c4()<s.c4()) return 1;
   if (c4()>s.c4()) return -1;
   if (c5()<s.c5()) return 1;
   if (c5()>s.c5()) return -1;
   if (c6()<s.c6()) return 1;
   if (c6()>s.c6()) return -1;
   if (c7()<s.c7()) return 1;
   if (c7()>s.c7()) return -1;
   if (c8()<s.c8()) return 1;
   if (c8()>s.c8()) return -1;
   return 0;
}

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7, class Y8>
bool X8<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8>::prnOn(ostream& str)const
{
   cpmp(x1);
   cpmp(x2);
   cpmp(x3);
   cpmp(x4);
   cpmp(x5);
   cpmp(x6);
   cpmp(x7);
   cpmp(x8);
   return true;
```

```
}

template <class Y1, class Y2, class Y3, class Y4, class Y5,
   class Y6, class Y7, class Y8>
bool X8<Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8>::scanFrom(istream& str)
{
   cpms(x1);
   cpms(x2);
   cpms(x3);
   cpms(x4);
   cpms(x5);
   cpms(x6);
   cpms(x7);
   cpms(x8);
   return true;
}

// In a context where X1,X2,X3,X4,X5,X6,X7,X8 are used as
// names for template arguments one could get in trouble
// by a statement
//
// using CpmArrays::X2;
//
// thus one should avoid this. Then the full name CpmArrays::X2 would
// be necessary  which is too long a name for this innocent construct.
// By the following definition we may use cpmX2, which follows the same
// notational scheme as cpmerror, cpmwrite, ...

#define cpmX2 CpmArrays::X2
#define cpmX3 CpmArrays::X3
#define cpmX4 CpmArrays::X4
#define cpmX5 CpmArrays::X5
#define cpmX6 CpmArrays::X6
#define cpmX7 CpmArrays::X7
#define cpmX8 CpmArrays::X8

} // CpmArrays

#endif
```

232 cpmzinterval.h

```
//? cpmzinterval.h
//? Status of work 2008-10-25.
//?
#ifndef CPM_ZINTERVAL_H_
#define CPM_ZINTERVAL_H_
/*
    Description:
      class for 'intervals' in Z

*/

#include <cpmsystem.h>

namespace CpmArrays{

   using CpmRoot::Z;
   using CpmRoot::R;
   using CpmRoot::Word;

//////////////////////// class IvZ ////////////////////////////////////////
// class IvZ will be used in the definition of the basic array template
// Vl<> which, in turn, will be used to define the set template S<>.
// So it is not possible at this point to relate IvZ with S<Z>
// although clearly each instance of IvZ determines uniquely
// an instance of S<Z>. The C++ classes IvZ and S<Z> are very
// different in implementation and efficiency. For instance
//    IvZ iv(1,10000);
// defines a 'set' of 10000 integers by memorizing just two
// integers (1 and 10001 in this case). Wheras
//    S<Z> s(10000);
// (which calls a private constructor and thus may appear only in
// member function definition code) allocates memory for
// 10000 integers. When Vl<> will have been defined, we can give
// as Vl<IvZ> an storage economic representation of arbitrary
// finite subsets of Z.
// That IvZ 'represents' subsets of Z expresses itself in
// the member function
//    bool hasElm(Z const& i)const
// which S<Z> defines with just the same signature.

class IvZ{// 'intervals of integers', i.e. contiguous finite subsets of Z
   // 'intervals of integers', i.e. contiguous finite subsets of Z
   // (void set included; this looks important since T-valued arrays
   // indexed by i's ranging over some instance of IvZ should turn
   // to be a generalization of V<T>). V<T> has void arrays, which
   // correspond to a void index set. No virtual functions!
   Z l_;
```

```
      // l for 'left', arbitrary integer
   Z r_;
      // r for 'right', integer >=l_
      // *this characterizes the subset of Z given by
      // { i \in Z | l_<=i<r_ }. Notice that l_==r_ characterizes
      // the (a ?) void set
   void order_(){ if (r_<l_){ Z temp=r_; r_=l_; l_=temp;} }
      // private mutating method to achieve a consistent state
   typedef IvZ Type;
public:
   CPM_IO
   CPM_ORDER
   IvZ():l_(0),r_(0){}
      // the void set
   IvZ(Z l, Z r, bool dummy):l_(l),r_(r){}
      // lean (unsave) construction giving l_ and r_ directly
      // Last argument only for having a distinction in signature
   IvZ(Z i, Z j):l_(i),r_(j){order_();r_++;}
      // the minimum contiguous subset of Z which contains both
      // i and j; no order restrictions to i,j. Notice that the
      // void set can't be obtained that way.
   IvZ(Word w, Z l, Z r):l_(l),r_(r){w;order_();}
      // construction giving l_ and r_ directly
      // First argument only for having a distinction in signature
   IvZ(R a, R b):l_(a<b ? (Z)::floor(a) : (Z)::floor(b)),
      r_(a<b ? (Z)::ceil(b) : (Z)::ceil(a)){r_++;}
      // construction of the smallest IvZ which contains a and b
   bool isVoid()const{ return l_==r_;}
   Z width()const{ return r_-l_;}
      // cardinality of the set given by *this
   Z card()const{ return r_-l_;}
      // cardinality of the set given by *this

// The intention behind inf() and sup() is that
// iv==IvZ(iv.inf(),iv.sup()) for every non-void iv \in IvZ
// Thus inf() and sup() have to stop if called for a void set.
   Z inf()const;
   Z sup()const;
   Z first()const{ return inf();}
   Z last()const{ return sup();}
   Z b()const{ return l_;}
      // begin
   Z n()const{ return r_;}
      // next, relative to the numbers belonging to *this
      // this is the next number with respect to standard order
   Z e()const{ return r_-1;}
      // end, actually the last number belonging to *this
      // if *this is non-void as a set. In any case it is defined
      // as r_-1.
      // Looping over an IvZ ivz can allways be done as
```

```
   // for (Z i=ivz.b();i<=ivz.e();++i) ... ivz[i]....
   // or, equally well
   // for (Z i=ivz.b();i<ivz.n();++i) ... ivz[i]....

Z operator[](Z i)const{ if (i<=1) return inf(); else return sup();}

Z relPos(Z i)const
   //: relative position
   // gives a code for the location of i relative to *this.
{
   if (i<inf()) return -1;
   else if (i>sup()) return 1;
   else return 0;
}

IvZ operator|(IvZ const& iv)const;
   // minimum interval that contains the union
   // join or l.u.b. (lowest upper bound) in lattice terminology

IvZ operator&(IvZ const& iv)const;
   // section, intersection
   // meet or g.l.b (greates lower bound) in lattice terminology.

bool hasElm(Z const& i)const{ return l_<=i && i<r_;}
   // has element
   // answers the question whether point i is an element of the
   // interval *this. S<Z> has a member function of the same
   // signature.

bool contains(const IvZ& iv)const;
   // says whether iv is a subset (true subset 'or equal') of *this
   // or not.

bool isSubsetOf(const IvZ& iv)const{  return iv.contains(*this);}
   // says whether *this is a subset (true subset 'or equal') of iv
   // or not.

IvZ operator+(Z s){return IvZ(l_+s,r_+s);}
   // shifts both ends of the interval by s
IvZ& operator+=(Z s){ l_+=s; r_+=s; return *this;}
IvZ& operator-=(Z s){ l_-=s; r_-=s; return *this;}
Word nameOf()const{ return "IvZ";}
Z ran(Z j=0)const;
   // random
   // Returns a 'randomly' selected element in *this
   // Here the argument j plays the same role as in
   //    R CpmRootX::randomR(Z j=0);
   // (see cpmtypes.h)
   // notice that the function ranVal(Z) would be expected
   // to return a IvZ, so the name ran is more appropriate
```

```
      // If *this is void, we return 0 and issue a warning.
   static IvZ byCar(Z e, Z quasiCard);
      //: by cardinality
      // returns an interval res with |quasiCard| elements
      // (thus res.isVoid()==true for quasiCard==0).
      // If quasiCard!=0, e is an element of res.
      // If quasiCard>0 all elements i of res satisfy i>=e
      // If quasiCard<0 all elements i of res satisfy i<=e
      // Notice that this is a static generation method and thus
      // does the job of a constructor. If one would implement it
      // as a constructor one would need an additional argument for
      // distiction from the IvZ(Z,Z) which already exists.
      // An advantage of static generation methods is that they can
      // be given an indicative name.
};

//////////////////////////// Implementation ////////////////////////////
// see also cpmzinterval.cpp

inline Z IvZ::inf()const
{
   if (isVoid()){
      cpmerror("Iv::inf() undefined for void instance");
      return 0; // never happens
   }
   else return l_;
}

inline Z IvZ::sup()const
{
   if (isVoid()){
      cpmerror("Iv::sup() undefined for void instance");
      return 0; // never happens
   }
   else return r_-1;
}

} // namespace

#endif
```

233 cpmzinterval.cpp

```
//? cpmzinterval.cpp
//? Status of work 2008-10-25.
//?
#include <cpmzinterval.h>
#include <cpmtypes.h>

using CpmRoot::Z;
using CpmArrays::IvZ;
using namespace CpmStd;

//////////////////////////// class IvZ ////////////////////////////////////

bool IvZ::contains(const IvZ& iv)const
{
   if (iv.isVoid()) return true;
      // the void set is subset of every set, even of the void set
   else{ // now iv is non-void
      if (isVoid()) return false;
         // a non-void set iv is never a subset of the void set
      else return l_<=iv.l_ && iv.r_<=r_;
   }
}

IvZ IvZ::operator|(IvZ const& iv)const
{
   if (iv.isVoid()) return *this;
      // the union with the void set is the original
   else{ // now iv is non-void
      if (isVoid()) return iv;
         // a non-void set iv is never a subset of the void set
      else { // now both *this and iv are non-void
         Z i1=inf();
         Z s1=sup();
         Z i2=iv.inf();
         Z s2=iv.sup();
         Z i=CpmRootX::inf<Z>(i1,i2);
         Z s=CpmRootX::sup<Z>(s1,s2);
         return IvZ(i,s);
      }
   }
}

IvZ IvZ::operator&(IvZ const& iv)const
{
   if (isVoid()||iv.isVoid()) return IvZ();
      // the section with the void set is the void set
   else{ // now both *this and iv are non-void
```

```
      Z s1=sup();
      Z i2=iv.inf();
      if (s1<i2) return IvZ();
      else{
         Z i1=inf();
         Z s2=iv.sup();
         if (i1>s2) return IvZ();
         else{ // now there is an overlap
             Z i=CpmRootX::sup<Z>(i1,i2);
             Z s=CpmRootX::inf<Z>(s1,s2);
             return IvZ(i,s);
         }
      }
   }
}

bool IvZ::prnOn(ostream& str)const
{
   cpmwt("IvZ");
   cpmp(l_);
   cpmp(r_);
   return true;
}

bool IvZ::scanFrom(istream& str)
{
   cpms(l_);
   cpms(r_);
   order_();
   return true;
}

Z IvZ::com(IvZ const& iv)const
{
   if (l_<iv.l_) return 1;
   if (l_>iv.l_) return -1;
   if (r_<iv.r_) return 1;
   if (r_>iv.r_) return -1;
   return 0;
}

Z IvZ::ran(Z j)const
{
   Z n=card();
   if (n==0){
      cpmwarning("IvZ::ran(Z j): interval is void, 0 returned");
      return 0;
   }
   else if (n==1){
      return l_;
```

```
   }
   else{
      CpmRoot::R r=CpmRoot::randomR(j);
      return l_+(Z)(r*n);
         // (Z)(r*n) takes values 0,1,...n-1 with equal probabilities.
         // These are n values, which is OK.
   }
}

IvZ IvZ::byCar(Z e, Z car)
{
// implementation by carefully using the lean constructor
   if (car>=0) return IvZ(e,e+car,true);
   else return IvZ(e+car+1,e+1,true);
      // notice that IvZ(e+car,e,true) does not contain e
}
```

234 pala.cpp

```
//? pala.cpp
//? Status of work 2008-10-25.
//?
/**********************************************************************
   pala.cpp

   Author: Ulrich Mutze
   Description: Application framework for programs of
   project pala

   History: Started 2005-03-15

**********************************************************************/
#include <cpmrigbodydyn.h>
#include <cpmsys2aps.h>
#include <cpmsysviewer.h>
#include <cpmactionprinciple.h>
   // these include cpmapplication.h

using CpmRoot::Z;
using CpmRoot::Word;
using CpmSystem::glue;
using CpmSystem::Message;
using CpmArrays::V;
using CpmFunctions::F;
using CpmApplication::Task;
   // a new (2005-04-23) powerful tool for implementing
   // menu-like interfaces to programs
using CpmProperties::SysControl;
using namespace CpmStd;

namespace{

// creating and running tasks is here defined as functions
// of a common generic signature which allows to combine
// them into complex selection and call structures (menus)

Z pala1(Word const& dummy)
// Notice that, by convention, an object of type F<Word,Z>
// is needed as the second argument of a Task-constructor.
// To define such an object is the reason for defining pala1().
// So it has to have a Word-typed argument for syntactic reasons.
{
   dummy;
   cpmmessage("pala1 started");
   cpmstatus("pala1 started",4);
   SysControl(); // see pala2c
```

```
   CpmPala1_3::App1RigBodyDyn a("pala1");
   a.setTitle("Rigid Body Dynamics");
   a.run();
   return 1;
}

Z pala2c(Word const& iniFile)
{
   Word loc("pala2c with ini-File "&iniFile);
   Z mL=1;
   CPM_MA
   using namespace CpmApplication;
   SysControl();
      // former tasks may have changed the static variables
      // in a way that differes from the state that
      // the present task experienced in independent
      // testing
   setDimApp(iniFile);
       // needed to initialize dimApp from ini-file
   if (dimApp==2){
      cpmmessage("dimApp==2");
      CpmPala2_2::Sys2Appl a(iniFile);
      a.setTitle("Particles and Walls");
      a.run();
   }
   else{
      cpmmessage("dimApp!=2");
      CpmPala2_3::Sys2Appl a(iniFile);
      a.setTitle("Particles and Walls");
      a.run();
   }
   CPM_MZ
   return 2;
}

Z pala3(Word const& dummy)
{
   dummy;
   cpmmessage("pala3 started");
   cpmstatus("pala3 started",4);
   SysControl(); // see pala2c
   CpmPalaAux::SysViewerAppl a("pala3");
   a.setTitle("Movie Viewer");
   a.run();
   cpmmessage("pala3 done");
   cpmstatus("pala3 done",4);
   return 3;
}

Z pala4(Word const& dummy)
```

```
// Started 2006-06-22
{
   dummy;
   cpmmessage("pala4 started");
   cpmstatus("pala4 started",4);
   SysControl();
   CpmPalaAction::ActionAppl a("pala4");
   a.setTitle("Action Principle");
   a.run();
   return 4;
}

}// namespace

Z CpmApplication::main_()
{
// we define tasks and combine them to a single one
   Z mL=0;
   Word loc("CpmApplication::main_()");
   CPM_MA
   CpmApplication::title_=Word("PaLa");
   Z nT=4;
   V< Task > vTask(nT);
   vTask[1]=Task("pala1",pala1);
   vTask[2]=Task::subMen("pala2",pala2c);
   vTask[3]=Task("pala3",pala3);
   vTask[4]=Task("pala4",pala4);
   Task rt=Task::men("pala",vTask);
// finally we execute them (by executing their combined version)
   Z res=rt.work();
   CPM_MZ
   return res;
}
```

235 smallexperiments.cpp

```
//? smallexperiments.cpp
//? Status of work 2008-10-25.
//?
// smallExperiments.cpp : Definiert den Einstiegspunkt fuer die
// Konsolenanwendung.

#include <cpmc.h>
#include <cpmdigraph.h>
#include <cpmfa.h>
#include <cpmapplication.h>
#include <cpmtestr.h>
#include <cpmrecordhandler.h>
#include <cpmmultiimage.h>

using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmFunctions;
using namespace CpmSystem;
using namespace CpmArrays;
using namespace CpmGraphics;
using namespace CpmImaging;
using namespace CpmApplication;
using namespace CpmGeo;
using namespace std;

R tWait=2;

void test1()
{
   cout<<"small experiment"<<endl;
   Vl<vector<C> > x(10);
   cout<<x.nameOf();
   Img24 img(500,500,rgb(20,250,0));
   img.setAntiAliasLines(1);
   img.fillEllipse(55,5,250,350,rgb(0,0,255));
   img.line(10,12,210,340,rgb(255,255,200));
   img.line(10,20,210,340,rgb(255,0,0));
   R2 p1(30,-2);
   R2 p2(15,28);
   R2 p3(25,28);
   img.fill(TriAng(p1,p2,p3),rgb(0,0,0));
   img.write();
   img.display(0,0);
   cpmwait(tWait,2);
}

R fa(Z i){return (i+1.)*(i+2.);}
```

```
Z fb(R x){ return cpmround(x);}

void test2()
// a single function f is used to construct a
// Vl<R> and a V<R>. Then the function is
// recovered from these vectors and the result is shown
// to coincide for the at the ends where the function
// value was stored in one of the vectors and not in the
// other. The graphical representation shows that this is
// actually the case.
{
   F<R,Z> f0(fb);
   F<Z,R> f(fa);
   Vl<R> v1(8,f);
   V<R>  v2(8,f);
   F<Z,R> g1=v1.fnc();
   F<Z,R> g2=v2.fnc();
   F<R,R> h1=f0&g1;
   F<R,R> h2=f0&g2;
   V< F<R,R> > vv(2);
   vv[1]=h1;
   vv[2]=h2;
   Graph gr;
   gr.setX(Iv(-2,12));
   gr.show(vv);
   cpmwait(tWait,2);
}

Z fi(Z i){ return (i-3)*(i-4);}
Z f0(Z i){ return i+20;}

void test3()
// testing the new constructor and the new chaining operation
// and stream interaction of class M<> and == operation.
// Everything works perfectly.
{
   Z n=12;
   V<Z> v(n,F<Z,Z>(fi));
   S<Z> s(v);
   M<Z,Z> m(s,F<Z,Z>(f0));
   Word ww("test3_out.txt");
   OFileStream out1(ww);
   out1()<<m;
   IFileStream in1(ww);
   M<Z,Z> m0;
   m0.scanFrom(in1());
   bool b=false;
   b=(m0==m);
   if (b) cout<<" writing and reading OK"<<endl;
   else cout<<" writing and reading failed"<<endl;
```

```
   M<Z,Z> x=m&m;
   cout<<"m="<<m;
   cout<<x;
   cpmwait(tWait,2);
}

//////////////////////////// test4() ////////////////////////////////

Z f1(Z i){ return (i-3)*(i-4);}
Z g1(Z i){ return i+20;}

Z f2(Z i){ return (i-3)*(i-4);}
Z g2(Z i){ return i+20;}

Z f3(Z i){ return (i-2)*(i-3);}
Z g3(Z i){ return i+27;}

void test4()
// behavioral standard test for strict value interface.
// works perfectly.
{
   Z n1=12;
   V<Z> v1(n1,F<Z,Z>(f1));
   S<Z> s1(v1);
   M<Z,Z> m1(s1,F<Z,Z>(g1));

   Z n2=13;
   V<Z> v2(n2,F<Z,Z>(f2));
   S<Z> s2(v2);
   M<Z,Z> m2(s2,F<Z,Z>(g2));

   Z n3=27;
   V<Z> v3(n3,F<Z,Z>(f3));
   S<Z> s3(v3);
   M<Z,Z> m3(s3,F<Z,Z>(g3));

   CpmTests::Sym< M<Z,Z> > s;
   cout<<"CpmTests::Sym gave "<<s(m1,m2)<<endl;

   CpmTests::DefaultConstructor< M<Z,Z> > dc;
   cout<<"CpmTests::DefaultConstructor gave "<<dc(m2)<<endl;

   CpmTests::CopyConstructor< M<Z,Z> > cc;
   cout<<"CpmTests::CopyConstructor gave "<<cc(m3)<<endl;

   CpmTests::Assignment< M<Z,Z> > as;
   cout<<"CpmTests::Assignment gave "<<as(m1,m2)<<endl;

   CpmTests::Assignment< M<Z,Z> > as2;
   cout<<"CpmTests::Assignment gave "<<as2(m3,m3)<<endl;
```

```
   CpmTests::StrictAssignment < M<Z,Z> > sas;
   cout<<"CpmTests::StrictAssignment gave "<<sas(m1,m2,m3)<<endl;

   cpmwait(tWait,2);
}

//////////////////////////// test5 ////////////////////////////////////////

void test5()
// to: A class template of value-like functions
{
   Z n=100;
   Z i0=27;
   vector< Fa<R,R> > Chebyshev(n);
   Fa<R,R> ACos(acos);
   Fa<R,R> Cos(cos);
   for (int i=0;i<n;++i) Chebyshev[i]=(ACos*i)&Cos;
   R xt=0.5;
   R y=Chebyshev[i0](xt);
   cout<<"y="<<y<< " -1 is OK "<<endl;
 /*  Graph gr;
   gr.setX(Iv(-1,1));
   gr.show(Chebyshev[i0],400); */
 //  cpmwait(2);
   R sum=0;
   vector< Fa<R,R> > ch=Chebyshev;
   vector< Fa<R,R> >::const_iterator j;
   for (j=ch.begin();j!=ch.end();++j) sum+=(*j)(xt);
   cout<<"sum="<<sum<<endl;
   Fa<R,R> fSum;
   for (j=ch.begin();j!=ch.end();++j) fSum+=*j;
   R sum1=fSum(xt);
   cout<<" sum - sum1 : "<<(sum-sum1)<<" should be 0 "<<endl;
   cpmwait(tWait,2);
}

//////////////////////////// test6 ////////////////////////////////////////

class A{
   R a_;
public:
   explicit A(R a=0):a_(a){}
   R f(Z n, R x, Z m=0)const
   { return ::pow(a_+x*n,m+1);}
};

R fAux(R const& x, A const& a, Z const& n)
{
   return a.f(n,x);
```

```
}

R fAux1(R const& x, A const& a, Z const& n, Z const& m)
{
   return a.f(n,x,m);
}

void test6()
{
   int dvn=6;
   vector<int> vn(dvn);
   vector<int>::iterator i;
   Z k=0;
   for (i=vn.begin();i!=vn.end();++i,++k) (*i)=k*k-5;
   A a0(2);

   vector <Fa<R,R> > vf;
   for (i=vn.begin();i!=vn.end();++i){
      vf.push_back(F3<R,A,int,int,R>(a0,*i,2)(fAux1));
   }
   Graph gr;
   V< F<R,R> > vfcpm((Z)vf.size());
   for (Z k=vfcpm.b();k<=vfcpm.e();++k) vfcpm[k]=vf[k-1];
   gr.show(vfcpm);
   cpmwait(tWait,2);
}

void test7()
{
   N n=99991;
   V<N> pf=CpmAlgorithms::prmFac(n);
   ostringstream ost;
   ost<<"prime factors of n="<<n<<" found as pf="<<pf<<endl;
   N n1=n;
   N n2=17;
   N g=CpmAlgorithms::gcd(n2,n1,true);
   ost<<"gcd("<<n2<<","<<n1<<") = "<<g<<endl;
   N ni=137;
   Word s("ulrich mutze");
   Word sr=s.rev();
   ost<<sr;
   cpmmessage(ost);
   cout<<Word(ost);
   cpmwait(tWait,2);
}

void test8()
{
   N ni=0, nf=99991;
   bool b;
```

```
   ostringstream ost;
   ost<<endl<<"Testing the numbers from "<<ni<<" to "<<nf<<
      " on primeness"<<endl;
   for (N i=ni;i<=nf;++i){
      b=CpmAlgorithms::isPrm(i);
      if (b) ost<<"number "<<i<<" is prime"<<endl;
   }
   cpmmessage(ost);
   cout<<Word(ost);
   cpmwait(tWait,2);
}

//////////////////// test9() ////////////////////////////////////////
void test9()
{
   using std::vector;
   int lin=3, col=5;
   vector< vector<double> > a(lin,vector<double>(col));
   a[2][4]=123.4;
   ostringstream ost;
   ost<<"a[2][4]="<<a[2][4];
   cpmmessage(ost);
   cout<<Word(ost);
   cpmwait(tWait,2);
}
//////////////////// test10() ////////////////////////////////////////
// Graph dynamics
// This is a purely 2D version of the functionality now provided
// uniformly for 2D and 3D in smallexperiments2.cpp test1().
class G{
   R fac_;
   R rot_;
   V<C> pos_;
   S<Z2> rep_;
   typedef G Type;
public:
   CPM_O
   CPM_NAM(G)
   G(V<Z2> const& vz2, R deg);
   void d_();
      // my D-operation as a mutating operation
   S<Z> ver()const;
      //: vertices
      // Set of vertices
   S<Z2> edg()const{ return rep_;}
      //:edges
   Z nVer()const{ return ver().card();}
   Z nEdg()const{ return edg().card();}
   R& fac(){ return fac_;}
   void mark(Graph& gr, R zoom=0.8, R sx=0, R sy=0)const;
```

```
};

G::G(V<Z2> const& vz2, R deg):fac_(1./3),rot_(deg*AngDeg),rep_(vz2)
{
   S<Z> ver;
   Z i;
   for (i=rep_.b();i<=rep_.e();++i){
      Z2 zi=rep_[i];
      ver.add(zi[1]);
      ver.add(zi[2]);
   }
   Z nv=ver.dim();
   pos_=V<C>(nv);
   if (nv>0){
      pos_[1]=C::expi(rot_);
      if (nv>1){
         R dphi=2*Pi/nv;
         C rot=C::expi(dphi);
         for (i=2;i<=nv;++i) pos_[i]=pos_[i-1]*rot;
      }
   }
   // The idea now is that vertex ver[i] has position pos_[i]
   // We have to see whether function d is able to maintain this
   // connection.
}

void G::d_()
{
// rep_ will become the new set of vertices. Thus we have to assign
// a position to each element of rep_. The elements of rep are
// internally numbered and this index will be used for numbering the
// positions.
   Z i,j,nr=rep_.dim();
   V<C> pos(nr);
   R fac1=1-fac_;
   for (i=1;i<=nr;++i){
      Z2 zi=rep_[i];
      Z2 zis(zi[2],zi[1]);
      Z lz=rep_.locate(zis);
      C c1=pos_[zi[1]];
      C c2=pos_[zi[2]];
      pos[i]=c1*fac1+c2*fac_;
   }
   pos_=pos;
   V<Z2> res;
   for (i=rep_.b();i<=rep_.e();++i){
      Z2 zi=rep_[i];
      for (j=rep_.b();j<=rep_.e();++j){
         Z2 zj=rep_[j];
         if (zi[2]==zj[1]){
```

```
            // then the new graph state has an edge from zi to zj
            // so that zi and zj are vertices of the new graph state
            // We need to associate positions with the vertices of
            // the new graph state in order to be able to draw the
            // new graph state.
            res<<Z2(i,j);
         }
      }
   }
   rep_=S<Z2>(res);
}

S<Z> G::ver()const
{
   S<Z> res;
   for (Z i=rep_.b();i<=rep_.e();++i){
      Z2 zi=rep_[i];
      res.add(zi[1]);
      res.add(zi[2]);
   }
   return res;
}

bool G::prnOn(ostream& str)const
{
   cpmwat;
   cpmp(rep_);
   cpmp(pos_);
   return true;
}

void G::mark(Graph& gr, R zoom, R sx, R sy)const
{
   Z n=nVer();
   if (n==0) return;
   gr.setAspRatHoldHeight(1);
      // clears the image content
   R saftyFac=1.1;
   Z method=0;
   gr.setXY(pos_,saftyFac,method);
   gr.shift(sx,sy);
   gr.scale(1./zoom);
   S<Z> ver;
   Z i;
   for (i=rep_.b();i<=rep_.e();++i){
      Z2 zi=rep_[i];
      Z z1=zi[1];
      Z z2=zi[2];
      ver.add(z1);
      ver.add(z2);
```

```
      C c1=pos_[z1];
      C c2=pos_[z2];
      C cm=(c1+c2)*0.5;
      gr.draw(c1,cm,RED);
      gr.draw(cm,c2,GREEN);
   }
   for (i=ver.b();i<=ver.e();++i){
      gr.mark(pos_[i],16,RED);
   }
}

V<Z2> firstGrowing()
{
   V<Z2> res(4);
   res[1]=Z2(1,2);
   res[2]=Z2(2,1);
   res[3]=Z2(1,3);
   res[4]=Z2(3,2);
   return res;
}

V<Z2> dieing()
{
   V<Z2> res(6);
   res[1]=Z2(1,2);
   res[2]=Z2(2,3);
   res[3]=Z2(2,4);
   res[4]=Z2(3,4);
   res[5]=Z2(3,5);
   res[6]=Z2(5,4);
   return res;
}

V<Z2> directedPair()
{
   V<Z2> res(5);
   res[1]=Z2(1,2);
   res[2]=Z2(2,1);
   res[3]=Z2(3,4);
   res[4]=Z2(4,3);
   res[5]=Z2(2,3);
   return res;
}

V<Z2> fourCycle()
{
   V<Z2> res(4);
   res[1]=Z2(1,2);
   res[2]=Z2(2,3);
   res[3]=Z2(3,4);
```

```
   res[4]=Z2(4,1);
   return res;
}

V<Z2> threePairs()
{
   V<Z2> res(6);
   res[1]=Z2(1,2);
   res[2]=Z2(2,1);
   res[3]=Z2(3,4);
   res[4]=Z2(4,3);
   res[5]=Z2(2,3);
   res[6]=Z2(3,2);
   return res;
}

V<Z2> twoTriplets()
{
   V<Z2> res(6);
   res[1]=Z2(1,2);
   res[2]=Z2(2,3);
   res[3]=Z2(3,1);
   res[4]=Z2(3,4);
   res[5]=Z2(4,5);
   res[6]=Z2(5,3);
   return res;
}

V<Z2> directedTriplets()
{
   V<Z2> res(7);
   res[1]=Z2(1,2);
   res[2]=Z2(2,3);
   res[3]=Z2(3,1);
   res[4]=Z2(3,4);
   res[5]=Z2(4,5);
   res[6]=Z2(5,6);
   res[7]=Z2(6,4);
   return res;
}

V<Z2> directedLoop()
{
   V<Z2> res(2);
   res[1]=Z2(1,2);
   res[2]=Z2(2,2);
   return res;
}

V<Z2> singleLoop()
```

```
{
   V<Z2> res(1);
   res[1]=Z2(1,1);
   return res;
}

void test10()
{
   Z n=2,sel;
   RecordHandler rch("test10.ini");
   bool write=false, debug=true;
   R tWait1=4,tWait=0.5,tWaitF=10,fac=0.25;
   R iniRotDegrees=0,zoom=0.8,sx=0,sy=0;
   V<Z> rep;
   Word sec("data");
   cpmrh(n);
   cpmrh(write);
   cpmrh(debug);
   cpmrh(sel);
   cpmrh(fac);
   cpmrh(iniRotDegrees);
   cpmrh(zoom);
   cpmrh(sx);
   cpmrh(sy);
   cpmrh(tWait1);
   cpmrh(tWait);
   cpmrh(tWaitF);
   V<Z2> vz2;
   if (sel==0){
      cpmrh(rep);
      Z dr=rep.dim();
      Z nr=dr/2;
      vz2=V<Z2>(nr);
      Z iR=1;
      for (Z k=1;k<=nr;++k){
         Z z1=rep[iR++];
         Z z2=rep[iR++];
         Z2 zk(z1,z2);
         vz2[k]=zk;
      }
   }
   else if (sel==1) vz2=firstGrowing();
   else if (sel==2) vz2=dieing();
   else if (sel==3) vz2=directedPair();
   else if (sel==4) vz2=threePairs();
   else if (sel==5) vz2=twoTriplets();
   else if (sel==6) vz2=directedTriplets();
   else if (sel==7) vz2=directedLoop();
   else if (sel==8) vz2=singleLoop();
   else             vz2=fourCycle();
```

```
   Graph gr;
   G g(vz2,iniRotDegrees);
   g.fac()=fac;
   R_Matrix m(3,n);
   R nInv=1./n;
   cpmprogress("loop",0);
   for (Z i=1;i<=n;++i){
      R prog=(i-1)*nInv;
  //    gr.clr();
      g.mark(gr,1+prog*(zoom-1),sx,sy);
      gr.dis(write);
      if (debug) cpmdebug(g);
      cpmprogress("loop",i*nInv);
      if (i==1 || i==n) cpmwait(tWait1,2); else cpmwait(tWait);
      m[1][i]=i;
      m[2][i]=g.nVer();
      m[3][i]=g.nEdg();
      if (i<n) g.d_();
   }
   gr.setAutoScale(1);
   gr.show(m);
   cpmwait(tWaitF,2);
}

void test11(void)
{
   Z i,n=11;
   R_Vector u(n);
   R_Vector t1(n);
   R_Vector t2(n);
   R_Vector t3(n-2);

   u[1]=5;
   u[2]=4.5;
   u[3]=4;
   u[4]=3.5;
   u[5]=3;
   u[6]=2.5;
   u[7]=2;
   u[8]=1.5;
   u[9]=1;
   u[10]=0.5;
   u[11]=0.3;

   t1[1]=0;
   t1[2]=8;
   t1[3]=17;
   t1[4]=28;
   t1[5]=39;
   t1[6]=54;
```

```
   t1[7]=70;
   t1[8]=93;
   t1[9]=124;
   t1[10]=174;
   t1[11]=210;

   t2[1]=0;
   t2[2]=4;
   t2[3]=9;
   t2[4]=16;
   t2[5]=20;
   t2[6]=27;
   t2[7]=35;
   t2[8]=47;
   t2[9]=63;
   t2[10]=90;
   t2[11]=110;

   t3[1]=0;
   t3[2]=17;
   t3[3]=35;
   t3[4]=56;
   t3[5]=80;
   t3[6]=109;
   t3[7]=144;
   t3[8]=190;
   t3[9]=252;

   R th=log(2.)*10e6*8e-6;
      // Halbwertszeit
   n=n-2; // fr krzere Tabelle
   V<R2> c1(n);
/***************
   Z nt=200;
   R ti=0;
   R tLast=210;
   R dt=(tLast-ti)/nt;
   V<R2> c2(nt);
   R U0=u[1];
   for (i=1;i<=nt;++i){
      c2[i]=R2(ti,U0*pow(0.5,ti/th));
      ti+=dt;
   }
*********************/
   for (i=1;i<=n;++i){
      R ti=t2[i];
      c1[i]=R2(ti,u[i]);
   }
   V< V<R2> > c("",c1/*,c2*/);
```

```
   Graph g;
   V<Color> vc("",BLACK,RED);
   g.mark(c,vc);
   V<Word> tt(3);
   tt[1]="x-Achse: Zeit in s, y-Achse: Spannung in V";
   tt[2]="10 MOhm in Reihe zum Voltmeter";
   tt[3]="9 Messwerte,  schwarz verbunden";
  // tt[3]="Die berechnete Kurve ist rot eingetragen.";
   g.addText(tt[1],0.5,0.6);
   g.addText(tt[2],0.5,0.55);
   g.addText(tt[3],0.5,0.50);
   g.dis(true);
   cpmwait(tWait,2);
}

void test12()
{
   // test of composing several images into one
   Word dr=Message::ext("../images");
   Word n11=glue(dr,"bussard.ppm");
   Word n12=glue(dr,"gnom.ppm");
   Word n21=glue(dr,"helmut_domes.ppm");
   Word n22=glue(dr,"kohlmeise1.ppm");

   VV<Img24> vv(2,2);
   vv[1][1]=Img24(n11);
   vv[1][2]=Img24(n12);
   vv[2][1]=Img24(n21);
   vv[2][2]=Img24(n22);
   Z m=750, n=1000;
   ImgR ir(vv,m,n);
   Graph g;
   ir.mark(g);
   g.dis(true);
   cpmwait(tWait,2);
}

void test13()
{
   // test of composing several images into one
   Word dr=Message::ext("../images");
   Word w1=glue(dr,"bussard.ppm");
   Word w2=glue(dr,"bussard.ppm");
   Word w3=glue(dr,"helmut_domes.ppm");
   Word w4=glue(dr,"gnom.ppm");
   Img24 im1(w1);
   Img24 im2(w2);
   Img24 im3(w3);
   Img24 im4(w4);
   Img24 im12=im1.pair(im2);
```

```
   Graph g;
   VV<Img24> vv(2,2);
   vv[1][1]=im1;
   vv[1][2]=im4;
   vv[2][1]=im3;
   vv[2][2]=im12;
   Z m=750, n=1000;
   MultiImage<Rh> mi(vv,m,n);
   mi.mark(g,1);
   g.dis();
   cpmwait(tWait,2);
}

Z CpmApplication::main_(void)
{
   Z mL=1;
   Z sel = args_.valInd(2) ? args_[2].toZ() : 13;
   Word loc("CpmApplication::main_(), sel="&cpm(sel));
   CPM_MA
   if (args_.valInd(3)) cpmverbose=args_[3].toZ();
   if (args_.valInd(4)) tWait=args_[4].toR();
   title_="smallexperiments";
   if (sel==1) test1();
   else if (sel==2) test2();
   else if (sel==3) test3();
   else if (sel==4) test4();
   else if (sel==5) test5();
   else if (sel==6) test6();
   else if (sel==7) test7();
   else if (sel==8) test8();
   else if (sel==8) test9();
   else if (sel==10) test10();
   else if (sel==11) test11();
   else if (sel==12) test12();
   else test13();
   CPM_MZ
   return 0;
}
```

236 smallexperiments2.cpp

```
//? smallexperiments2.cpp
//? Status of work 2008-10-25.
//?
// smallExperiments.cpp : Definiert den Einstiegspunkt für die
// Konsolenanwendung.
// This is a good and lean program structure for small but
// non-trivial experiments. Experiment with the compiler at hand
// so that topic swithch in function main_ can be set by
// re-compilation.
//
#include <cpmv.h>
#include <cpmc.h>
#include <cpmviewport.h>
#include <cpmimg24.h>
#include <cpmgraph.h>
#include <cpmdigraph.h>
#include <cpmfa.h>
#include <cpmapplication.h>
#include <cpmtestr.h>
#include <cpmrecordhandler.h>
#include <cpmdimdef.h>

using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmFunctions;
using namespace CpmSystem;
using namespace CpmArrays;
using namespace CpmGraphics;
using namespace CpmImaging;
using namespace CpmApplication;
using namespace CpmGeo;
using namespace std;

#include <cpmdim.h>
#include <cpmcamera.h>

#if defined(CPM_DIMDEF3)
   using namespace CpmDim3;
   using namespace CpmCamera3;
#else
   using namespace CpmDim2;
   using namespace CpmCamera2;
#endif

////////////////////////// test1() ////////////////////////////////////////////
// class for studying the iteration of line graphs
class G{ // G for graph, here actually directed graph (digraph)
```

```
   R fac_;
      // a parameter that controls the propagation of pos_
   V<Spc> pos_;
      // this is an iteresting additional structure; here for
      // visualization. The basic defining quantity for instances of
      // G is rep_ and not pos_
   S<Z2> rep_;
      // essential data
   typedef G Type;
public:
   CPM_O
   CPM_NAM(G)
   G(V<Z2> const& vz2, Camera& ca);
   void d_();
      // my D-operation as a mutating operation
      // in the literature concerning line graphs it is
      // normally denoted L (making L(G) out of G)
   S<Z> ver()const;
      //: vertices
      // Set of vertices
   S<Z2> edg()const{ return rep_;}
      //:edges
   Z nVer()const{ return ver().card();}
   Z nEdg()const{ return edg().card();}
   R& fac(){ return fac_;}
   X2<Spc,R> envSph()const{ return Spc::envSph(pos_);}
      //: envelopping sphere
   void mark(Camera& ca)const;
      //: mark
};

G::G(V<Z2> const& vz2, Camera& ca):fac_(1./3),rep_(vz2)
{
   S<Z> ver;
   Z i;
   for (i=rep_.b();i<=rep_.e();++i){
      Z2 zi=rep_[i];
      ver.add(zi[1]);
      ver.add(zi[2]);
   }
   Z nv=ver.dim();
   if (nv==4){
      pos_=Spc::tet();
   }
   else if (nv==5){
      R h4=::sqrt(6.0)/12.;
      Spc p1(0,0,h4);
      pos_=Spc::tet().prepend(p1);
   }
   else{
```

```
      V<Vec> hom=Vec::homUnitVec(nv);
      pos_=V<Spc>(nv);
      for (i=1;i<=nv;++i) pos_[i]+=hom[i];
   }
}

void G::d_()
{
// rep_ will become the new set of vertices. Thus we have to assign
// a position to each element of rep_. The elements of rep are
// internally numbered and this index will be used for numbering the
// positions.
   Z i,j,nr=rep_.dim();
   V<Spc> pos(nr);
   for (i=1;i<=nr;++i){
      Z2 zi=rep_[i];
      Z2 zis(zi[2],zi[1]);
      Spc p1=pos_[zi[1]];
      Spc p2=pos_[zi[2]];
      pos[i]=p1.versus(p2,fac_);
   }
   pos_=pos;
   V<Z2> res;
   for (i=rep_.b();i<=rep_.e();++i){
      Z2 zi=rep_[i];
      for (j=rep_.b();j<=rep_.e();++j){
         Z2 zj=rep_[j];
         if (zi[2]==zj[1]){
            // then the new graph state has an edge from zi to zj
            // so that zi and zj are vertices of the new graph state
            // We need to associate positions with the vertices of
            // the new graph state in order to be able to draw the
            // new graph state.
            res<<Z2(i,j);
         }
      }
   }
   rep_=S<Z2>(res);
}

S<Z> G::ver()const
{
   S<Z> res;
   for (Z i=rep_.b();i<=rep_.e();++i){
      Z2 zi=rep_[i];
      res.add(zi[1]);
      res.add(zi[2]);
   }
   return res;
}
```

```
bool G::prnOn(ostream& str)const
{
   cpmwat;
   cpmp(rep_);
   cpmp(pos_);
   return true;
}

void G::mark(Camera& ca)const
{
   const Angle an(30);
   const R relLen=0.25;
      // how to indicate the direction of edges
   const R r=0.01;
      // radius of 'reflexivity loop' ('Schlinge' in German)

   Vec e=ca.get_h();
   for (Z i=rep_.b();i<=rep_.e();++i){
      Z2 zi=rep_[i];
      Z z1=zi[1];
      Z z2=zi[2];
      Spc p1=pos_[z1];
      if (z1!=z2){
         Spc p2=pos_[z2];
         ca.mark(p1,p2);
         Spc pc=p1.cen(p2);
         Vec vc=(p2-pc)*relLen;
         AxVec ax=e.cross(vc);
         vc*=AxVec(an,ax);
         ca.mark(pc,pc+vc);
      }
      else{
         ca.mark(p1,r);
      }
   }
}

V<Z2> firstGrowing()
{
   V<Z2> res(4);
   res[1]=Z2(1,2);
   res[2]=Z2(2,1);
   res[3]=Z2(1,3);
   res[4]=Z2(3,2);
   return res;
}

V<Z2> dieing()
{
```

```
   V<Z2> res(6);
   res[1]=Z2(1,2);
   res[2]=Z2(2,3);
   res[3]=Z2(2,4);
   res[4]=Z2(3,4);
   res[5]=Z2(3,5);
   res[6]=Z2(5,4);
   return res;
}

V<Z2> directedPair()
{
   V<Z2> res(5);
   res[1]=Z2(1,2);
   res[2]=Z2(2,1);
   res[3]=Z2(3,4);
   res[4]=Z2(4,3);
   res[5]=Z2(2,3);
   return res;
}

V<Z2> fourCycle()
{
   V<Z2> res(4);
   res[1]=Z2(1,2);
   res[2]=Z2(2,3);
   res[3]=Z2(3,4);
   res[4]=Z2(4,1);
   return res;
}

V<Z2> threePairs()
{
   V<Z2> res(6);
   res[1]=Z2(1,2);
   res[2]=Z2(2,1);
   res[3]=Z2(3,4);
   res[4]=Z2(4,3);
   res[5]=Z2(2,3);
   res[6]=Z2(3,2);
   return res;
}

V<Z2> twoTriplets()
{
   V<Z2> res(6);
   res[1]=Z2(1,2);
   res[2]=Z2(2,3);
   res[3]=Z2(3,1);
   res[4]=Z2(3,4);
```

```
   res[5]=Z2(4,5);
   res[6]=Z2(5,3);
   return res;
}

V<Z2> directedTriplets()
{
   V<Z2> res(7);
   res[1]=Z2(1,2);
   res[2]=Z2(2,3);
   res[3]=Z2(3,1);
   res[4]=Z2(3,4);
   res[5]=Z2(4,5);
   res[6]=Z2(5,6);
   res[7]=Z2(6,4);
   return res;
}

V<Z2> directedLoop()
{
   V<Z2> res(2);
   res[1]=Z2(1,2);
   res[2]=Z2(2,2);
   return res;
}

V<Z2> singleLoop()
{
   V<Z2> res(1);
   res[1]=Z2(1,1);
   return res;
}


void test1()
{
   V<Word> fl("","./control/test1.ini");
   RecordHandler rch(fl);
   Z n=2,sel;
   bool write=false, debug=true;
   R tWait1=4,tWait=0.5,tWaitF=10,fac=0.25;
   R zoom=0.8,tilt=0,pan=0;
   R height,aspRat,viewingAngle,frac,convAngle;
   V<Z> rep;
   Word sec("data");
   cpmrh(n);
   cpmrh(write);
   cpmrh(debug);
   cpmrh(sel);
   cpmrh(fac);
```

```
   cpmrh(zoom);
   cpmrh(tilt);
   cpmrh(pan);
   cpmrh(tWait1);
   cpmrh(tWait);
   cpmrh(tWaitF);
   cpmrh(height);
   cpmrh(aspRat);
   cpmrh(viewingAngle);
   cpmrh(frac);
   cpmrh(convAngle);
   V<Z2> vz2;
   if (sel==0){
      cpmrh(rep); // reading a list of edges from ini-file
      Z dr=rep.dim();
      Z nr=dr/2;
      vz2=V<Z2>(nr);
      Z iR=1;
      for (Z k=1;k<=nr;++k){
         Z z1=rep[iR++];
         Z z2=rep[iR++];
         Z2 zk(z1,z2);
         vz2[k]=zk;
      }
   }
// using edge lists from coded functions
   else if (sel==1) vz2=firstGrowing();
   else if (sel==2) vz2=dieing();
   else if (sel==3) vz2=directedPair();
   else if (sel==4) vz2=threePairs();
   else if (sel==5) vz2=twoTriplets();
   else if (sel==6) vz2=directedTriplets();
   else if (sel==7) vz2=directedLoop();
   else if (sel==8) vz2=singleLoop();
   else             vz2=fourCycle();
   Graph gr;
   Camera ca(height,aspRat,Angle(viewingAngle),frac, Angle(convAngle));
   if (Spc::dimension()==2) ca.setColBac(Color(LIGHTBLUE));
   G g(vz2,ca);
   g.fac()=fac;
   R_Matrix m(3,n);
   R_Matrix scale(2,n);
   R nInv=1./n;
   cpmprogress("loop",0);

   for (Z i=1;i<=n;++i){
      R prog=(i-1)*nInv;
      X2<Spc,R> xsr=g.envSph();
      ca.dirTo_(xsr.c1(),xsr.c2(),1./zoom, Angle(tilt),Angle(pan));
      ca.clr();
```

```
      g.mark(ca);
      ca.dis(write);
      if (debug) cpmdebug(g);
      cpmprogress("loop",i*nInv);
      if (i==1 || i==n) cpmwait(tWait1,2); else cpmwait(tWait);
      m[1][i]=i;
      m[2][i]=g.nVer();
      m[3][i]=g.nEdg();
      scale[1][i]=i;
      scale[2][i]=g.envSph().c2();

      if (i<n) g.d_();
   }
   gr.clr();
   gr.setAutoScale(1);
   gr.setText("number of edges and vertices");
   gr.show(m);
   cpmwait(tWaitF,2);
   gr.setText("size of graph");
   gr.show(scale);
   cpmwait(tWaitF,2);
}

//////////////////////// test2 ////////////////////////////////////////
// Good example for a short program with a rather general interface.
// iterative solution of z^3=1
// Fractals

C fu(C const& z){ return z*(2./3.)+C(1)/(z*z*3);}
   // the function to be iterated

C pfu(C const& z, Z const& n, bool const& stab)
// Methods of iteration
{
   C zz=z;
   if (stab){
      for (Z i=1;i<=n;++i){
         C za=fu(zz);
         C zb=fu(za);
         zz=(za+zb)*0.5;
      }
   }
   else{
      for (Z i=1;i<=n;++i){
         zz=fu(zz);
      }
   }
   return zz;
}
```

```
void test2()
{
// selecting an input-file and converting it into data
   V<Word> fl("","./control/test2.ini");
   RecordHandler rch(fl);
// declaring of data which can hold the data that
// already exist in rch
   R x1,x2,y1,y2,gamma,tWait=20;
   Z p=1;
   bool stabilize=false,zeroIsZero=true;
// reading the data from rch
   Word sec("data");
   cpmrh(x1);
   cpmrh(x2);
   cpmrh(y1);
   cpmrh(y2);
   cpmrh(p);
   cpmrh(gamma);
   cpmrh(stabilize);
   cpmrh(zeroIsZero);
   cpmrh(tWait);

// doing the intended computations with the data
   Iv ivx(x1,x2);
   Iv ivy(y1,y2);
   Graph g;
   g.setAspRatHoldHeight(1);
   g.setX(ivx);
   g.setY(ivy);

   F<C,C> f=F2<C,Z,bool,C>(p,stabilize)(pfu);
   g.show(f,gamma,zeroIsZero);
   cpmwait(tWait,2);
}

//////////////////////// test3 ////////////////////////////////////////
// Motion of particles on the surface of a sphere
// This is a nice mechanical system, since the particles
// cannot escape observation. The configuration space is compact.
// Works in 2D and 3D according to the setting in cpmdimdef.h
// Is the testbed for the implementation of Vec::homUnitVec, where a
// simplified version (less adjustibilities) of this class system is
// employed

class Pb{ // class for defining single partices
   Vec x_;
      // position relative to the center of the sphere
   AxVec omg_;
      // omega, angular velocity with respect
      // to the center of the sphere
```

```
   R m_;
      // mass
   R q_;
      // 'charge'
   static R eps_;
      // epsilon, regularization parameter
public:
   CPM_NAM(Pb)
   Pb(Vec const&x, Vec const& v, R m=1, R q=1): x_(x.polDec().c2()),
      omg_(x_.cross(v)), m_(m), q_(q){}

   Pb(): x_(1,0,0), omg_(), m_(1), q_(1){}

   Vec x()const{ return x_;}
   AxVec omg(){ return omg_;}
   void rec_(AxVec const& torque, R dt){ omg_+=(torque*(dt/m_));}
      //: reaction
      // to a 'Drehstoß'
   Spc pos()const{ return Spc()+x_;}
   Vec v()const{ return x_.cross(omg_);}
   R m()const{ return m_;}
   R q()const{ return q_;}
   static void setEps(R eps){ eps_=eps;}

   AxVec tor(Pb const& p)const;
     // torque felt by *this due to the presence of particle p

   void step_(R dt);

   void stop_(){ omg_=AxVec();}

   void mark(Camera& ca)const{ ca.mark(pos());}

};

void Pb::step_(R dt)
{
   AxVec rot=omg_.exp(dt);
   x_*=rot;
}

R Pb::eps_=1e-4;

AxVec Pb::tor(Pb const& p)const
{
   Vec to_p=p.x_-x_;
   R r=to_p.normalize_();
   R fac=-q_*p.q_/(r*r+eps_);
   Vec f=to_p*fac;
   return x_.cross(f);
```

```
 }

class PbSys{ // particle system
   V<Pb> vp_;
public:
   Z np()const{ return vp_.dim();}
   PbSys(V<Pb> const& vp):vp_(vp){}
   void step_(R dt);
   void stop_();
   void mark(Camera& ca)const;
};

void PbSys::step_(R dt)
// direct midpoint integrator
{
   Z i,j,n=np();
   R tau=dt*0.5;

   for (i=1;i<=n;++i) vp_[i].step_(tau);

   for (i=1;i<=n;++i){
      AxVec tor_i;
      for (j=1;j<=n;++j){
         if (j!=i) tor_i+=vp_[i].tor(vp_[j]);
      }
      vp_[i].rec_(tor_i,dt);
   }

   for (i=1;i<=n;++i) vp_[i].step_(tau);
}

void PbSys::mark(Camera& ca)const
{
   for (Z i=vp_.b();i<=vp_.e();++i) vp_[i].mark(ca);
}

void PbSys::stop_()
{
   for (Z i=vp_.b();i<=vp_.e();++i) vp_[i].stop_();
}

void test3()
// Shows how particle which initially are distributed randomly over a
// sphere move into equilibrium positions with respect to their
// repulsive electrostatic (regularized) forces.
// Includes visualization.
{
   V<Word> fl("","./control/test3.ini");
   RecordHandler rch(fl);
   Camera ca;
```

```
   Z i,n=12,nSteps=500;
   Z stopPeriod=100;
   R dtFac=0.02, tWait1=1, eps;
// reading data from file
   Word sec("data");
   cpmrh(n);
   cpmrh(nSteps);
   cpmrh(stopPeriod);
   cpmrh(dtFac);
   cpmrh(tWait1);
   cpmrh(eps);

   R dt=dtFac/sqrt(1.*n);
   Pb::setEps(eps);
   V<Pb> p(n);
   for (i=1;i<=n;++i){
      Vec xi=Vec::ranUnitVec();
      Vec omgi=Vec();
      p[i]=Pb(xi,omgi);
   }
   PbSys ps(p);
   ca.dirTo_(Spc(),1,1.25,Angle(0),Angle(0));
   cpmdebug(Spc::dimension());
   if (Spc::dimension()==2){
      cpmdebug(Spc::dimension());
      ca.setColBac(Color(LIGHTBLUE));
   }
   R nStepsInv=1./nSteps;

   for (i=1;i<=nSteps;++i){
      ca.clr();
      ps.mark(ca);
      cpmprogress("loop",nStepsInv*i);
      ca.dis();
      if (i==1) cpmwait(tWait1);
      if (i%stopPeriod==0) ps.stop_();
      ps.step_(dt);
   }
}

//////////////////////////// test4 ////////////////////////////////////////
// Dirac equation on a graph-related Hilbert space


void test4()
{
   V<Word> fl("","./control/test4.ini");
   RecordHandler rch(fl);
}
```

```
//////////// application framework ////////////////////////////////////

Word CpmApplication::title_="se";

Z CpmApplication::main_(void)
{
   cpmverbose=1;
 //  test1();
  // test2();
  //  test3();
   test4();
   return 0;
}
```

237 test1a.cpp

```
//? test1a.cpp
//? Status of work 2008-10-25.
//?
// test1a.cpp : Defines the entry point for the console application.

#include <cpmc.h>
#include <cpmvr.h>
#include <cpmfr.h>
#include <cpmapplication.h>
#include <cpmsr.h>
#include <cpmq.h>
#include <cpmtestr.h>
#include <cpmsi.h>
#include <cpmlogic1.h>
#include <cpmrecord.h>

using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmSystem;
using namespace CpmArrays;
using namespace CpmFunctions;
using namespace CpmPhysics;
using namespace CpmTests;
using namespace CpmMathFound;

void testAdHoc()
{
   V<V<R> > x(3,V<R>(2));
   cpmdebug(x);
}

// input on the commandline after test1 an integer between 1 and 4
// for increasigly complicated test. If a second integer follows, this
// will be interpreted as 'complexity'-parameter (10 as default)
int main(int argc, char* argv[]) // g++ does not accept Z here
{
   const Z selDefault=16;
   const Z compDefault=12;
   cpmverbose=2;
   bool doAll=false;
   Z doCount=0;

   Z i;
   V<Word> arg=CpmApplication::comLine(argc,argv);

   Z sel= argc<2 ? selDefault:  arg[2].toZ();
   Z comp=argc<3 ? compDefault: arg[3].toZ();
```

```
cpmmessage("start");

if (sel==1||doAll){
   Test_r<Vr<R>,R> a(comp);
   doCount++;
}
if (sel==2||doAll){
   Test_r<Vr<C>,C> a(comp);
   doCount++;
}
if (sel==3||doAll){
   Test_r<Fr<C,R>,R> b(comp);
   doCount++;
}
if (sel==4||doAll){
   Test_r<Fr<Vr<C>,C>,C> b(comp);
   doCount++;
}
if (sel==5||doAll){
   Test_r<Vr< Fr<Q,C> >, Fr<Q,C> > b(comp);
   doCount++;
}
if (sel==6||doAll){
   Test_r<Si,R> b(comp,1,1);
   doCount++;
}
if (sel==7||doAll){
   // express Ohm in natural units
   Si v=VOLTAGE(); // = 1 Volt
   Si i=CURRENT(); // = 1 Ampere
   Si ohm=v/i; // = 1 Ohm
   Si res=ohm.fromSiToNat();
   cout<<res.toWord(); //
      // res.val =3.87405e-5 = 1/25813
      // qv[1]=0  corresponds to c
      // qv[2]=1  corresponds to h
      // qv[3]=0  corresponds to G
      // qv[4]=-2 corresponds to e
      // qv[5]=qv[6]=qv[7]=0,
      // Thus 1 Ohm = 1/25813 * h * e^-2 = 3.87405e-5 * h * e^-2
      // which agrees with Hering, Martin, Stohrer:
      // Physik für Ingenieure,  VDI 1988 p. 544 (8-45)
   doCount++;
}
if (sel==8||doAll){
   Test_c<C>(comp,1,0); // test of complex particularities
   doCount++;
}
if (sel==9||doAll){
```

```
      Si::PlanckUnits(); // more quantities expressed by
         // natural units
      doCount++;
   }
   if (sel==10||doAll){

      Prop x1(1);
      Prop x2("",true);
      Prop x3("",false);
      PropLogic p1(x1);
      PropLogic p2(x2);
      PropLogic p3(x3);
      PropLogic p4=p2.vel(p3);
      PropLogic p5=p4.iff(p1);
      Word name=p5.toWord();
      B b(p5.eval());
      Word ev=b.toWord();
      Word at=p5.getAtoms().toWord();
      cpmmessage("name of p5: "&name);
      cpmmessage("value of p5: "&ev&"; false is correct");
      cpmmessage("atoms of p5: "&at);
      doCount++;
   }
   if (sel==11||doAll){
      Prop x1(1);
      Prop x2("");
      PropLogic p1(x1);
      PropLogic p2(x2);
      PropLogic p3=p1.non().vel(p2);
      PropLogic p4=p1.implies(p2);
      PropLogic p5=p3.iff(p4);
      PropLogic p6=p3.iff(p4.non());
      B b5(p5.taut());
      B b6(p6.taut());
      cpmmessage("p5.taut() ? "&b5.toWord()&"; true is correct");
      cpmmessage("p6.taut() ? "&b6.toWord()&"; false is correct");
      doCount++;
   }
   if (sel==12||doAll){
      Prop x1(1);
      Prop x2("");
      Prop x3("");
      Prop x4("");
      PropLogic p1(x1);
      PropLogic p2(x2);
      PropLogic p3(x3);
      PropLogic p4(x4);
      V<PropLogic> prem(2);
      prem[1]=p1.et(p2);
      prem[2]=p3;
```

```
      PropLogic p5=p1.vel(p3);
      PropLogic p6=p4.et(p3);
      B b5(p5.implied(prem));
      B b6(p6.implied(prem));
      cpmmessage("p5.implied(prem) ? "&b5.toWord()&"; true is correct");
      cpmmessage("p6.implied(prem) ? "&b6.toWord()&"; false is correct");
      doCount++;
   }
   if (sel==13||doAll){
      Test_logic<PropLogic> b(comp);
      doCount++;
   }
   if (sel==14||doAll){
      Test_set<Sr<Z>,Z> b(comp);
      doCount++;
   }
   if (sel==15||doAll){
      Test_set< Sr< Vr<Z> >, Vr<Z> > b(comp);
         // corresponding test for Vr<C> creates error. random generator
         // of Vr<> should be skrutinized
         // Vr<Z> seems not to work with suitable test elements
      doCount++;
   }
   if (sel==16||doAll){
      testAdoc();
      doCount++;
   }
   for (i=1;i<=argc;i++) cout<<endl<<"arg["<<i<<"]="
      <<arg[i]<<endl;
   cout<<doCount<<" tests of 15 done"<<endl;
   cpmmessage("end");
   char quit = '\0', quit0=quit;
   while (quit == quit0)
   {
        cout<<endl<<"If nothing can be seen in the console window"<<endl;
        cout<<"set the text color in the property menue" << endl;
        cout<<"of this window (for all windows of the same name)."<<endl;
        cout<<"Input any character to quit" << endl;
        cin >> quit;
   }
   return 0;
}
```

238 test1b.cpp

```
//? test1b.cpp
//? Status of work 2008-10-25.
//?
#include <iostream>

#include <cpmv.h>
#include <cpmp.h>

using namespace CpmArrays;
using namespace CpmRoot;
using namespace std;

// Theme is clarification of behavior of pointers
// and of polymorphism. Was named pointers.cpp till
// 2002-12-22

///// proper initializations of pointers ////////////

void test1(void)
{
   cout<<endl<<"Initialization of pointer by instance";
   R* rp;
   R a=45;
   rp=&a;
   cout<<endl<<*rp<<" should equal "<<a;
}

/**************** illegal and useless *******************

void test2(void)
{
   cout<<endl<<"Initialization of pointer by instance, alternative\
    method";
   cout<<"Actually illegal (2000-8-29)";
   R a=45;
   R* rp;
   *rp=a;
   cout<<endl<<*rp<<" should equal "<<a;
}

*******************************************************/

void test3(void)
{
   cout<<endl<<"Initialization of reference by instance";
   R a=55;
   R& rr=a;
```

```
   cout<<endl<<rr<<" should equal "<<a;
   R* rp;
   rp=&rr;
   cout<<endl<<*rp<<" should equal "<<rr;
   R* rp2;
   rp2=&a; // OK
   cout<<endl<<*rp2<<" should equal "<<a;
   rp2=&rr; // compiles, runs OK
   cout<<endl<<*rp2<<" should equal "<<rr;
   *rp2=rr; // runs now OK
   cout<<endl<<*rp2<<" should equal "<<rr;
   cout<<endl<<"end of test3()"<<endl;
}

void test4(void)
// Shows that vector components, although being references
// do not create problems when becoming undefined by leaving
// their scope.
{
   cout<<endl<<"no reference catastrophe";
   R a=0;
   {
      V<R> v(10,3.3456);
      a=v[5]; // one could think that 'a' would
      // be bound to the reference v[5] which becomes undefined
      // after leaving the block. However, a is a 'value' and
      // assigning a reference to it means copying the value
      // to which the reference pointed at the moment of assignement
      // into the value of a.
   }
   cout<<endl<<a<<" expectation is "<<3.3456;
   cout<<endl<<"end of test4()"<<endl;
}

void test5(void)
{
   cout<<endl<<"reference catastrophe, undefined reference";
   // actually the reference catastrophe does not (always?) happen
   R* ap;
   {
      V<R> v(10,3.3456);
      ap=&v[5];
   }
   // the address stored in ap, no longer marks a memory area
   // holding the value 3.3456, since v became destructed in leaving
   // the block.
   cout<<endl<<*ap<<" naive expectation is "<<3.3456;
   cout<<endl<<"end of test5()"<<endl;
}
```

```
void test6(void)
// Why there is a run-time problem ?
// 2000-8-29 No longer a problem !!!!!!!
{
   cout<<endl<<"no reference catastrophe, no runtime acess failure";
   R* ap;
   {
      V<R> v(10,3.3456);
      //*ap=v[5]; // creates run-time error
      // the following works:
      R x=v[5];
      ap=&x; // interestingly, *ap=x also creates run-time error,
         // presently I don't understand this
         // 2000-8-29: every C programmer should understand this;
         // *ap=x operates on a non-initialized pointer !!!
         // only ap=... initializes it.
   }
   cout<<endl<<*ap<<" expectation is "<<3.3456;
   cout<<endl<<"end of test6()"<<endl;
}

/// polymorphism of non-member functions ///////////
/////////////// hidden virtuality /////////////////

class X{
protected:
   Z i;
public:
   virtual X* clone()const{ return new X(*this);}
   virtual Z eval()const{return i;}
   virtual Word nameOf()const{ return Word("X");}
   X(Z i_=0):i(i_){}
   Z const& get_i()const{ return i;}

   friend X operator+(const X& a, const X& b)
      // do not define inline; would not be recognized as
      // defined in the body of test7()
   { return X(a.eval()+b.eval());} // does not work
      // with the compiler Visual C++ 7.xxx it does work!

    friend void priX(const X& x){ cout<<endl<<"X "<<x.i;}
    friend ostream& operator <<(ostream& out, X const& x)
    { out<<"X: "<<x.i<<endl; return out;}
};

/*************** definition has to be outside the class ******/
// would we have defined a function plus() instead of
// operator + the inline definition in the class would have worked too
// compiler bug ?????
// "Stroustrup. p. 279 Like a member declaration, a friend declaration
```

```
// does not introduce a name into an enclosing scope'
// This indicates where the compiler builders may have got their
// misconception from.

//X operator+(const X& a, const X& b)
// function is 'hidden virtual' since it calls a
// virtual function. If the argument is Y&, Y::eval()
// gets called as it should be !!!!
//   { return X(a.eval()+b.eval());}
/***************/

class Y: public X{
public:
   X* clone()const{ return new Y(*this);} // support for using
   // the polymorphic array Vp
   Word nameOf()const{ return Word("Y");}
   Z eval()const{return 2*i;}
   Y(Z i_=0):X(i_){}
   friend ostream& operator <<(ostream& out, Y const& y){ out<<
      "Y: "<<y.get_i()<<endl; return out;}
};

class Zx: public X{ // don't use Z since this is an alias of int
public:
   X* clone()const{ return new Zx(*this);}
   Word nameOf()const{ return Word("Zx");}
   Z eval()const{return 3*i;}
   Zx(Z i_=0):X(i_){}
   friend ostream& operator <<(ostream& out, Zx const& z){ out<<
      "Zx: "<<z.get_i()<<endl; return out;}
};

void test7()
{
   Y y1(10);
   Y y2(20);
   cout<<endl<<"y1+y2=";
   priX(y1+y2);
   cout<<endl<<"60 expected"<<endl;
   X xa(13);
   Y ya(137);
   cout<<xa<<ya<<endl;
   cout<<endl<<"end of test7()"<<endl;
}

// polymorphism and iteration

void test8()
{
   X a1(1);
```

```
   Y a2(2);
   Zx a3(3);
   X a4(4);
   cout<<endl<<"Direct calls to overloaded operator << :"<<endl;
   cout<<a1<<a2<<a3<<a4;
      // This calls for each ai the right operator. Notice that
      // this is no loop!
   cout<<endl<<"Direct calls to virtual function eval() :"<<endl;
   cout<<a1.eval()<<" "<<a2.eval()<<" "<<a3.eval()<<" "<<a4.eval()<<endl;
// Looping:
   Z n=4,i;
   Vp<X> vp(n); // polymorphic array, referring to clone base X
   // notice the canonical assignment to the components
   vp[1]=a1;
   vp[2]=a2;
   vp[3]=a3;
   vp[4]=a4;
   cout<<endl<<"Iteration over calls to overloaded operator <<"<<endl;
   for (i=1;i<=n;i++){
      cout<<vp[i]() ; // notice '()' for getting to the
      // value of the component
   }
   cout<<endl<<"Iteration over calls to virtual function eval"<<endl;
   for (i=1;i<=n;i++){
      cout<<vp[i]().eval()<<" ";
      // notice '()' for getting to the
      // value of the component
   }
   cout<<endl<<"end of test8()"<<endl;
   // virtual function mechanism works

   // For the overloaded operator '<<' always the version of base X
   // is called. This is a bit strange since the compiler has all
   // information at hand to choose the right overloaded version
   // depending on i. But this is would ask the compiler to understand
   // loops and function binding in a way he is not used to.
   // Actually, the compiler has to use the same function body for
   // operator<< for the whole loop. Doing the binding based on the value
   //
   // of i, would make compilation a highly non-local task that could
   // take extremely long time if n is large. So it is no question that
   // one needs late binding for the correct selection and this can
   // be achieved by no other means than virtual functions
   // (C++ is that way).

   // As a result: a usefull class hierarchy has to do common
   // infra-structure such as streaming and comparison via call to
   // virtual functions just as it is done in C+-.
   // Simply overloading friend functions does not work due to the C++
   // grammar.
```

```
   // This is to be remembered!
   // In the past I have lost confidence in the MS compilers not to
   // mishandle friend functions and thus banished them from the C+-
   // framework. Here I learn (what I knew several times already), that
   // even if they are handled properly, they are nor a sound basis for
   // polymorphic programming.
}

void test9()
// testing the new class CpmRoot::Name<>
// which replaces function template function CpmRoot::nameOf
// since 2006-12-23. The advantage of the new method is that
// one can defines names also for non-cpm classes such as the
// std containers.
{
   X x(137);
   cout<<endl<<x.nameOf();
   Name<X> nx;
   cout<<endl<<nx(x);
   Y y(137);
   cout<<endl<<x.nameOf();
   Name<Y> ny;
   cout<<endl<<ny(y);
   Name< vector<X> > nvx;
   vector<X> vxx(3,x);
   cout<<endl<<nvx(vxx);
   cout<<endl<<"end of test9()"<<endl;
}

int main(int argc, char *argv[])
{
   cout<<endl<<" main started ";
   test1();
   //test2();
   test3();
   test4();
   test5();
   test6(); //serious runtime problem, no longer !!!!
   test7();
   test8();
   test9();
   cout<<endl<<" main done ";
   char quit = '\0', quit0=quit;
   while (quit == quit0)
   {
        cout <<endl<<"If nothing can be seen in the console window"<<endl\
        ;
        cout << "set the text color in the property menue" << endl;
        cout << "of this window (for all windows of the same name"<<endl;
        cout << "Press any key to quit " << endl;
```

```
        cin >> quit;
    }

    return 1;
}
```

239 testcpm0.cpp

```
//? testcpm0.cpp
//? Status of work 2008-10-25.
//?
// Here main() is defined directly, without making use of
// stuff in CpmApplication. Especially, no data come from an ini-file.
// They are either coded default values or come from the command line.

#include <cpmtestr.h>

using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmSystem;
using namespace CpmArrays;
using namespace CpmFunctions;
using namespace CpmTests;
using namespace std;



void testAdHoc()
{
 //  V<vector< Vl<C> > > x(3,vector< Vl<C> >(2, Vl<C>(7)));
   V<V<R> > x(3,V<R>(2));
   cpmdebug(x);
}

void info()
{
   cout<<endl<<"Takes one or two integer arguments.";
   cout<<endl<<"The first in {0,1,...11}, the second arbitrary.";
   cout<<endl<<"The first argument selects the test function to be run, ";
   cout<<endl<<"where 0 means that all 11 function run in succession.";
   cout<<endl<<"The topics are: 1-6: Test_r, 7: Test_c,";
   cout<<endl<<"8-10: polymorphism, 11: Test_F ";
   cout<<endl<<"The second argument sets the computational complexity";
   cout<<endl<<"and thus the execution time. Default value is 24.";
}

int main(int argc, char* argv[])
// traditional C-main
{
   Z i, doCount=0, sel=30, comp=24;
   bool doAll=true;
   //bool doAll=false;
   V<Word> arg=comLine(argc,argv);

   if (arg.valInd(2)){
```

```
    Word w1=arg[2];
    if (w1=="?"){
         info();
         return 0;
    }
    else{
      sel=arg[2].toZ();
      if (sel!=0) doAll=false;
   }
}
if (arg.valInd(3)) comp=arg[3].toZ();
   // arg[1] is the name of the executable

R errorSum=0;

cpmmessage("start");

if (sel==1||doAll){
   Test_r<Vr<R>,R> a(comp,1,1);
   errorSum+=a.val();
   doCount++;
}
if (sel==2||doAll){
   Test_r<Vr<C>,C> a(comp,1,1);
   errorSum+=a.val();
   doCount++;
}
if (sel==3||doAll){
   Test_r<Fr<C,R>,R> b(comp,1,0);
   errorSum+=b.val();
   doCount++;
}
if (sel==4||doAll){
   Test_r<Fr<Vr<C>,R>,R> b(comp,1,0);
   errorSum+=b.val();
   doCount++;
}
if (sel==5||doAll){
   Test_c<C> b(comp,1,1); // test of complex particularities
   errorSum+=b.val();
   doCount++;
}
if (sel==6||doAll){
   Test_set<Sr<Z>,Z> b(comp,1,1);
   errorSum+=b.val();
   doCount++;
}
if (sel==7||doAll){
   typedef  Vr<C> Ty;
   Test_set< Sr< Ty >, Ty > b(comp,1,1);
```

```
      errorSum+=b.val();
      doCount++;
   }
   if (sel==8||doAll){
      Vo<C> v(13,C(0,5));
      Vr<C> w(12, C(4,1));
      TestOfPolymorphism< Vo<C>, Vr<C>, Pp< Vo<C> >  > top(v,w);
      doCount++;
   }
   if (sel==9||doAll){
      Test_Vp< V<C>, Vo<C>, Va<C> , Vr<C>  >  b(comp,1);
      errorSum+=b.val();
      doCount++;
   }
   if (sel==10||doAll){
      Test_Vp< F<C,R>, Fo<C,R>, Fa<C,R> , Fr<C,R>  >  b(comp,1);
      errorSum+=b.val();
      doCount++;
   }
   if (sel==11||doAll){
      Test_F< Z, C, Vr<R>, R  >  b(comp);
      errorSum+=b.val();
      doCount++;
   }
   if (sel==12||doAll){
      testAdHoc();
   }
   cout<<endl;
   for (i=arg.b();i<=arg.e();++i) cout<<"arg["<<i<<"]="<<arg[i];
   cout<<doCount<<" tests of 11 done"<<endl;
   cpmmessage("end");
   ostringstream ost;
   ost<<" Total error sum ="<<errorSum<<"  end";
   cpmmessage(ost,-1);
   ostringstream ost2;
   ost2<<endl<<"If nothing can be seen in the console window"<<endl;
   ost2<<"set the text color in the property menue" << endl;
   ost2<<"of this window (for all windows of the same name)."<<endl;
   cpmmessage(ost2);
   char quit = '\0', quit0=quit;
   while (quit == quit0)
   {
      cout<<"Input any character to quit" << endl;
      cin >> quit;
   }
   return 0;
}
```

240 vqm2.cpp

```
//? vqm2.cpp
//? Status of work 2008-10-25.
//?
/*********************************************************************
   vqm2.cpp

   Author: Ulrich Mutze
   Description: 'Visual Quantum Mechanics'
      // Version of the program that comes close in
      // structure to the myexcel2 program which seems to
      // be a practical compromise between flexibility and
      // transparency of structure
   History:
*********************************************************************/

#include <vqm2app1.h>
#include <vqm2app2.h>
#include <vqm2app3.h>
#include <vqm2.h>

using namespace CpmRoot;
using namespace CpmSystem;
using namespace CpmApplication;
using namespace CpmVQM2;

Z CpmApplication::main_(void)
// main program from a functional point of view. The actual basic
// application main(), or some Windows equivalent, is presently defined
// cpmapplication.cpp.
{
   cpmmessage("main_() Begin");
   cpmverbose=2;
   using namespace CpmApplication;
   title_="vqm2";
   Z sel = args_.valInd(2) ? args_[2].toZ() : 1;
   cpmverbose = args_.valInd(3) ? args_[3].toZ() : 2;
   R tWait = args_.valInd(4) ? args_[4].toR() : 1;
   cpmmessage("sel="&cpmwrite(sel));
   cpmmessage("cpmverbose="&cpmwrite(cpmverbose));
   Z nn=namList.dim();
   if (sel<1) cpmerror("main_(): sel<1");
   else if (sel>nn){
      cpmmessage("nothing done");
      cpmwait(2);
   }
   else appList[sel]();
      // this calls the selected program
```

```
   cpmwait(tWait);
   cpmmessage("main_() End");
   return 0;
}
```

241 vqm2app1.cpp

```
//? vqm2app1.cpp
//? Status of work 2008-10-25.
//?
/*********************************************************************
   vqm2app1.cpp
*********************************************************************/
#include <vqm2app1.h>
#include <vqm2.h>

#include <cpmframes.h>
#include <cpmgreg.h>
#include <cpmrfuncrec.h>
#include <cpmoperator.h>
#include <cpmtrj.h>
#include <cpmfieldoperator.h>
#include <cpmstate2.h>

using namespace CpmSystem;
using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmArrays;
using namespace CpmFunctions;
using namespace CpmGraphics;
using namespace CpmImaging;
using namespace CpmQuantumMechanics;
using CpmAlgorithms::CyclicAlarm;
using CpmTime::Greg;

//////////////////////// App1 ////////////////////////////////

void CpmVQM2::App1::doTheWork()
{
   Z mL=1;
   Word loc("CpmVQM2::App1::doTheWork()");
   CPM_MA
   Word sec("data");
   R gamma,tWait;
   cpmrh(gamma);
   cpmrh(tWait);
   Color::setGamma(gamma);
   CpmGraphics::Graph g;
   Z d1=60;
   Z d2=60;
   Vl< Vl<C> > vcc(d1,Vl<C>(d2));
   Z d1h=d1/2;
   Z d2h=d2/2;
   for (Z i=0;i<d1;i++){
```

```
      for (Z j=0;j<d2;j++){
         C val(i-d1h,j-d2h);
         vcc[i][j]=val;
      }
   }
   g.addText("Colors corresponding to complex numbers");
   g.mark(vcc,Iv(),1,true,false);
   g.display();
   cpmwait(tWait);
   CPM_MZ
}

////////////////////////////// App2 ////////////////////////////
// new version devoted to study of dephasing and localization
// golf problem
// radical changes and simplifications of App2 allowed since the
// old version is stored


namespace{

R jitterFunc(C const& c, R_Func const& f, R const& jt, R const& omega )
{
   R x=c[1];
   R t=c[2];
   R res=f(x);
   if (x<=0) return res;
   else{
      R shift=::cos(t*omega)*jt;
      return res+shift;
   }
}

} // namespace

void CpmVQM2::App2::doTheWork()
{
   Z mL=1;
   Word loc("CpmVQM2::App2::doTheWork()");
   CPM_MA

   bool useSI,standardMomentum,cutMomentum,commutator,doublicate,
      buildSpcObj, enfSym,test,showSpcAsBars,
      fileObs,fileSpeObj,filePsi,
      spcObjFromFile,psiFromFile,
      useSplitBoost,useRelCplCon;

   Z d,iterMax,buildT,buildV,buildS,buildR, buildPsi,
      ipolOrd,ipolMet,modeOfDyn;
```

```
   R s,mass,kp,ks1,ks2,ks3,kr,boostRel,gamma,eps,eps1,eps2,
      polyFac,bernsteinFac,bezierFac,ipolFac,gaussFac,indFac,
      bernsteinX,bernsteinY,showObs,showSpcObj,showSpc,showPsi,
      phiStep;

   Word file;
   V<Word> line1;
   V<Word> line2;
   V<Word> line3;
   V<Word> line4;

   V<Z> cN;

   V<R> waitTimes,poly,bernstein,bezierX,bezierY,ipolX,ipolY,
      gaussX,gaussSigma,gaussY,indX,indWidth,indY,cRe,cIm;
      // notice that by far the most data come from sections
      // potential, state1, ..., state3 which do not shine up
      // here explicitely

// reading data
// I choose to read all data in succession although for most settings
// in run_contol only a part of these data is needed in the code
// to be executed furtheron. Letting each control path read only those
// data that it really needs would create complicated interdependencies
// of the various control paths, resulting in the need for doublicating
// code. This also would complicate further extensions.
//////////////////////////////////////////////////////////////////////
   Word sec="about";
//////////////////////////////////////////////////////////////////////
   cpmrh(file);
   cpmrh(line1);
   cpmrh(line2);
   cpmrh(line3);
   cpmrh(line4);

//////////////////////////////////////////////////////////////////////
   sec="run preparation";
//////////////////////////////////////////////////////////////////////
   cpmrh(waitTimes);
   cpmrh(gamma);
   cpmrh(fileObs);
   cpmrh(fileSpeObj);
   cpmrh(filePsi);
   cpmrh(spcObjFromFile);
   cpmrh(psiFromFile);
   cpmrh(showObs);
   cpmrh(showSpcObj);
   cpmrh(showSpc);
   cpmrh(showPsi);
   cpmrh(showSpcAsBars);
```

```
   cpmrh(phiStep);
   cpmrh(modeOfDyn);

////////////////////////////////////////////////////////////////////////
   sec="system data";
////////////////////////////////////////////////////////////////////////
   cpmrh(d);
   cpmrh(s);
   cpmrh(mass);
   cpmrh(useSI);
   cpmrh(doublicate);
   cpmrh(buildSpcObj);
   cpmrh(commutator);
   cpmrh(buildT);
   cpmrh(buildV);
   cpmrh(buildS);
   cpmrh(buildR);
   cpmrh(buildPsi);
   cpmrh(useRelCplCon);
   cpmrh(kp);
   cpmrh(ks1);
   cpmrh(ks2);
   cpmrh(ks3);
   cpmrh(kr);
   cpmrh(boostRel);
   cpmrh(useSplitBoost);

////////////////////////////////////////////////////////////////////////
   sec="kinetic energy";
////////////////////////////////////////////////////////////////////////
   cpmrh(standardMomentum);
   cpmrh(cutMomentum);

////////////////////////////////////////////////////////////////////////
   sec="potential";
////////////////////////////////////////////////////////////////////////
   cpmrh(polyFac);
   cpmrh(bernsteinFac);
   cpmrh(bernsteinX);
   cpmrh(bernsteinY);
   cpmrh(bezierFac);
   cpmrh(ipolFac);
   cpmrh(gaussFac);
   cpmrh(indFac);
   cpmrh(poly);
   cpmrh(bernstein);
   cpmrh(bezierX);
   cpmrh(bezierY);
   cpmrh(ipolX);
   cpmrh(ipolY);
```

```
   cpmrh(gaussX);
   cpmrh(gaussSigma);
   cpmrh(gaussY);
   cpmrh(indX);
   cpmrh(indWidth);
   cpmrh(indY);
   cpmrh(ipolOrd);
   cpmrh(ipolMet);

//////////////////////////////////////////////////////////////////////
   sec="state1";
//////////////////////////////////////////////////////////////////////
   cpmrh(ipolMet);
   cpmrh(ipolOrd);
   cpmrh(polyFac);
   cpmrh(bernsteinFac);
   cpmrh(bernsteinX);
   cpmrh(bernsteinY);
   cpmrh(bezierFac);
   cpmrh(ipolFac);
   cpmrh(gaussFac);
   cpmrh(indFac);
   cpmrh(poly);
   cpmrh(bernstein);
   cpmrh(bezierX);
   cpmrh(bezierY);
   cpmrh(ipolX);
   cpmrh(ipolY);
   cpmrh(gaussX);
   cpmrh(gaussSigma);
   cpmrh(gaussY);
   cpmrh(indX);
   cpmrh(indWidth);
   cpmrh(indY);

//////////////////////////////////////////////////////////////////////
   sec="state2";
//////////////////////////////////////////////////////////////////////
   cpmrh(ipolMet);
   cpmrh(ipolOrd);
   cpmrh(polyFac);
   cpmrh(bernsteinFac);
   cpmrh(bernsteinX);
   cpmrh(bernsteinY);
   cpmrh(bezierFac);
   cpmrh(ipolFac);
   cpmrh(gaussFac);
   cpmrh(indFac);
   cpmrh(poly);
   cpmrh(bernstein);
```

```
   cpmrh(bezierX);
   cpmrh(bezierY);
   cpmrh(ipolX);
   cpmrh(ipolY);
   cpmrh(gaussX);
   cpmrh(gaussSigma);
   cpmrh(gaussY);
   cpmrh(indX);
   cpmrh(indWidth);
   cpmrh(indY);

//////////////////////////////////////////////////////////////////////
   sec="state3";
//////////////////////////////////////////////////////////////////////
   cpmrh(ipolMet);
   cpmrh(ipolOrd);
   cpmrh(polyFac);
   cpmrh(bernsteinFac);
   cpmrh(bernsteinX);
   cpmrh(bernsteinY);
   cpmrh(bezierFac);
   cpmrh(ipolFac);
   cpmrh(gaussFac);
   cpmrh(indFac);
   cpmrh(poly);
   cpmrh(bernstein);
   cpmrh(bezierX);
   cpmrh(bezierY);
   cpmrh(ipolX);
   cpmrh(ipolY);
   cpmrh(gaussX);
   cpmrh(gaussSigma);
   cpmrh(gaussY);
   cpmrh(indX);
   cpmrh(indWidth);
   cpmrh(indY);

//////////////////////////////////////////////////////////////////////
   sec="psi";
//////////////////////////////////////////////////////////////////////
   cpmrh(ipolMet);
   cpmrh(ipolOrd);
   cpmrh(polyFac);
   cpmrh(bernsteinFac);
   cpmrh(bernsteinX);
   cpmrh(bernsteinY);
   cpmrh(bezierFac);
   cpmrh(ipolFac);
   cpmrh(gaussFac);
   cpmrh(indFac);
```

```
   cpmrh(poly);
   cpmrh(bernstein);
   cpmrh(bezierX);
   cpmrh(bezierY);
   cpmrh(ipolX);
   cpmrh(ipolY);
   cpmrh(gaussX);
   cpmrh(gaussSigma);
   cpmrh(gaussY);
   cpmrh(indX);
   cpmrh(indWidth);
   cpmrh(indY);

//////////////////////////////////////////////////////////////////////
   sec="linear combinations of eigen-vectors";
//////////////////////////////////////////////////////////////////////
   cpmrh(cN);
   cpmrh(cRe);
   cpmrh(cIm);

//////////////////////////////////////////////////////////////////////
   sec="controling spectral representation";
//////////////////////////////////////////////////////////////////////
   cpmrh(enfSym);
   cpmrh(test);
   cpmrh(iterMax);
   cpmrh(eps);
   cpmrh(eps1);
   cpmrh(eps2);

//////////////////////////////////////////////////////////////////////
// notice that there will be a further reading activity from a section
//
//   run control
//
// in the form:
//   Record rtc=(rch.getRecord())("run control","TrjCon");

// setting the stage

   Color::setGamma(gamma);
   Graph g;
   g.setWithOrigin(0);
   Z i;

   if (useSI) QMBase::setUnits("SI");
   else QMBase::setUnits("1");
   QMBase::setMass(mass);
   QMBase::setSize(s);
   QMBase::setPhiStep(phiStep);
```

```
Iv ivi(-0.5,0.5);
Iv ivf(-s*0.5,s*0.5);
R lambda0=useRelCplCon ? QMBase::lambdaOsc(d) : 1.;

R tw2=waitTimes[2];
Z sP=512;

// defining the Observable ob and (optionally) its spectral object spo
Observable ob;
SpectralObject spo;
Operator op;
FieldOperator fop;

bool spoValid=false;

cpmmessage("point 1 reached");

if (spcObjFromFile){
   cpmmessage("Trying to read spectral object from file");
   IFileStream ifs1(inFiles[1]);
   spo.scanFrom(ifs1());
   spoValid=true;
   if(showSpcObj>0){
      g.addText("Spectral object from file");
      spo.show(g);
      cpmwait(showSpcObj);
   }
}
else{
   Word sec2="R_FuncRec";
   Record rp,rs1,rs2,rs3;

   if (buildV>0){
      sec="potential";
      rp=(rch.getRecord())(sec,sec2);
   }

   if (buildS>=1){
      sec="state1";
      rs1=(rch.getRecord())(sec,sec2);
   }

   if (buildS>=2){
      sec="state2";
      rs2=(rch.getRecord())(sec,sec2);
   }

   if (buildS>=3){
      sec="state3";
      rs3=(rch.getRecord())(sec,sec2);
```

```
      }

// defining operators
      Observable p,T,V_,S_(d),R_(d);
      cpmassert(buildT>=0,loc);
      cpmassert(buildT<=4,loc);
      if (buildT<0||buildT>5) buildT=2;

      if (buildT>0){
         if (standardMomentum)
            p=Observable("momentum",d);
               // multiplication operator in discrete Fourier space
         else
            p=Observable("pF",d);
               // finite difference

         if (cutMomentum) p=p.cut();

         if (buildT==1){
            T=p;
         }
         else if (buildT==2){
            if (standardMomentum) T=p*p*(1./(2*mass));
            else T=Observable("eKinF",d);
         }
         else if (buildT==3){
            T=Observable("eKinRel",d);
         }
         else if (buildT==4){
            T=Observable("boost",d);
         }
         else ; // can't happen
      }
      if (buildV>0){
         R_FuncRec fr(rp);
         R_Func fx=fr();
            // notice that a mathematical function of 'the
            // position' does not make sense without normalizing
            // the position to a dimension-less quantity. So it
            // is OK to consider all functions that deal with
            // distributions in space as defined on the normalized
            // interval [-0.5,0.5]
         fx*=(lambda0*kp);
         if (tw2>0){
            g.setX(ivi); // not rescaled
            g.addText("Continuous version of potential");
            g.show(fx,sP);
            cpmwait(tw2);
         }
         V_=Observable::pot(fx,d);
```

```
      // this is defined in a way that despite of an actual
      // position range [-size/2, size/2] function fx will be
      // always evaluated in [-1/2, 1/2]

   R2_Func fxy(fx);
   op=Operator::sinParHam(fxy,d);
   R2_Func fpair;
   // trivial interaction between pairs to have a good test
   // by comparing the dynamics from op and fop
   fop=FieldOperator::manParHam(fxy,fpair,d);
}
if (buildS>=1){
   R_FuncRec fs1(rs1);
   R_Func gs1=fs1();
   if (tw2>0){
      g.setX(ivi);
      g.addText("State 1");
      g.show(gs1,sP);
      cpmwait(tw2);
   }
   State s1(gs1,d);
   S_+=(Observable("projector",s1)*ks1);

   if (buildS>=2){
      R_FuncRec fs2(rs2);
      R_Func gs2=fs2();
      if (tw2>0){
         g.setX(ivi);
         g.addText("State 2");
         g.show(gs2,sP);
         cpmwait(tw2);
      }
      State s2(gs2,d);
      S_+=(Observable("projector",s2)*ks2);
   }

   if (buildS>=3){
      R_FuncRec fs3(rs3);
      R_Func gs3=fs3();
      if (tw2>0){
         g.setX(ivi);
         g.addText("State 3");
         g.show(gs3,sP);
         cpmwait(tw2);
      }
      State s3(gs3,d);
      S_+=(Observable("projector",s3)*ks3);
   }
   S_*=lambda0;
}
```

```
if (buildR>0){
   R_=R_.ranVal()*(kr*lambda0);
}

Observable o1(d); // zero operator
if (buildT>0) o1+=T;

Observable o2(d); // zero operator
if (buildV>0) o2+=V_;
if (buildS>0) o2+=S_;
if (buildR>0) o2+=R_;

ob=commutator ? (o1*o2-o2*o1)*C::I : o1+o2 ;

if (doublicate) ob=ob.prdFock();

if (buildSpcObj){
   spo=ob.spectralRepresentation(enfSym,test,iterMax,eps,eps1,eps2\
   )\
   ;
   spoValid=true;
}

g.clr();

if (showObs>0&&showSpcObj>0&&spoValid){
   if (ob.dim()<128){
      Frames frs(g,Z(1),Z(2));
      Graph g1(frs[1][1]);
      Graph g2(frs[1][2]);
      g1.addText("The observable as a complex matrix");
      g2.addText("Spectrum and eigenvectors of the observable");
      ob.show(g1);
      spo.show(g2);
      cpmwait(showObs+showSpcObj);
   }
   else{
      g.addText("The observable as a complex matrix");
      ob.show(g);

      cpmwait(showObs);
      g.clr();
      g.addText("Spectrum and eigenvectors of the observable");
      spo.show(g);
      cpmwait(showSpcObj);
   }
}
else if (showObs>0){
   g.addText("The observable as a complex matrix");
```

```
         ob.show(g);
        // g.addText("The observable as a complex matrix");
         cpmwait(showObs);
      }
      else if (showSpcObj>0&&spoValid){
         g.addText("Spectrum and eigenvectors of the observable");
         spo.show(g);
         cpmwait(showSpcObj);
      }
      else ;

      if (fileObs){
         OFileStream out1(outFiles[1]);
         out1()<<ob;
      }
      if (fileSpeObj&&buildSpcObj){
         OFileStream out2(outFiles[2]);
         out2()<<spo;
      }
      if (showSpc>0&&spoValid){
         R_Vector sp=spo.getSpectrum();
         cpmdata<<"Spectrum of the Observable="<<sp;
         Z bars=showSpcAsBars ? 1 : 0;
         R xPosTitle=0.05;
         g.show(sp,Color(RED),bars,
            "Spectrum of the Observable",xPosTitle);
         cpmwait(showSpc);
      }
   }

   cpmmessage("point 2 reached");

// defining the initial state psi
   State psi;
   bool psiValid=false;

   if (psiFromFile){
      IFileStream ifs2(inFiles[2]);
      psiValid=psi.scanFrom(ifs2());
      if (showPsi>0){
         g.addText("State psi from file");
         psi.showAsCurves(g);
         cpmwait(showPsi);
      }
   }
   else{
      if (spcObjFromFile) d=spo.dim();
      if (buildPsi==1){
         Record rpsi=(rch.getRecord())("psi","R_FuncRec");
         R_FuncRec fpsi(rpsi);
```

```
      R_Func gpsi=fpsi();
      if (showPsi>0){
         g.setX(ivi);
         g.addText("Continuous version of state psi");
         g.show(gpsi,sP);
         cpmwait(showPsi);
      }
      psi=State(gpsi,d);
      psiValid=true;
   }
   else if (buildPsi==2){
      cpmassert(spoValid,loc);
      Z cd0=cN.dim();
      Z cd1=cRe.dim();
      Z cd2=cIm.dim();
      cpmassert(cd0==cd1,loc);
      cpmassert(cd1==cd2,loc);
      Z cdm=CpmRootX::inf<Z>(cd0,cd1,cd2);
      psi=State(d);
      Z dm=CpmRootX::inf<Z>(cdm,d);
      for (i=1;i<=dm;++i){
         Z ispo=d-cN[i];
         if (ispo>=0 && ispo<d){
            psi+=(spo[d-cN[i]]*C(cRe[i],cIm[i]));
            // only existing eigen-vectors will be taken
         }
         else{
            cpmwarning(loc&": eigenvector with index "&cpm(ispo)&
               " does not exist, d="&cpm(d));
         }
      }
      psi.nor_();
      psiValid=true;
      cpmdata<<endl<<"State psi="<<psi;
   }
   else{
      ;
   }

   if (psiValid){
      R vRef=QMBase::vMax(d);
      R vB=vRef*boostRel;
      if (vB!=0){
         if (useSplitBoost)
            psi=psi.splitBoost(vB);
         else
            psi=psi.boost(vB);
      }
      if (showPsi>0){
         g.addText("State psi");
```

```
            psi.showAsCurves(g);
            cpmwait(showPsi);
            Z1 d1(d);
            bool b2=d1.isPowOfTwo();
            if (b2){
               State psiMom=psi.toMomSpc();
               g.addText("State psi in discrete momentum space");
               psiMom.showAsCurves(g);
            }
            cpmwait(showPsi);
         }
         if (filePsi){
            OFileStream out3(outFiles[3]);
            out3()<<psi;
         }
      }
   }

   cpmmessage("point 3 reached");

// showing the dynamical state evolution if this is
// requested by the input data
   R tw3=waitTimes[3];
   Record rtc=(rch.getRecord())("run control","TrjCon");
   TrjCon tc(rtc);
   Z nSteps=tc.nSteps_;
   Z readTrjFromFile=0;
   Z levOfPed=0; // level of pedantry
   rch.read("run control","readTrjFromFile",readTrjFromFile, levOfPed);
      // if reading failed, readTrjFromFile still has its previous value
      // which is 0. So older ini-files which don't have this parameter
      // (introduced 2004-10-11) continue to be valid for their particlar
      // task
   if (nSteps<=0 &&readTrjFromFile<=0){
      // then we have neither to read or create a trajectory
      CPM_MZ
      return;
   }

   if (readTrjFromFile>0){
      IFileStream in4(inFiles[4]);
      if (readTrjFromFile==1){
         State psiIrr; // Irr for irrelevant, has no function
         SpectralObject spoIrr;
         TrjSR trj1(spoIrr,psiIrr,tc);
         trj1.play(g,in4());
      }
      else if (readTrjFromFile==2){
         State psiIrr;
         Observable obIrr;
```

```
      TrjDMI_Obs trj2(obIrr,psiIrr,tc);
      trj2.play(g,in4());
   }
   else if (readTrjFromFile==3){
      State psiIrr;
      Operator opIrr;
      TrjDMI_Opr trj3(opIrr,psiIrr,tc);
      trj3.play(g,in4());
   }
   else if (readTrjFromFile==4){
      FieldState fsIrr;
      FieldOperator foIrr;
      TrjDMI_FilOpr trj3(foIrr,fsIrr,tc);
      trj3.play(g,in4());
   }
   else{
      cpmerror(loc&": readTrjFromFile="
         &cpm(readTrjFromFile)&" not defined");
   }
}
else{

   OFileStream out4(outFiles[4]);
      // output file for a single trajectory or the first one
      // in a tandem trajectory
   OFileStream out5(outFiles[5]);
      // output file for the second trajectory of a tandem
      // trajectory
   cpmmessage("point 4 reached");
   Z lim=1;
   if (nSteps>0 && psiValid){
      cpmmessage("point 5 reached");
      if (modeOfDyn==1){
         cpmassert(spoValid,loc);
         TrjSR trj(spo,psi,tc);
         trj.run(g,out4());
         cpmwait(tw3);
      }
      else if (modeOfDyn==2){
         TrjDMI_Obs trj(ob,psi,tc);
         trj.run(g,out4());
         cpmwait(tw3);
      }
      else if (modeOfDyn==3){
         TrjSR trj1=spoValid ? TrjSR(spo,psi,tc): TrjSR(ob,psi,tc);
         TrjDMI_Obs trj2(ob,psi,tc);
         Vp<TrjBase> vtrj(2); // polymorphic array
         vtrj[1]=trj1;
         vtrj[2]=trj2;
      //  Trj2 trj(trj1,trj2); // would also be OK
```

```
      Trj2 trj(vtrj);
      trj.run(g,out4(),out5());
      tw3*=4;
      cpmwait(tw3);
   }
   else if (modeOfDyn==4){
      TrjDMI_Obs trj1(ob,psi,tc);
      TrjDMI_Opr trj2(op,psi,tc);
      Vp<TrjBase> vtrj(2);
      vtrj[1]=trj1;
      vtrj[2]=trj2;
      Trj2 trj(vtrj);
      trj.run(g,out4(),out5());
      tw3*=4;
      cpmwait(tw3);
   }
   else if (modeOfDyn==5){
      TrjDMI_Obs trj(ob,psi,tc);
      trj.run(g,out4());
      cpmwait(tw3);
   }
   else if (modeOfDyn==6){
      TrjDMI_Opr trj(op,psi,tc);
      trj.run(g,out4());
      cpmwait(tw3);
   }
   else if (modeOfDyn==7){
      FieldState fs(d,1,lim);
      fs=fs.crt(psi);
      TrjDMI_FilOpr trj(fop,fs,tc);
      trj.run(g,out4());
      cpmwait(tw3);
   }
   else if (modeOfDyn==8){

      TrjDMI_Opr trj1(op,psi,tc);
      FieldState fs(d,1,lim);
      fs=fs.crt(psi);
      TrjDMI_FilOpr trj2(fop,fs,tc);

      Vp<TrjBase> vtrj(2);
      vtrj[1]=trj1;
      vtrj[2]=trj2;
      Trj2 trj(vtrj);

      trj.run(g,out4(),out5());
      cpmwait(tw3);
   }

   else if (modeOfDyn==9){
```

```
      V<R> shiftRel("",-0.2,0,0.15,0.3 );
      Z nShift=shiftRel.dim();
      V<State> states(nShift);

      for (i=1;i<=nShift;++i){
         R si=QMBase::getSize()*shiftRel[i];
         states[i]=psi.shift(si);
      }

      cpmmessage("states made");
      cpmdebug(nShift);

      Z lim1=1;
      Z lim2=1024;

      FieldState fs1(d,1,lim1);
      for (i=1;i<=nShift;++i){
         fs1=fs1.crt(states[i]);
      }
      fs1.nor_();
      cpmmessage("fs1 done");

      TrjDMI_FilOpr trj1(fop,fs1,tc);

      FieldState fs2(d,1,lim2);
      for (i=1;i<=nShift;++i){
         fs2=fs2.crt(states[i]);
      }
      fs2.nor_();
      cpmmessage("fs2 done");
      TrjDMI_FilOpr trj2(fop,fs2,tc);
//  TrjDMI_Opr trj2(op,states[1],tc);

      Vp<TrjBase> vtrj(2);
      vtrj[1]=trj1;
      vtrj[2]=trj2;
      Trj2 trj(vtrj);

      trj.run(g,out4(),out5());
      cpmwait(tw3);
   }
   else if (modeOfDyn==10){
      V<R> shiftRel("",-0.05,0.05 );
      Z nShift=shiftRel.dim();
      V<State> states(nShift);

      for (i=1;i<=nShift;++i){
         R si=QMBase::getSize()*shiftRel[i];
         states[i]=psi.shift(si);
      }
```

```
            cpmmessage("states made");
            Z lim1=10;
            FieldState fs1(d,1,lim1);
            for (i=1;i<=nShift;++i){
               fs1=fs1.crt(states[i]);
            }
            fs1.nor_();
            TrjDMI_FilOpr trj(fop,fs1,tc);
            trj.run(g,out4());
            cpmwait(tw3);
         }
         else if (modeOfDyn==11){
            R jitter=0.1;
            R omega=1e9;
            R_Func f1=R_Func::x4*16;
            R2_Func f2=F3<C,R_Func,R,R,R>(f1,jitter,omega)(jitterFunc);
            f2*=lambda0*kp;
            op=Operator::sinParHam(f2,d);
            TrjDMI_Opr trj(op,psi,tc);
            trj.run(g,out4());
            cpmwait(tw3);
         }
         else if (modeOfDyn==12){
          //  g.clr(BLACK);
            cpmmessage("modeOfDyn==12 started");
            Operator hf=Operator::freePar(d);
            SpectralObject hole("x",d);
            Iv ivHole(-0.45,-0.4);
            R_Func fHole=R_Func::charFunc(ivHole);
            hole.map_(fHole);
            Z n1,n2,n3;
            sec="system data";
            cpmrhf(n1);
            cpmrhf(n2);
            cpmrhf(n3);
            psi.setBra_(n1,n2,n3);
            Trj2DMI<State,Operator,SpectralObject> trj(psi,hf,hole,tc);
            trj.run(g,out4());
            cpmwait(tw3);
         }
         else{
            cpmmessage(loc&": nothing more to do");
         }
      }
   }
   CPM_MZ
}


/****************************** old and good *************************
```

```
////////////////////////////// App2 ////////////////////////////


namespace{
//R_Func f1=F2<R,R_Func,R,R>(f0,jitter)(jitterFunc);

R jitterFunc(C const& c, R_Func const& f, R const& jt, R const& omega )
{
   R x=c[1];
   R t=c[2];
   R res=f(x);
   if (x<=0) return res;
   else{
      R shift=::cos(t*omega)*jt;
      return res+shift;
   }
}

} // namespace

void CpmVQM2::App2::doTheWork()
{
   Z mL=1;
   Word loc("CpmVQM2::App2::doTheWork()");
   CPM_MA

   bool useSI,standardMomentum,cutMomentum,commutator,doublicate,
      buildSpcObj, enfSym,test,showSpcAsBars,
      fileObs,fileSpeObj,filePsi,
      spcObjFromFile,psiFromFile,
      useSplitBoost,useRelCplCon;

   Z d,iterMax,buildT,buildV,buildS,buildR, buildPsi,
      ipolOrd,ipolMet,modeOfDyn;

   R s,mass,kp,ks1,ks2,ks3,kr,boostRel,gamma,eps,eps1,eps2,
      polyFac,bernsteinFac,bezierFac,ipolFac,gaussFac,indFac,
      bernsteinX,bernsteinY,showObs,showSpcObj,showSpc,showPsi,
      phiStep;

   Word file;
   V<Word> line1;
   V<Word> line2;
   V<Word> line3;
   V<Word> line4;

   V<Z> cN;

   V<R> waitTimes,poly,bernstein,bezierX,bezierY,ipolX,ipolY,
      gaussX,gaussSigma,gaussY,indX,indWidth,indY,cRe,cIm;
```

```
      // notice that by far the most data come from sections
      // potential, state1, ..., state3 which do not shine up
      // here explicitely

// reading data
// I choose to read all data in succession although for most settings
// in run_contol only a part of these data is needed in the code
// to be executed furtheron. Letting each control path read only those
// data that it really needs would create complicated interdependencies
// of the various control paths, resulting in the need for doublicating
// code. This also would complicate further extensions.
//////////////////////////////////////////////////////////////////////
   Word sec="about";
//////////////////////////////////////////////////////////////////////
   cpmrh(file);
   cpmrh(line1);
   cpmrh(line2);
   cpmrh(line3);
   cpmrh(line4);

//////////////////////////////////////////////////////////////////////
   sec="run preparation";
//////////////////////////////////////////////////////////////////////
   cpmrh(waitTimes);
   cpmrh(gamma);
   cpmrh(fileObs);
   cpmrh(fileSpeObj);
   cpmrh(filePsi);
   cpmrh(spcObjFromFile);
   cpmrh(psiFromFile);
   cpmrh(showObs);
   cpmrh(showSpcObj);
   cpmrh(showSpc);
   cpmrh(showPsi);
   cpmrh(showSpcAsBars);
   cpmrh(phiStep);
   cpmrh(modeOfDyn);

//////////////////////////////////////////////////////////////////////
   sec="system data";
//////////////////////////////////////////////////////////////////////
   cpmrh(d);
   cpmrh(s);
   cpmrh(mass);
   cpmrh(useSI);
   cpmrh(doublicate);
   cpmrh(buildSpcObj);
   cpmrh(commutator);
   cpmrh(buildT);
   cpmrh(buildV);
```

```
   cpmrh(buildS);
   cpmrh(buildR);
   cpmrh(buildPsi);
   cpmrh(useRelCplCon);
   cpmrh(kp);
   cpmrh(ks1);
   cpmrh(ks2);
   cpmrh(ks3);
   cpmrh(kr);
   cpmrh(boostRel);
   cpmrh(useSplitBoost);

//////////////////////////////////////////////////////////////////////
   sec="kinetic energy";
//////////////////////////////////////////////////////////////////////
   cpmrh(standardMomentum);
   cpmrh(cutMomentum);

//////////////////////////////////////////////////////////////////////
   sec="potential";
//////////////////////////////////////////////////////////////////////
   cpmrh(polyFac);
   cpmrh(bernsteinFac);
   cpmrh(bernsteinX);
   cpmrh(bernsteinY);
   cpmrh(bezierFac);
   cpmrh(ipolFac);
   cpmrh(gaussFac);
   cpmrh(indFac);
   cpmrh(poly);
   cpmrh(bernstein);
   cpmrh(bezierX);
   cpmrh(bezierY);
   cpmrh(ipolX);
   cpmrh(ipolY);
   cpmrh(gaussX);
   cpmrh(gaussSigma);
   cpmrh(gaussY);
   cpmrh(indX);
   cpmrh(indWidth);
   cpmrh(indY);
   cpmrh(ipolOrd);
   cpmrh(ipolMet);

//////////////////////////////////////////////////////////////////////
   sec="state1";
//////////////////////////////////////////////////////////////////////
   cpmrh(ipolMet);
   cpmrh(ipolOrd);
   cpmrh(polyFac);
```

```
   cpmrh(bernsteinFac);
   cpmrh(bernsteinX);
   cpmrh(bernsteinY);
   cpmrh(bezierFac);
   cpmrh(ipolFac);
   cpmrh(gaussFac);
   cpmrh(indFac);
   cpmrh(poly);
   cpmrh(bernstein);
   cpmrh(bezierX);
   cpmrh(bezierY);
   cpmrh(ipolX);
   cpmrh(ipolY);
   cpmrh(gaussX);
   cpmrh(gaussSigma);
   cpmrh(gaussY);
   cpmrh(indX);
   cpmrh(indWidth);
   cpmrh(indY);

//////////////////////////////////////////////////////////////////////
   sec="state2";
//////////////////////////////////////////////////////////////////////
   cpmrh(ipolMet);
   cpmrh(ipolOrd);
   cpmrh(polyFac);
   cpmrh(bernsteinFac);
   cpmrh(bernsteinX);
   cpmrh(bernsteinY);
   cpmrh(bezierFac);
   cpmrh(ipolFac);
   cpmrh(gaussFac);
   cpmrh(indFac);
   cpmrh(poly);
   cpmrh(bernstein);
   cpmrh(bezierX);
   cpmrh(bezierY);
   cpmrh(ipolX);
   cpmrh(ipolY);
   cpmrh(gaussX);
   cpmrh(gaussSigma);
   cpmrh(gaussY);
   cpmrh(indX);
   cpmrh(indWidth);
   cpmrh(indY);

//////////////////////////////////////////////////////////////////////
   sec="state3";
//////////////////////////////////////////////////////////////////////
   cpmrh(ipolMet);
```

```
   cpmrh(ipolOrd);
   cpmrh(polyFac);
   cpmrh(bernsteinFac);
   cpmrh(bernsteinX);
   cpmrh(bernsteinY);
   cpmrh(bezierFac);
   cpmrh(ipolFac);
   cpmrh(gaussFac);
   cpmrh(indFac);
   cpmrh(poly);
   cpmrh(bernstein);
   cpmrh(bezierX);
   cpmrh(bezierY);
   cpmrh(ipolX);
   cpmrh(ipolY);
   cpmrh(gaussX);
   cpmrh(gaussSigma);
   cpmrh(gaussY);
   cpmrh(indX);
   cpmrh(indWidth);
   cpmrh(indY);

//////////////////////////////////////////////////////////////////////////
   sec="psi";
//////////////////////////////////////////////////////////////////////////
   cpmrh(ipolMet);
   cpmrh(ipolOrd);
   cpmrh(polyFac);
   cpmrh(bernsteinFac);
   cpmrh(bernsteinX);
   cpmrh(bernsteinY);
   cpmrh(bezierFac);
   cpmrh(ipolFac);
   cpmrh(gaussFac);
   cpmrh(indFac);
   cpmrh(poly);
   cpmrh(bernstein);
   cpmrh(bezierX);
   cpmrh(bezierY);
   cpmrh(ipolX);
   cpmrh(ipolY);
   cpmrh(gaussX);
   cpmrh(gaussSigma);
   cpmrh(gaussY);
   cpmrh(indX);
   cpmrh(indWidth);
   cpmrh(indY);

//////////////////////////////////////////////////////////////////////////
   sec="linear combinations of eigen-vectors";
```

```
//////////////////////////////////////////////////////////////////////
   cpmrh(cN);
   cpmrh(cRe);
   cpmrh(cIm);

//////////////////////////////////////////////////////////////////////
   sec="controling spectral representation";
//////////////////////////////////////////////////////////////////////
   cpmrh(enfSym);
   cpmrh(test);
   cpmrh(iterMax);
   cpmrh(eps);
   cpmrh(eps1);
   cpmrh(eps2);

//////////////////////////////////////////////////////////////////////
// notice that there will be a further reading activity from a section
//
//   run control
//
// in the form:
//   Record rtc=(rch.getRecord())("run control","TrjCon");

// setting the stage

   Color::setGamma(gamma);
   Graph g;
   g.setWithOrigin(0);
   Z i;

   if (useSI) QMBase::setUnits("SI");
   else QMBase::setUnits("1");
   QMBase::setMass(mass);
   QMBase::setSize(s);
   QMBase::setPhiStep(phiStep);
   Iv ivi(-0.5,0.5);
   Iv ivf(-s*0.5,s*0.5);
   R lambda0=useRelCplCon ? QMBase::lambdaOsc(d) : 1.;

   R tw2=waitTimes[2];
   Z sP=512;

   // defining the Observable ob and (optionally) its spectral object spo
   Observable ob;
   SpectralObject spo;
   Operator op;
   FieldOperator fop;

   bool spoValid=false;
```

```
   if (spcObjFromFile){
      cpmmessage("Trying to read spectral object from file");
      IFileStream ifs1(inFiles[1]);
      spo.scanFrom(ifs1());
      spoValid=true;
      if(showSpcObj>0){
         g.addText("Spectral object from file");
         spo.show(g);
         cpmwait(showSpcObj);
      }
   }
   else{
      Word sec2="R_FuncRec";
      Record rp,rs1,rs2,rs3;

      if (buildV>0){
         sec="potential";
         rp=(rch.getRecord())(sec,sec2);
      }

      if (buildS>=1){
         sec="state1";
         rs1=(rch.getRecord())(sec,sec2);
      }

      if (buildS>=2){
         sec="state2";
         rs2=(rch.getRecord())(sec,sec2);
      }

      if (buildS>=3){
         sec="state3";
         rs3=(rch.getRecord())(sec,sec2);
      }

// defining operators
      Observable p,T,V_,S_(d),R_(d);
      cpmassert(buildT>=0,loc);
      cpmassert(buildT<=4,loc);
      if (buildT<0||buildT>5) buildT=2;

      if (buildT>0){
         if (standardMomentum)
            p=Observable("momentum",d);
               // multiplication operator in discrete Fourier space
         else
            p=Observable("pF",d);
               // finite difference

         if (cutMomentum) p=p.cut();
```

```
      if (buildT==1){
         T=p;
      }
      else if (buildT==2){
         if (standardMomentum) T=p*p*(1./(2*mass));
         else T=Observable("eKinF",d);
      }
      else if (buildT==3){
         T=Observable("eKinRel",d);
      }
      else if (buildT==4){
         T=Observable("boost",d);
      }
      else ; // can't happen
   }
   if (buildV>0){
      R_FuncRec fr(rp);
      R_Func fx=fr();
         // notice that a mathematical function of 'the
         // position' does not make sense without normalizing
         // the position to a dimension-less quantity. So it
         // is OK to consider all functions that deal with
         // distributions in space as defined on the normalized
         // interval [-0.5,0.5]
      fx*=(lambda0*kp);
      if (tw2>0){
         g.setX(ivi); // not rescaled
         g.addText("Continuous version of potential");
         g.show(fx,sP);
         cpmwait(tw2);
      }
      V_=Observable::pot(fx,d);
         // this is defined in a way that despite of an actual
         // position range [-size/2, size/2] function fx will be
         // always evaluated in [-1/2, 1/2]

      R2_Func fxy(fx);
      op=Operator::sinParHam(fxy,d);
      R2_Func fpair;
      // trivial interaction between pairs to have a good test
      // by comparing the dynamics from op and fop
      fop=FieldOperator::manParHam(fxy,fpair,d);
   }
   if (buildS>=1){
      R_FuncRec fs1(rs1);
      R_Func gs1=fs1();
      if (tw2>0){
         g.setX(ivi);
         g.addText("State 1");
```

```
      g.show(gs1,sP);
      cpmwait(tw2);
   }
   State s1(gs1,d);
   S_+=(Observable("projector",s1)*ks1);

   if (buildS>=2){
      R_FuncRec fs2(rs2);
      R_Func gs2=fs2();
      if (tw2>0){
         g.setX(ivi);
         g.addText("State 2");
         g.show(gs2,sP);
         cpmwait(tw2);
      }
      State s2(gs2,d);
      S_+=(Observable("projector",s2)*ks2);
   }

   if (buildS>=3){
      R_FuncRec fs3(rs3);
      R_Func gs3=fs3();
      if (tw2>0){
         g.setX(ivi);
         g.addText("State 3");
         g.show(gs3,sP);
         cpmwait(tw2);
      }
      State s3(gs3,d);
      S_+=(Observable("projector",s3)*ks3);
   }
   S_*=lambda0;
}

if (buildR>0){
   R_=R_.ranVal()*(kr*lambda0);
}

Observable o1(d); // zero operator
if (buildT>0) o1+=T;

Observable o2(d); // zero operator
if (buildV>0) o2+=V_;
if (buildS>0) o2+=S_;
if (buildR>0) o2+=R_;

ob=commutator ? (o1*o2-o2*o1)*C::I : o1+o2 ;

if (doublicate) ob=ob.prdFock();
```

```
if (buildSpcObj){
   spo=ob.spectralRepresentation(enfSym,test,iterMax,eps,eps1,eps2\
   )\
   ;
   spoValid=true;
}

g.clr();

if (showObs>0&&showSpcObj>0&&spoValid){
   if (ob.dim()<128){
      Frames frs(g,Z(1),Z(2));
      Graph g1(frs[1][1]);
      Graph g2(frs[1][2]);
      g1.addText("The observable as a complex matrix");
      g2.addText("Spectrum and eigenvectors of the observable");
      ob.show(g1);
      spo.show(g2);
      cpmwait(showObs+showSpcObj);
   }
   else{
      g.addText("The observable as a complex matrix");
      ob.show(g);

      cpmwait(showObs);
      g.clr();
      g.addText("Spectrum and eigenvectors of the observable");
      spo.show(g);
      cpmwait(showSpcObj);
   }
}
else if (showObs>0){
   g.addText("The observable as a complex matrix");
   ob.show(g);
  // g.addText("The observable as a complex matrix");
   cpmwait(showObs);
}
else if (showSpcObj>0&&spoValid){
   g.addText("Spectrum and eigenvectors of the observable");
   spo.show(g);
   cpmwait(showSpcObj);
}
else ;

if (fileObs){
   OFileStream out1(outFiles[1]);
   out1()<<ob;
}
if (fileSpeObj&&buildSpcObj){
   OFileStream out2(outFiles[2]);
```

```
         out2()<<spo;
      }
      if (showSpc>0&&spoValid){
         R_Vector sp=spo.getSpectrum();
         cpmdata<<"Spectrum of the Observable="<<sp;
         Z bars=showSpcAsBars ? 1 : 0;
         R xPosTitle=0.05;
         g.show(sp,Color(RED),bars,
            "Spectrum of the Observable",xPosTitle);
         cpmwait(showSpc);
      }
   }

// defining the initial state psi
   State psi;
   bool psiValid=false;

   if (psiFromFile){
      IFileStream ifs2(inFiles[2]);
      psiValid=psi.scanFrom(ifs2());
      if (showPsi>0){
         g.addText("State psi from file");
         psi.showAsCurves(g);
         cpmwait(showPsi);
      }
   }
   else{
      if (spcObjFromFile) d=spo.dim();
      if (buildPsi==1){
         Record rpsi=(rch.getRecord())("psi","R_FuncRec");
         R_FuncRec fpsi(rpsi);
         R_Func gpsi=fpsi();
         if (showPsi>0){
            g.setX(ivi);
            g.addText("Continuous version of state psi");
            g.show(gpsi,sP);
            cpmwait(showPsi);
         }
         psi=State(gpsi,d);
         psiValid=true;
      }
      else if (buildPsi==2){
         cpmassert(spoValid,loc);
         Z cd0=cN.dim();
         Z cd1=cRe.dim();
         Z cd2=cIm.dim();
         cpmassert(cd0==cd1,loc);
         cpmassert(cd1==cd2,loc);
         Z cdm=CpmRootX::inf<Z>(cd0,cd1,cd2);
         psi=State(d);
```

```
         Z dm=CpmRootX::inf<Z>(cdm,d);
         for (i=1;i<=dm;++i){
            Z ispo=d-cN[i];
            if (ispo>=0 && ispo<d){
               psi+=(spo[d-cN[i]]*C(cRe[i],cIm[i]));
               // only existing eigen-vectors will be taken
            }
            else{
               cpmwarning(loc&": eigenvector with index "&cpm(ispo)&
                  " does not exist, d="&cpm(d));
            }
         }
         psi.nor_();
         psiValid=true;
         cpmdata<<endl<<"State psi="<<psi;
      }
      else{
         ;
      }

      if (psiValid){
         R vRef=QMBase::vMax(d);
         R vB=vRef*boostRel;
         if (vB!=0){
            if (useSplitBoost)
               psi=psi.splitBoost(vB);
            else
               psi=psi.boost(vB);
         }
         if (showPsi>0){
            g.addText("State psi");
            psi.showAsCurves(g);
            cpmwait(showPsi);
            State psiMom=psi.toMomSpc();
            g.addText("State psi in discrete momentum space");
            psiMom.showAsCurves(g);
            cpmwait(showPsi);
         }

         if (filePsi){
            OFileStream out3(outFiles[3]);
            out3()<<psi;
         }
      }
   }

// showing the dynamical state evolution if this is
// requested by the input data
   R tw3=waitTimes[3];
   Record rtc=(rch.getRecord())("run control","TrjCon");
```

```
TrjCon tc(rtc);
Z nSteps=tc.nSteps_;
Z readTrjFromFile=0;
Z levOfPed=0; // level of pedantry
rch.read("run control","readTrjFromFile",readTrjFromFile, levOfPed);
   // if reading failed, readTrjFromFile still has its previous value
   // which is 0. So older ini-files which don't have this parameter
   // (introduced 2004-10-11) continue to be valid for their particlar
   // task
if (nSteps<=0 &&readTrjFromFile<=0){
   // then we have neither to read or create a trajectory
   CPM_MZ
   return;
}

if (readTrjFromFile>0){
   IFileStream in4(inFiles[4]);
   if (readTrjFromFile==1){
      State psiIrr; // Irr for irrelevant, has no function
      SpectralObject spoIrr;
      TrjSR trj1(spoIrr,psiIrr,tc);
      trj1.play(g,in4());
   }
   else if (readTrjFromFile==2){
      State psiIrr;
      Observable obIrr;
      TrjDMI_Obs trj2(obIrr,psiIrr,tc);
      trj2.play(g,in4());
   }
   else if (readTrjFromFile==3){
      State psiIrr;
      Operator opIrr;
      TrjDMI_Opr trj3(opIrr,psiIrr,tc);
      trj3.play(g,in4());
   }
   else if (readTrjFromFile==4){
      FieldState fsIrr;
      FieldOperator foIrr;
      TrjDMI_FilOpr trj3(foIrr,fsIrr,tc);
      trj3.play(g,in4());
   }
   else{
      cpmerror(loc&": readTrjFromFile="
         &cpm(readTrjFromFile)&" not defined");
   }
}
else{

   OFileStream out4(outFiles[4]);
      // output file for a single trajectory or the first one
```

```
   // in a tandem trajectory
OFileStream out5(outFiles[5]);
   // output file for the second trajectory of a tandem
   // trajectory

Z lim=1;
if (nSteps>0 && psiValid){
   if (modeOfDyn==1){
      cpmassert(spoValid,loc);
      TrjSR trj(spo,psi,tc);
      trj.run(g,out4());
      cpmwait(tw3);
   }
   else if (modeOfDyn==2){
      TrjDMI_Obs trj(ob,psi,tc);
      trj.run(g,out4());
      cpmwait(tw3);
   }
   else if (modeOfDyn==3){
      TrjSR trj1=spoValid ? TrjSR(spo,psi,tc): TrjSR(ob,psi,tc);
      TrjDMI_Obs trj2(ob,psi,tc);
      Vp<TrjBase> vtrj(2); // polymorphic array
      vtrj[1]=trj1;
      vtrj[2]=trj2;
   //  Trj2 trj(trj1,trj2); // would also be OK
      Trj2 trj(vtrj);
      trj.run(g,out4(),out5());
      tw3*=4;
      cpmwait(tw3);
   }
   else if (modeOfDyn==4){
      TrjDMI_Obs trj1(ob,psi,tc);
      TrjDMI_Opr trj2(op,psi,tc);
      Vp<TrjBase> vtrj(2);
      vtrj[1]=trj1;
      vtrj[2]=trj2;
      Trj2 trj(vtrj);
      trj.run(g,out4(),out5());
      tw3*=4;
      cpmwait(tw3);
   }
   else if (modeOfDyn==5){
      TrjDMI_Obs trj(ob,psi,tc);
      trj.run(g,out4());
      cpmwait(tw3);
   }
   else if (modeOfDyn==6){
      TrjDMI_Opr trj(op,psi,tc);
      trj.run(g,out4());
      cpmwait(tw3);
```

```
}
else if (modeOfDyn==7){
   FieldState fs(d,1,lim);
   fs=fs.crt(psi);
   TrjDMI_FilOpr trj(fop,fs,tc);
   trj.run(g,out4());
   cpmwait(tw3);
}
else if (modeOfDyn==8){

   TrjDMI_Opr trj1(op,psi,tc);
   FieldState fs(d,1,lim);
   fs=fs.crt(psi);
   TrjDMI_FilOpr trj2(fop,fs,tc);

   Vp<TrjBase> vtrj(2);
   vtrj[1]=trj1;
   vtrj[2]=trj2;
   Trj2 trj(vtrj);

   trj.run(g,out4(),out5());
   cpmwait(tw3);
}

else if (modeOfDyn==9){
   V<R> shiftRel("",-0.2,0,0.15,0.3 );
   Z nShift=shiftRel.dim();
   V<State> states(nShift);

   for (i=1;i<=nShift;++i){
      R si=QMBase::getSize()*shiftRel[i];
      states[i]=psi.shift(si);
   }

   cpmmessage("states made");
   cpmdebug(nShift);

   Z lim1=1;
   Z lim2=1024;

   FieldState fs1(d,1,lim1);
   for (i=1;i<=nShift;++i){
      fs1=fs1.crt(states[i]);
   }
   fs1.nor_();
   cpmmessage("fs1 done");

   TrjDMI_FilOpr trj1(fop,fs1,tc);

   FieldState fs2(d,1,lim2);
```

```
      for (i=1;i<=nShift;++i){
         fs2=fs2.crt(states[i]);
      }
      fs2.nor_();
      cpmmessage("fs2 done");
      TrjDMI_FilOpr trj2(fop,fs2,tc);
   //  TrjDMI_Opr trj2(op,states[1],tc);

      Vp<TrjBase> vtrj(2);
      vtrj[1]=trj1;
      vtrj[2]=trj2;
      Trj2 trj(vtrj);

      trj.run(g,out4(),out5());
      cpmwait(tw3);
   }
   else if (modeOfDyn==10){
      V<R> shiftRel("",-0.05,0.05 );
      Z nShift=shiftRel.dim();
      V<State> states(nShift);

      for (i=1;i<=nShift;++i){
         R si=QMBase::getSize()*shiftRel[i];
         states[i]=psi.shift(si);
      }

      cpmmessage("states made");
      Z lim1=10;
      FieldState fs1(d,1,lim1);
      for (i=1;i<=nShift;++i){
         fs1=fs1.crt(states[i]);
      }
      fs1.nor_();
      TrjDMI_FilOpr trj(fop,fs1,tc);
      trj.run(g,out4());
      cpmwait(tw3);
   }
}
else if (modeOfDyn==11){
      R jitter=0.1;
      R omega=1e9;
      R_Func f1=R_Func::x4*16;
      R2_Func f2=F3<C,R_Func,R,R,R>(f1,jitter,omega)(jitterFunc);
      f2*=lambda0*kp;
      op=Operator::sinParHam(f2,d);
      TrjDMI_Opr trj(op,psi,tc);
      trj.run(g,out4());
      cpmwait(tw3);
}
else{
```

```
         cpmmessage(loc&": nothing more to do");
      }
   }
   CPM_MZ
}
*****************************************************************/

//////////////////////////////// App3 ////////////////////////////////

namespace{
   R gauss(const R& x, const R& x0, const R& sigma)
   {
      R y=(x-x0)/sigma;
      return exp(-y*y);
   }
}

void CpmVQM2::App3::doTheWork()
{
   Z mL=2;
   Word loc("CpmVQM2::App3::doTheWork()");
   CPM_MA

   Z n,lim;
   bool twoParticles,useSI,useFockSpace;
   R mass,phiStep,x1,x2,sigma1,sigma2,v1Rel,v2Rel;
   R lambda1,lambda2,size;
   Word sec("system data");

   rch.read(sec,"dimension",n);
   rch.read(sec,"phiStep",phiStep);
   rch.read(sec,"useSI",useSI);
   rch.read(sec,"size",size);
   rch.read(sec,"mass",mass);
   rch.read(sec,"lim",lim);
   rch.read(sec,"x1",x1);
   rch.read(sec,"x2",x2);
   rch.read(sec,"sigma1",sigma1);
   rch.read(sec,"sigma2",sigma2);
   rch.read(sec,"v1Rel",v1Rel);
   rch.read(sec,"v2Rel",v2Rel);
   rch.read(sec,"lambda1",lambda1);
   rch.read(sec,"lambda2",lambda2);
   rch.read(sec,"useFockSpace",useFockSpace);
   rch.read(sec,"twoParticles",twoParticles);

   if (useSI) QMBase::setUnits("SI");
   else QMBase::setUnits("1");
   QMBase::setSize(size);
   QMBase::setMass(mass);
```

```
QMBase::setPhiStep(phiStep);
Frame fr;

F<R,R> shape1=F2<R,R,R,R>(x1,sigma1)(gauss);
CpmFunctions::R_Func f1(shape1);
State psi1(f1,n);
psi1.nor_();

F<R,R> shape2=F2<R,R,R,R>(x2,sigma2)(gauss);
CpmFunctions::R_Func f2(shape2);
State psi2(f2,n);
psi2.nor_();

Observable boost("n",n);
R vRef=QMBase::vMax(n);

if (v1Rel!=0){
   R v1=v1Rel*vRef;
   psi1=psi1.boost(v1);
}

if (twoParticles && v2Rel!=0){
   R v2=v2Rel*vRef;
   psi2=psi2.boost(v2);
}

Observable E("eKinF",n);
R lambdaRef=QMBase::lambdaOsc(n);
if (lambda1!=0){
   Observable x("x",n);
   R lambdaE=lambda1*lambdaRef;
   Observable V=x*x*lambdaE;
   E+=V;
}

Record rtc=(rch.getRecord())("run control","TrjCon");
TrjCon tc(rtc);

OFileStream out4(outFiles[4]);

Graph g;

TrjDMI_Obs trj1(E,psi1,tc);

if (!useFockSpace){
   trj1.run(g,out4());
}
else{
   OFileStream out5(outFiles[5]);
```

```
      FieldOperator fop=FieldOperator::cook1(E);
      FieldState fs(n,1,lim);
      fs=fs.crt(psi1);
      if (!twoParticles){
         TrjDMI_FilOpr trj2(fop,fs,tc);
         Vp<TrjBase> vtrj(2);
         vtrj[1]=trj1;
         vtrj[2]=trj2;

         Trj2 trj(vtrj);
         trj.run(g,out4(),out5());
      }
      else{
         fs=fs.crt(psi2);
         if (lambda2!=0.){
            R_Func x2=R_Func::x2;
            R lambdaU=lambda2*lambdaRef;
            Observable U=Observable::pot2(x2,n,lambdaU);
            fop+=FieldOperator::cook2(U);
         }
         TrjDMI_FilOpr trj2(fop,fs,tc);
         trj2.run(g,out5());
      }
   }
   CPM_MZ
}

//////////////////////////////// App4 ////////////////////////////////
// scattering

namespace{
   R clbFnc(R const& x, R const& r, R const& fac)
   {
      return fac/::sqrt(x*x+r*r);
   }
} // namespace

void CpmVQM2::App4::doTheWork()
{
   Z mL=1;
   Word loc("CpmVQM2::App4::doTheWork()");
   CPM_MA

   bool useSI, runFromFile=false;
   Z d,nPar=0;
   R s,mass,lam1Rel,lam2Rel,xRel,wRel,bstRel,
      gamma,showPsi,phiStep;
   V<R> waitTimes;
   V<Z> statistics;
```

```
   Word file;
   V<Word> line1;
   V<Word> line2;
   V<Word> line3;
   V<Word> line4;

// reading data
// I choose to read all data in succession although for most settings
// in run_contol only a part of these data is needed in the code
// to be executed furtheron. Letting each control path read only those
// data that it really needs would create complicated interdependencies
// of the various control paths, resulting in the need for doublicating
// code. This also would complicate further extensions.

//////////////////////////////////////////////////////////////////////
   Word sec="about";
//////////////////////////////////////////////////////////////////////
   cpmrh(file);
   cpmrh(line1);
   cpmrh(line2);
   cpmrh(line3);
   cpmrh(line4);

//////////////////////////////////////////////////////////////////////
   sec="run preparation";
//////////////////////////////////////////////////////////////////////
   cpmrh(waitTimes);
   cpmrh(gamma);
   cpmrh(showPsi);
   cpmrh(phiStep);
   cpmrh(statistics);
   cpmrh(runFromFile);

//////////////////////////////////////////////////////////////////////
   sec="system data";
//////////////////////////////////////////////////////////////////////
   V<R> shifts1;
   V<R> shifts2;

   cpmrh(d);
   cpmrh(s);
   cpmrh(mass);
   cpmrhf(nPar);
   cpmrhf(shifts1);
   cpmrhf(shifts2);
   cpmrh(useSI);
   cpmrh(lam1Rel);
   cpmrh(lam2Rel);

   cpmrh(xRel);
```

```
   cpmrh(wRel);
   cpmrh(bstRel);

//////////////////////////////////////////////////////////

// setting the stage

   Color::setGamma(gamma);
   Graph g;
   g.setWithOrigin(0);
   Z i;
   if (useSI) QMBase::setUnits("SI");
   else QMBase::setUnits("1");
   QMBase::setMass(mass);
   QMBase::setSize(s);
   QMBase::setPhiStep(phiStep);
   Iv ivi(-0.5,0.5);
   Iv ivf(-s*0.5,s*0.5);
   R pMax=QMBase::pMax(d);
   R vMax=QMBase::vMax(d);
   R eMax=pMax*pMax/(2*mass);

   R tw1=waitTimes[1];

   Z sP=512;

// defining the FieldOperator fop
   // first the confining potential ~ x^8

   R fac1=eMax*256*lam1Rel;
      // 256 is the reciprocal value of the maximum value
      // of the un-scaled confining potential
   R_Func pot1=R_Func::x4*R_Func::x4*fac1; // ~x^8 rather flat
      // till close to the rims of the domain
   // second the pair potential
   R r0=s/d;
   R fac2=eMax*r0*lam2Rel;
      // r0 is the reciprocal value of the maximum value of the un-scaled
      // Coulomb potential
   R_Func pot2=F2<R,R,R,R>(r0,fac2)(clbFnc);

   R2_Func p1(pot1);
   R2_Func p2(pot2);
   // and now the corresponding many-particle Hamiltonian:
   FieldOperator fop=FieldOperator::manParHam(p1,p2,d);
 // initial state:
   bool unitHeg=true; // aha heg = height
   R_Func gpsi1=R_Func::Gauss(xRel,wRel,unitHeg);
   R_Func gpsi2=R_Func::Gauss(-xRel,wRel,unitHeg);
   if (showPsi>0){
```

```
      g.setX(ivi);
      g.addText("Continuous version of state psi1+psi2");
      g.show(gpsi1+gpsi2,sP);
      cpmwait(showPsi);
   }
   State psi1(gpsi1,d);
   State psi2(gpsi2,d);
   R v1=bstRel*vMax;
   // so negative bstRel makes the particles move together
   R v2=-v1;
   psi1=psi1.boost(v1);
   psi2=psi2.boost(v2);

   Record rtc=(rch.getRecord())("run control","TrjCon");
          // Here the section 'run control' comes in
   rtc.prnOn(cpmdata);
   TrjCon tc(rtc);

   Z nSteps=tc.nSteps_;
   Z sd1=shifts1.dim(),sd2=shifts2.dim();
   Z sds=sd1+sd2;
   V<State> states;
   if (nPar==0 || sds==0 ){ // taking the two prepared states
      states=V<State>(2);
      states[1]=psi1;
      states[2]=psi2;
   }
   else{ // making more states by translation
      nPar=sd1+sd2;
      states=V<State>(nPar);
      for (i=states.b();i<=states.e();++i){
         if (i<=sd1) states[i]=psi1.shift(shifts1[i]);
         else states[i]=psi2.shift(shifts2[i-sd1]);
      }
   }
   Z ns=states.dim();
   Z sd=statistics.dim();
      // This is allowed to be larger than 1; the two runs are done in
      // parallel for method comparison

// creating dynamical state evolution, not reading it from file
   if (!runFromFile){
      Z lim1=statistics[1];
      FieldState fs1(d,1,lim1); // Fock vaccum
      for (i=states.b();i<=states.e();++i) fs1=fs1.crt(states[i]);
         // creating a multi-particle state by sequential application
         // of creation operators
      fs1.nor_();
      cpmmessage("fs1 done");
      OFileStream out1(outFiles[1]);
```

```
      TrjDMI_FilOpr trj1(fop,fs1,tc);
      if (sd==1){ // usual case if a single system (many particles,
         // though) is to be run
         trj1.run(g,out1());
         cpmwait(tw1);
      }
      else if (sd==2) {
         Z lim2=statistics[2];
         FieldState fs2(d,1,lim2);
         for (i=1;i<=ns;++i) fs2=fs2.crt(states[i]);
         fs2.nor_();
         cpmmessage("fs2 done");
         OFileStream out2(outFiles[2]);
         TrjDMI_FilOpr trj2(fop,fs2,tc);
         Vp<TrjBase> trj(2);
         trj[1]=trj1;
         trj[2]=trj2;
         Trj2 trj_2(trj);
         trj_2.run(g,out1(),out2());
         cpmwait(tw1);
      }
      else{
         cpmerror("statistics must have one or two entries");
      }
   }
// showing the dynamical state evolution from file
// This is the modality to be used when a series of images is needed
// in order to make a gif-file.
   else{
      Graph g2;
      if (tc.disMode_>=3 && tc.imgPer_>0){
         Frame fr1;
         R ari=fr1.getAspRatInv();
         R ySub=0.45; // 0<ySub<=1, allows to have the images not too
         // large
         Frame fr2(0.1,0.1,ySub*ari,ySub);
            // we make the image frame smaller in order not to overload
            // ImageMagick::convert in making a gif-file
         g2=Graph(fr2);
      }
      FieldState fsIrr;
      FieldOperator fopIrr;
      IFileStream in1(inFiles[1]);
      TrjDMI_FilOpr trj1(fopIrr,fsIrr,tc);
      if (sd==1){
         trj1.play(g2,in1());
      }
      else if (sd==2) {
         IFileStream in2(inFiles[2]);
         TrjDMI_FilOpr trj2(fopIrr,fsIrr,tc);
```

```
         Vp<TrjBase> trj(2);
         trj[1]=trj1;
         trj[2]=trj2;
         Trj2 trj_2(trj);
         trj_2.play(g2,in1(),in2());
      }
      else{
         cpmerror("statistics must have one or two entries");
      }
   }
   CPM_MZ
}

////////////////////////////// App4b ////////////////////////////
// Hubbard model

void CpmVQM2::App4b::doTheWork()
{
   Z mL=1;
   Word loc("CpmVQM2::App4b::doTheWork()");
   CPM_MA

   Z statistics=1,d,nPar;
   R gamma,showPsi,aHub=-1,bHub=1;
   bool pbc=true;
   V<R> waitTimes;

   Word file;
   V<Word> line1;
   V<Word> line2;
   V<Word> line3;
   V<Word> line4;

// reading data
// I choose to read all data in succession although for most settings
// in run_contol only a part of these data is needed in the code
// to be executed furtheron. Letting each control path read only those
// data that it really needs would create complicated interdependencies
// of the various control paths, resulting in the need for doublicating
// code. This also would complicate further extensions.

//////////////////////////////////////////////////////////////////////
   Word sec="about";
//////////////////////////////////////////////////////////////////////
   cpmrh(file);
   cpmrh(line1);
   cpmrh(line2);
   cpmrh(line3);
   cpmrh(line4);
```

```
//////////////////////////////////////////////////////////////////////
   sec="run preparation";
//////////////////////////////////////////////////////////////////////
   cpmrh(waitTimes);
   cpmrh(gamma);
   cpmrh(showPsi);
   cpmrh(statistics);

//////////////////////////////////////////////////////////////////////
   sec="system data";
//////////////////////////////////////////////////////////////////////

   cpmrhf(d);                                                        \

   cpmrhf(nPar);
   cpmrhf(aHub);
   cpmrhf(bHub);
   cpmrhf(pbc);

//////////////////////////////////////////////////////////

// setting the stage

   Color::setGamma(gamma);
   Graph g;
   g.setWithOrigin(0);
   Z i;
   R tw1=waitTimes[1];

 // defining the FieldOperator fop

   FieldOperator fop=FieldOperator::hub(aHub,bHub,d);
   Record rtc=(rch.getRecord())("run control","TrjCon");
          // Here the section 'run control' comes in
   rtc.prnOn(cpmdata);
   TrjCon tc(rtc);

   Z nSteps=tc.nSteps_;
   Z lim=statistics;
      // 1 for fermions, >1 for bosons
   Z test=16;
   R testRes=FieldState::test(test);
   R eps=1e-8;
   if (testRes>eps) cpmerror("There were errors in test");
   State s0(d);
   V<State> vs(nPar);
   for (i=vs.b();i<=vs.e();++i) vs[i]=s0.ranVal();
   FieldState fs(vs,lim,pbc);
   OFileStream out1(outFiles[1]);
   TrjDMI_FilOpr trj(fop,fs,tc);
```

```
   tc.prnOn(cpmcerr);
   trj.run(g,out1());
   cpmwait(tw1,2);
   CPM_MZ
}

//////////////////////// App5 ////////////////////////////////////////////

void CpmVQM2::App5::doTheWork()
// First experiences with the new class State2 (started 2006-07)
// Now (2006-09) grown into the most advanced experimental version for
// new research features such as time reversal and 'optical theorem'
{
   Z mL=1;
   Word loc("CpmVQM2::App5::doTheWork()");
   CPM_MA
//////////////////////////////////////////////////////////////////////
   Word sec("run control");
//////////////////////////////////////////////////////////////////////
   Z nSteps,iReversal,reversalMethod,nSmooth,nConcentrate,viewingPeriod,
      docuPeriod,imagingPeriod,ipolMethod=1,optThr=0;
   R tWait=0,gamma=1;
   cpmrh(nSteps);
   cpmrh(iReversal);
   cpmrh(reversalMethod);
   cpmrh(nSmooth);
   cpmrh(nConcentrate);
   cpmrh(viewingPeriod);
   cpmrh(docuPeriod);
   cpmrh(imagingPeriod);
   cpmrh(tWait);
   cpmrh(ipolMethod);
   cpmrh(optThr);
   cpmrh(gamma);

//////////////////////////////////////////////////////////////////////
   sec="system data";
//////////////////////////////////////////////////////////////////////
   Z i;
   Z d1=60,d2=60,stepMet=3;
   cpmrh(d1);
   cpmrh(d2);
   R size1=1.,size2=1.,massRel1,massRel2,vRel1,vRel2;
   R dtRel=0, xRel1=0, xRel2=0, wRel1=0, wRel2=0;
   R enrMagFac=1e9;
   cpmrh(xRel1);
   cpmrh(xRel2);
   cpmrh(wRel1);
   cpmrh(wRel2);
   bool enfNor=false,useSI=false,noRel=false,crossway=true;
```

```
   bool autoStep=false;
   cpmrh(size1);
   cpmrh(size2);
   cpmrh(noRel);
   cpmrh(massRel1);
   cpmrh(massRel2);
   cpmrh(vRel1);
   cpmrh(vRel2);
   R alpha,beta,mu,eps,p;
   cpmrh(alpha);
   cpmrh(beta);
   cpmrh(mu);
   cpmrh(eps);
   cpmrh(p);
   cpmrh(useSI);
   cpmrh(dtRel);
   cpmrh(enfNor);
   cpmrh(autoStep);
   cpmrh(crossway);
   cpmrh(enrMagFac);
   cpmrh(stepMet);
// start working
   Word units=useSI ? "SI":"1";
   OFileStream out1(outFiles[1]);
   Greg date;
   date.now();
   out1()<<"// ";
   date.writeFormatted(out1());
   out1()<<"// output of CpmVQM2::App5::doTheWork()"<<endl;
   out1()<<"units="<<units;
   QMBase::setUnits(units);

   QMBase::setSize(size1);
      // to get the right lattice related data for particle 1
   R m1= noRel ? massRel1 : massRel1*QMBase::mMax(d1);

   QMBase::setSize(size2); // to get the right output from mMax
   R m2=noRel ? massRel2 : massRel2*QMBase::mMax(d2);

   State2Data data;
   data.m1=m1;
   data.m2=m2;
   data.size1=size1;
   data.size2=size2;
   data.alpha=alpha;
   data.beta=beta;
   data.mu=mu;
   data.eps=eps;
   data.p=p;
   data.crossway=crossway;
```

```
   State2::set_d(data);

   // initial state:
   bool unitHeg=true; // aha heg = height
   R_Func gpsi1=R_Func::Gauss(xRel1,wRel1,unitHeg);
   R_Func gpsi2=R_Func::Gauss(xRel2,wRel2,unitHeg);

   State s1(gpsi1,d1);
   State s2(gpsi2,d2);
// That boosting instances of State correctly, the class
// QMBase has to be adjusted, needs e.g. member mass
   QMBase::setMass(m1);
   QMBase::setSize(size1);
   R v1=noRel ? vRel1 : vRel1*0.5*QMBase::vMax(d1);
   out1()<<"v1="<<v1<<endl;
      // 0.5 inserted since without this, vRel1=1
      // corresponds to a wave function mainly at
      // Nyquist frequency. Our representation of the Hamilton
      // operator is not very suitable for such strongly
      // varying wave functions
   s1=s1.boost(v1);
 //  s1.reg_(); // no significant influence
   Observable T1("eKinF",d1);
   R enr1=T1.expVal(s1);
   out1()<<"enr1="<<enr1<<endl;

   QMBase::setMass(m2);
   QMBase::setSize(size2);
   R v2=noRel ? vRel2 :  vRel2*0.5*QMBase::vMax(d2);
   out1()<<"v2="<<v2<<endl;
   s2=s2.boost(v2);
 //  s2.reg_();
   Observable T2("eKinF",d2);
   R enr2=T2.expVal(s2);
   out1()<<"enr2="<<enr2<<endl;

   Graph gr;
   Z sym=1;
   cpmrh(sym);
   cpmrh(crossway);
      // 0 for non-identical particles
      // -1 for fermions
      // else for bosons
   if (d1!=d2){
      if (sym!=0){ // would result in index errors
         sym=0;
         cpmwarning("sym changed to 0 since d1!=d2");
         cpmwait(1);
      }
```

```
  }
  State2 s(s1,s2);
  if (sym!=0){
     State2 sSym(s2,s1);
     if (sym==-1) sSym*=-1;
     s+=sSym;
     s.nor_(1);
  }
  // now s1 and s2 are no longer needed as State-instances
  State2Trj st(s);
  st.revMet_=reversalMethod;
  for (i=0;i<nConcentrate;++i){
     st.concentrate_();
     st.x().mark(gr,ipolMethod, gamma);
     gr.display();
     cpmwait(tWait+2);
  }
  st.smooth_(3);
  State2Trj stIni(st);
  C enr3=st.enr();
  out1()<<"complex en3 ="<<enr3<<endl;
  R enrInt=enr3.real()-enr1-enr2;
  out1()<<"enrInt ="<<enrInt<<endl;
  out1()<<"This should valish for alpha==beta==0"<<endl;
  R dt0=noRel ? dtRel : dtRel*st.dtLimit();
 // R epsDifferentiation=1e-6;
  R epsDifferentiation=1e-1;
  R dtDiff=dt0*epsDifferentiation;
  out1()<<"dt0="<<dt0<<endl;
  R tTot=nSteps*dt0;
  R turns=tTot*v1/size1;
  out1()<<"turns="<<turns<<endl;
  Z nDoc=docuPeriod>0 ? cpmround(1.*nSteps/docuPeriod) : 0;
  R_Matrix docMat(5,nDoc);
  R_Matrix doc1Mat(3,nSteps);
  R_Matrix doc2Mat(3,nSteps);
  R_Matrix doc3Mat(3,nSteps);

  CyclicAlarm mark(viewingPeriod);
  CyclicAlarm docu(docuPeriod);
  CyclicAlarm img(imagingPeriod);

  R t, er0, er0Inv;
  R nInv=nSteps<2 ? 1. : 1./(nSteps-1);
  Z iDoc=1, iImg=1;
  st.stepMet_=stepMet;
  st.enfNor_=enfNor;
  st.autoStep_=autoStep;
  R spAct=0,spPrev,nAct=0,nPrev=0,nPrevPrev=0,mr,nx,xx,spx;
  // mr is an interesting quantity, it is very small for regular
```

```
// time steps and explodes if the norm and the energy do so. Since it
// involves data from three consecutive time steps its initialization
// is a bit complicated.
C xv;
State2 sIni=st.x();
Z k;
R dt=(optThr==0? dt0 : dt0-dtDiff);
Z kMax=(optThr==0? 1 : 3);
for (k=1;k<=kMax;++k){
   st=stIni;
   st.set_dt_(dt);
   for (i=0,t=0.5*dt; i<nSteps; ++i,t+=dt){
      R ip=i*nInv;
      cpmprogress("loop",ip,1);
      nx=st.x().absVal();
      xx=(st.x()|st.x()).real();
      xv=(st.x()|st.v());
      spx=(st.v()|st.x()).real();
      R dn=nx-1.;
      if (i==0){
         nPrevPrev=nx;
         mr=0;
      }
      else if (i==1){
         nPrev=nx;
         mr=0;
      }
      else if (i==2){
         nAct=nx;
         spAct=spx;
         mr=0;
      }
      else{
         nPrevPrev=nPrev;
         nPrev=nAct;
         nAct=nx;
         spPrev=spAct;
         spAct=spx;
      // mr=(nAct-nPrevPrev)*nPrev-2*dt*spPrev;
         mr=nAct-nPrevPrev-2*dt*spPrev; // best option
      // mr=nAct*nAct-nPrevPrev*nPrevPrev-4*dt*spPrev;
      }
      C ste=st.enr();
      R er=ste.real();
      if (i==0){
         er0=er;
         er0Inv=er0==0 ? 1  : 1./er0;
      }
      R der=(er-er0)*er0Inv;
      R ei=ste.imag()*er0Inv;
```

```
      cpmvalue("dn",dn,2);
      cpmvalue("der",der,3);
      cpmvalue("ei",ei,4);
      der*=enrMagFac; // for docu graphics
      if (i==iReversal) st.con_();
      if (mark()){
         st.x().mark(gr,ipolMethod, gamma);
         gr.display();
         cpmwait(tWait);
      }
      if (docu()&& iDoc<=nDoc){
         docMat[1][iDoc]=t;
         docMat[2][iDoc]=dn;
         docMat[3][iDoc]=der;
         docMat[4][iDoc]=ei;
         docMat[5][iDoc]=mr;
         iDoc++;
      }
      if (img()){
         Word imf="app5_";
         gr.write(imf&cpm(iImg++,3));
      }
      if (optThr!=0){
         Z ipp=i+1;
         if (k==1){
            doc1Mat[1][ipp]=xx;
            doc1Mat[2][ipp]=xv.real();
            doc1Mat[3][ipp]=xv.imag();
         }
         else if (k==2){
            doc2Mat[1][ipp]=xx;
            doc2Mat[2][ipp]=xv.real();
            doc2Mat[3][ipp]=xv.imag();
         }
         else if (k==3){
            doc3Mat[1][ipp]=xx;
            doc3Mat[2][ipp]=xv.real();
            doc3Mat[3][ipp]=xv.imag();
         }
         else cpmerror("bad value "&cpm(k)&" of k");
      }
      st.step_();
   }
   if (nSmooth>0){
      for (i=0;i<nSmooth;++i){
         st.smooth_(1);
         st.x().mark(gr,ipolMethod, gamma);
         gr.display();
         cpmwait(tWait+0.5);
      }
```

```
         cpmwait(tWaitWorkScreen_);
      }
      dt+=dtDiff;
   }

   if (optThr==0){
      State2 sFin=st.x();
      sFin*=-1.;
      sFin.con_();
      R diff=(sIni+sFin).norm();
      cpmmessage("diff="&cpm(diff));
      if (docuPeriod>0) gr.show(docMat);
   }
   else{ // curve which shows the validity of the 'optical theorem'
      R_Matrix docX(4,nSteps);
      R magnificationFactor=5e7;
      for (i=1;i<=nSteps;++i){
         R b=(doc2Mat[2][i]-doc2Mat[2][1])*(i-1)*dt0;
         R a1=doc1Mat[1][i]-doc1Mat[1][1];
         R a2=doc2Mat[1][i]-doc2Mat[1][1];
         R a3=doc3Mat[1][i]-doc3Mat[1][1];
         R a=dt0*(a3-a1)/(4*dtDiff)-a2;
         docX[1][i]=i;
        // docX[2][i]=a;
        // docX[3][i]=(a-b)*magnificationFactor;
         docX[2][i]=a1;
         docX[3][i]=a2;
         docX[4][i]=a3;
      }
      gr.show(docX);
   }
   cpmwait(tWaitWorkScreen_);
   CPM_MZ
}

void CpmVQM2::App6::doTheWork()
// Initial state of particle 1 is a Gaussian and
// initial state of particle 2 is eigenstate of
// of the harmonic oscillator built into State2.
// Aim is to see excitation or de-excitation of this
// state as a result of interaction with particle 1.
// This should be used for the image production for the publication.
// 2006-11-03: visualization of the 'reduced' states added.
{
   Z mL=1;
   Word loc("CpmVQM2::App6::doTheWork()");
   CPM_MA

// Getting input from ini file
//////////////////////////////////////////////////////////////////////
```

```
   Word sec("run control");
//////////////////////////////////////////////////////////////////////
   Z nSteps,viewingPeriod,docuPeriod,imagingPeriod,ipolMethod=1,
      iViewReducedState=1;
   R tWait=0,gamma=1,beginFadingOut,endFadingOut,
      tWaitDocMat1,tWaitDocMat2,tWaitDocMat3,magViewReducedState;
   V<Z> viewLevels;
   bool showAsCurves=false;
   cpmrh(nSteps);
   cpmrh(beginFadingOut);
   cpmrh(endFadingOut);
   cpmrh(viewingPeriod);
   cpmrh(docuPeriod);
   cpmrh(viewLevels);
   cpmrh(showAsCurves);
   cpmrh(iViewReducedState);
   cpmrh(magViewReducedState);
   cpmrh(tWaitDocMat1);
   cpmrh(tWaitDocMat2);
   cpmrh(tWaitDocMat3);
   cpmrh(imagingPeriod);
   cpmrh(tWait);
   cpmrh(ipolMethod);
   cpmrh(gamma);

//////////////////////////////////////////////////////////////////////
   sec="system data";
//////////////////////////////////////////////////////////////////////

   Z d1, d2, stepMet=3, ePowers=0, box1=1, box2=1;

   R size1=1., size2=1., massRel1, massRel2, vRel1,
      alpha, beta, mu, eps, p, dtRel, xRel1, wRel1,
      misFac=1, tWaitH2=0, tWait_so=0, tWaitV12, enrMagFac=1,
      trnMagFac=1, trnVisExp=1, normTol=-1,
      phaseJitter1=0,phaseJitter2=0;
   V<Z> excLevels;

   bool enfNor=false, useSI=false,
      crossway=true, autoStep=false;

   cpmrh(d1);
   cpmrh(d2);
   cpmrh(size1);
   cpmrh(size2);
   cpmrh(box1);
   cpmrh(box2);
   cpmrh(massRel1);
   cpmrh(massRel2);
   cpmrh(vRel1);
```

```
   cpmrh(xRel1);
   cpmrh(wRel1);
   cpmrh(excLevels);

   cpmrh(alpha);
   cpmrh(beta);
   cpmrh(mu);
   cpmrh(eps);
   cpmrh(p);
   cpmrhf(enfNor);
   cpmrhf(normTol);
   cpmrhf(phaseJitter1);
   cpmrhf(phaseJitter2);
   cpmrh(useSI);
   cpmrh(dtRel);
   cpmrh(ePowers);
   cpmrh(misFac);
   cpmrh(tWaitH2);
   cpmrh(tWait_so);
   cpmrh(tWaitV12);
   cpmrh(crossway);
   cpmrh(enrMagFac);
   cpmrh(trnMagFac);
   cpmrh(trnVisExp);
   cpmrh(stepMet);
   cpmrh(autoStep);
//////////////////// we are in App6::doTheWork() ////////////////////
// Starting the work
   OFileStream out1(outFiles[1]);
      // output during reading data and doing
      // preparing computations
   OFileStream out2(outFiles[2]);
      // data concerning nu and epsilon
   OFileStream out3(outFiles[3]);
      // data concerning reduced states
   Greg date;
   date.now();
   out1()<<"// ";
   date.writeFormatted(out1());
   out1()<<"// output of CpmVQM2::App6::doTheWork()"<<endl;
   Word units=useSI ? "SI":"NAT";
   out1()<<"units="<<units;
   QMBase::setUnits(units);
   QMBase::setSize(size1);
      // to get the right lattice related data for particle 1
   out1()<<"size1="<<size1<<endl;
   R m1=massRel1*QMBase::mMax(d1);
   out1()<<"m1="<<m1<<endl;

   QMBase::setSize(size2); // to get the right output from mMax
```

```
   out1()<<"size2="<<size2<<endl;
   R m2=massRel2*QMBase::mMax(d2);
   out1()<<"m2="<<m2<<endl;

   State2Data data;
   data.m1=m1;
   data.m2=m2;
   data.size1=size1;
   data.size2=size2;
   data.box1=box1;
   data.box2=box2;
   data.alpha=alpha;
   out1()<<"alpha="<<alpha<<endl;
   data.beta=beta;
   out1()<<"beta="<<beta<<endl;
   data.mu=mu;
   data.eps=eps;
   data.p=p;
   data.crossway=crossway;
//////////////////// we are in App6::doTheWork() ////////////////////
// initial state
// Particle 1 has Gauss shape
   bool unitHeight=true; // avoids normalization factor
   R_Func gpsi0=R_Func::Gauss(0,wRel1,unitHeight);
   R_Func gpsi1=R_Func::Gauss(xRel1,wRel1,unitHeight);
   State s1Prel(gpsi0,d1);
      // will be needed as a state at center,
      // not boosted and not shifted
   State s1(gpsi1,d1);
// That boosting instances of State correctly, the class
// QMBase has to be adjusted, needs e.g. member mass
   QMBase::setMass(m1);
   QMBase::setSize(size1);
   R v1=vRel1*0.5*QMBase::vMax(d1);
   out1()<<"boost velocity of particle 1: "<<v1<<endl;
      // 0.5 inserted since without this vRel1=1
      // corresponds to a wave function mainly at
      // Nyquist frequency. Our representation of the Hamilton
      // operator is not very suitable for such strongly
      // varying wave functions
   Observable T1("eKinF",d1);
   R enr_s1=T1.expVal(s1);
   out1()<<"energy of particle1 prior to boost: "<<enr_s1<<endl;
   s1=s1.boost(v1);
  // s1=s1.smooth();
   enr_s1=T1.expVal(s1);
   out1()<<"energy of particle1 after boost: "<<enr_s1<<endl;
   R enrKin1=m1*0.5*v1*v1;
   out1()<<"kinetic energy of particle 1 by mass and velocity:\
    "<<enrKin1<<endl;
```

```
   R enr1=0.5*(enr_s1+enrKin1); // for stability (paranoia?)
   out1()<<"energy of particle 1 (mean of both): "<<enr1<<endl;

// Particle 2 should be an eigenstate of the harmonic
// oscillator represented by potential

   QMBase::setMass(m2);
   QMBase::setSize(size2);
   Observable T2("eKinF",d2);
      // F finite difference form
      // class observable --untouched for years-- workes perfectly
      // in the present context.

   R beta0=QMBase::lambdaOsc(d2);
   out1()<<"natural oscillator coupling constant: "<<beta0<<endl;
   bool adjustBeta=beta!=0 && !excLevels.isVoid();
   if (adjustBeta){
      Z dn=excLevels.last()-excLevels.first();
      cpmassert(dn>0,loc);
      R fac=enr1*QMBase::get_hDiracInv()/dn;
      R betaAdjusted=0.5*m2*fac*fac;
      data.beta=betaAdjusted;
      out1()<<"betaAdjusted="<<betaAdjusted<<endl;
   }
   R_Func pot=R_Func::x2*data.beta*misFac;
      // Potential with beta already set to the final value
      // harmonic oscillator potential, the factor misFac allows to
      // use here a different coupling constant than the one used
      // in the potential of State2

   Observable V2("x",d2,pot);
      // function pot of position operator
   Observable H2=T2+V2;
      // Hamiltonial of a harmonic oscillator

   Graph gr;

   if (tWaitH2>0){
      H2.show(gr);
      cpmwait(tWaitH2);
      gr.clear();
   }
//////////////////// we are in App6::doTheWork() ////////////////////

   SpectralObject so;
   {  // encapsulating the heavy initialization of so
      bool enforceSymmetry=true;
         // our H2 is known to be symmetric
      bool test=false;
         // true would consume a lot of time but worked in my test cases
```

```
      // perfectly
   Z iterMax=128;
   R eps0=0;
   R eps1=1e-7,eps2=1e-6; // test values from the past
   so=H2.spectralRepresentation(enforceSymmetry,
   test,iterMax,eps0,eps1,eps2); // this is a big beast
   so=so.order(); // as so is created, its order is
   // counter intuitive (index 0 should correspond to the
   // ground-state). I decided not to change function
   // spectralRepresentation
   // to include ordering, in order not to break old code.
   // So, a new memberfunction of SpectralObject does the job
 }
 so.prnOn(out1());
 if (tWait_so>0){
   so.showSpectrum(gr);
   cpmwait(tWait_so);
   gr.clr();
   so.show(gr);
   cpmwait(tWait_so);
   gr.clr();
 }
 Z iIni=excLevels.first(), iFin=excLevels.last();
 State s2=so[iIni];
 s2=s2.symmetrize(); // experiment
// s2=s2.smooth();
 Z i,k;
 V<State> eStates(excLevels.dim());
 Z vld=viewLevels.dim();
 V<bool> bv(vld,false);
 for (i=1;i<=vld;++i) bv[i]=(viewLevels[i]>=0);
 viewLevels=viewLevels.select(bv); // using non-negative levels only
 vld=viewLevels.dim();
 V<State> vStates(viewLevels.dim());
 for (i=eStates.b(); i<=eStates.e(); ++i) eStates[i]=so[excLevels[i]];
 for (i=vStates.b(); i<=vStates.e(); ++i) vStates[i]=so[viewLevels[i]];
 if (adjustBeta){
   // only in this case there are two eigenvectors of H2 in the game
   R enrTrn=so(iFin).real()-so(iIni).real();
   // energy gap to the next higher excited state if there is
   // one
   out1()<<"transition energy of particle 2: "<<enrTrn<<endl;
   // this needs to be related to the kinetic energy of incoming
   // particle 1:

   // the enr1!=0
   R enrRelDiff=(enr1-cpmabs(enrTrn))/enr1;
   out1()<<"enrRelDiff="<<enrRelDiff<<
      " this should be small, otherwise the adjustment mechanism\
       failed"<<
```

```
         endl;
      State s2Fin=so[iFin];
      State2::set_d(data);
         // we need masses and sizes
      State2 sIni(s1Prel,s2);
      State2 sFin(s1Prel,s2Fin);
      C trnMatElm0=(sIni|sIni.V12());
      C trnMatElm1=(sFin|sIni.V12());
         // transition matrix element. Notice that the interaction
         // operator acts on State2, so that we have to have a
         // 'one particle content' in the state. s1Prel is a good
         // candidate for giving V12 the chance of a high contribution.
      out1()<<"Transition matrix element: "<<trnMatElm1;
      out1()<<"Stay at home matrix element: "<<trnMatElm0;
// We now multiply alpha by such a positve factor so that
// an initial value 1 gives a natural value to the new alpha
      if (alpha!=0){// we need a value in order get the sign
         R tmeR=trnMatElm1.absVal();
         cpmassert(tmeR>0,loc);
         R fac=1./tmeR;
         out1()<<"Adjustment factor fac="<<fac<<endl;
         if (fac<1000){
            alpha*=fac;
               // alpha (not data.alpha) will be used as the reference
               // for the fading mechanism.
            data.alpha=alpha;
            out1()<<"Adjusted alpha="<<data.alpha<<endl;
         }
         else{
            out1()<<"Alpha was not adjusted. alpha="<<data.alpha<<endl;
         }
      }
   }
// From now on there are intances of State2 so that 'data' becomes
// effective
//////////////////// we are in App6::doTheWork() ////////////////////

   State2::set_d(data);
   State2 s(s1,s2);
   s.nor_();
   if (tWaitV12>0){ // showing the interaction potential V12
      // by applying it to a flat state
      State s1Flat(d1,C(1.));
      State s2Flat(d2,C(1.));
      State2 sFlat(s1Flat,s2Flat);
      State2 sGraph=sFlat.V12();
      sGraph.mark(gr,ipolMethod, gamma);
      gr.display();
      cpmwait(tWaitV12);
   }
```

```
   // now s1 and s2 are no longer needed as State-instances
   State2Trj st(s);
   st.setPhaseJitter_(phaseJitter1,phaseJitter2);
   R dtLim= ePowers>0 ? s.dtSup(ePowers,tWait,gamma) : st.dtLimit();
      // dtSup is the superior method
   out1()<<"dtLim="<<dtLim<<endl;
   R dt=dtRel*dtLim;
   out1()<<"dt="<<dt<<endl;
   R tTot=nSteps*dt;
   out1()<<"tTot="<<tTot<<endl;
   R turns=tTot*v1/size1;
   out1()<<"turns="<<turns<<endl;

   bool fading=false;
   R tbf=2*tTot, tef=tbf;

   if (beginFadingOut>=0 && endFadingOut>=0 &&
       beginFadingOut<=1 && endFadingOut<=1 &&
       beginFadingOut<endFadingOut)
   {
      fading=true;
      tbf=tTot*beginFadingOut;
      tef=tTot*endFadingOut;
   }

   Z nDocu=docuPeriod>0 ? cpmround(1.*nSteps/docuPeriod) : 0;
 //  R_Matrix docMat1(4,nDocu);
   R_Matrix docMat1(2,nDocu);
   R_Matrix docMat2(1+eStates.dim(),nDocu);
   R_Matrix docMat3(3,nDocu);

   CyclicAlarm mark(viewingPeriod);
   CyclicAlarm docu(docuPeriod);
   CyclicAlarm img(imagingPeriod);
//////////////////// we are in App6::doTheWork() ////////////////////
   C xv0;
   R t,xx0,xv0AbsInv,xx0AbsInv;

   Z iDoc=1, iImg=1;
   R nInv=1./nSteps;
   st.stepMet_=stepMet;
   st.autoStep_=autoStep;
   st.enfNor_=enfNor;
   st.normTol_=normTol;
   st.set_dt_(dt); //
   R scl=1./(dt*dt);
   Frame frx;
   vld=viewLevels.dim(); // may be 0
   Frames frv;
   V<Graph> grv(vld);
```

```
   if (vld>0){
      frv=Frames(frx,vld,1);
      for (i=grv.b();i<=grv.e();++i) grv[i]=Graph(frv[i][1]);
   }
   for (i=0,t=0; i<nSteps; ++i,t+=dt){
      R ip=i*nInv;
      cpmprogress("loop",ip,1);
      if (fading && t>=tbf && t<=tef){
         R pf=(t-tbf)/(tef-tbf);
         R fadeFac=R_Func::hill()(pf);
         data.alpha=alpha*fadeFac;
            // fading should be relative to the adjusted value
            // of alpha
         State2::set_d(data);
      }
      R xx=st.xx();
      C xv=st.xv();
      if (i==0){
         xx0=xx;
         xv0=xv;
         R xxa=cpmabs(xx0);
         R xva=xv0.absVal();
         xx0AbsInv=(xxa==0 ? 1 : 1./xxa);
         xv0AbsInv=(xva==0 ? 1 : 1./xva);
      }
      R nu=xx-xx0;
      C epsilon=xv-xv0;

      R nuRel=nu*xx0AbsInv;
      //C epsilonRel=epsilon*xv0AbsInv;
      C epsilonRel=epsilon; // experiment

      R nuScl=nu*scl;
      C epsilonScl=epsilon*scl;

// since epsilon is complex we divide it into real and imaginary part
      R epsRel1=epsilonRel[1];
      R epsRel2=epsilonRel[2];
      R epsScl1=epsilonScl[1];
      R epsScl2=epsilonScl[2];
      cpmvalue("nuRel",nuRel,2); // nu and epsilon refer to the
      // publication
      cpmvalue("alpha",data.alpha,3);
      epsRel2*=enrMagFac; // for docu graphics
      if (mark()){
         st.x().mark(gr,ipolMethod, gamma);
         gr.display();
         cpmwait(tWait);
      }
      if (docu()){
```

```
      // docMat1,out2()
         docMat1[1][iDoc]=t;
         docMat1[2][iDoc]=nu;
         docMat3[1][iDoc]=t;
         docMat3[2][iDoc]=epsRel1;
         docMat3[3][iDoc]=epsRel2;
      //   docMat1[3][iDoc]=epsRel1;
      //   docMat1[4][iDoc]=epsRel2;
         out2()<<t<<" ";
// relative values
         out2()<<nuRel<<" ";
         out2()<<epsRel1<<" ";
         out2()<<epsRel2<<" ";
// scaled values
// Important new aspect of scaling: It is not necessary to create the
// data for various values of of the time step in a single program run
// since gnuplot allows to represent data from different files in a
// single diagram.
         out2()<<nuScl<<" "; // scl: scaled
         out2()<<epsScl1<<" "; // 1: real part
         out2()<<epsScl2<<" "; // 2: imaginary part
         out2()<<endl;
      // docMat2, out3()
         docMat2[1][iDoc]=t;
// listing the amplidudes of the states associated with
// the oscillator states determined by sList
         out3()<<t<<" ";
         for (k=eStates.b();k<=eStates.e();++k){
            State xredk=st.x().red2(eStates[k].con());
               // 2006-11-14: eStates[k] changed to eStates[k].con()
            R wk=xredk.absSqr();
            if (trnVisExp!=1.0) wk=pow(wk,trnVisExp);
            if (k%2==0){
               wk*=trnMagFac;
            }
            docMat2[1+k][iDoc]=wk;
            out3()<<wk<<" ";
         }
         out3()<<endl;
// graphical part of docu, only active if vStates is not void
         for (k=vStates.b();k<=vStates.e();++k){
            State xredk=st.x().red2(vStates[k].con());
            if (showAsCurves){ // showing the reduced state directly
               if (iDoc==iViewReducedState){
                  // flexible method for avoiding the unsteady
                  // screen presentation resulting from auto-scaling
                  // each frame.
                  for (Z l=grv.b();l<=grv.e();l++){
                     grv[l].setAutoScale(0);
                     grv[l].scaleY(magViewReducedState);
```

```
                        // the Y-range at iDoc==iViewReducedState might
                        // be not large enough for the frames to follow.
                        // So we magnify by a suitable 'security factor'
                     }
                  }//iDoc==iViewReducedState
                  xredk.showAsCurves(grv[k]);
                  cpmwait(tWait);
               } // if (showAsCurves)
               else{
                  State2 sShow(xredk,vStates[k]);
                     // showing an interesting State2-version of
                     // the reduced state.
                  sShow.mark(gr,ipolMethod, gamma);
                  gr.display();
                  cpmwait(tWait);
               }
            }
            iDoc++;
         } // if docu()
         if (img()){
            Word imf="app6_";
            Word imn=cpm(iImg,3);
            out1()<<"image # "<<iImg<<" belongs to t="<<t<<endl;
            gr.write(imf&imn);
            iImg++;
         }
         st.step_();
      }
      if (docuPeriod>0 && tWaitDocMat1>0 ){
         gr.show(docMat1);
         cpmwait(tWaitDocMat1);
      }
      if (docuPeriod>0 && tWaitDocMat2>0 ){
         gr.show(docMat2);
         cpmwait(tWaitDocMat2);
      }
      if (docuPeriod>0 && tWaitDocMat3>0 ){
         gr.show(docMat3);
         cpmwait(tWaitDocMat3);
      }
      cpmwait(tWaitWorkScreen_);
      CPM_MZ
   }

   void CpmVQM2::App7::doTheWork()
   // This is a simple program aiming at testing the consistency
   // classes State, Observable, SpectralObject and particularly
   // the boost operation. Result: a state at rest gets the correct
   // velocity and kinetic energy by the boost operation.
   // The reason for making this part was to make sure that the strange
```

```
// behavior found in App6 is not caused by State::boost.
// This is clear now!
{
   Z mL=2;
   Word loc("CpmVQM2::App7::doTheWork()");
   CPM_MA
//////////////////////////////////////////////////////////////////////
   Word sec("run control");
//////////////////////////////////////////////////////////////////////
   Z viewingPeriod;
   R travelRel, tWait=0;
   cpmrh(travelRel);
   cpmrh(viewingPeriod);
   cpmrh(tWait);

//////////////////////////////////////////////////////////////////////
   sec="system data";
//////////////////////////////////////////////////////////////////////

   Z d1;
   R size1=1.,massRel1,vRel1;
   R dtRel=0, xRel1=0, wRel1=0;
   bool useSI=false,noRel=false;

   cpmrh(d1);
   cpmrh(size1);
   cpmrh(noRel);
   cpmrh(massRel1);
   cpmrh(xRel1);
   cpmrh(wRel1);
   cpmrh(vRel1);
   cpmrh(useSI);
   cpmrh(dtRel);
// start working
   Word units=useSI ? "SI":"1";
   OFileStream out1(outFiles[1]);
   Greg date;
   date.now();
   out1()<<"// ";
   date.writeFormatted(out1());
   out1()<<"// output of CpmVQM2::App7::doTheWork()"<<endl;
   out1()<<"units="<<units;
   QMBase::setUnits(units);
   QMBase::setSize(size1);
      // to get the right lattice related data for particle 1
   R m1= noRel ? massRel1 : massRel1*QMBase::mMax(d1);
   // That boosting instances of State correctly, the class
// QMBase has to be adjusted, needs e.g. member mass
   QMBase::setMass(m1);
   out1()<<"d1="<<d1<<endl;
```

```
   out1()<<"size1="<<size1<<endl;
   out1()<<"m1="<<m1<<endl;

// kinetic energy as spectral object, which is the most efficient
// type of object for implementing dynamics

   bool ipt=CpmRootX::Z1(d1).isPowOfTwo();
// two Hamiltonians to select (indirectly via d1)
   Observable ekin= ipt? Observable("eKin",d1) : Observable("eKinF",d1);
// initial state:
   bool unitHeg=true; // aha heg = height
   R_Func gpsi1=R_Func::Gauss(xRel1,wRel1,unitHeg);
   State s1(gpsi1,d1);
   R enr=(s1|ekin*s1).real();
   out1()<<"Kinetic energy befor boost: "<<enr<<endl;
   R v1=noRel ? vRel1 : vRel1*0.5*QMBase::vMax(d1);
   out1()<<"v1="<<v1<<endl;
      // 0.5 inserted since without this vRel1=1
      // corresponds to a wave function mainly at
      // Nyquist frequency. Our representation of the Hamilton
      // operator is not very suitable for such strongly
      // varying wave functions
   s1=s1.boost(v1);
   enr=(s1|ekin*s1).real();
   out1()<<"Kinetic energy after boost: "<<enr<<endl;
   enr=0.5*m1*v1*v1;
   out1()<<"Kinetic energy from mass and velocity: "<<enr<<endl;
   Graph gr;
   R dt=noRel ? dtRel : dtRel*QMBase::tRef(d1);
   out1()<<"dt direct ="<<dt<<endl;
   R travel=noRel ? travelRel : travelRel*size1;
   cpmassert(v1!=0.,loc);
   R tTot=travel/v1;
   Z nSteps=cpmround(tTot/dt);
   out1()<<"nSteps ="<<nSteps<<endl;
   s1.showAsCurves(gr);
   cpmwait(1+tWait);
      // inspecting the initial state
   if (nSteps>0){
      dt=tTot/nSteps;
      out1()<<"dt adjusted ="<<dt<<endl;
      CyclicAlarm cyc(viewingPeriod);
      R nInv=1./nSteps;
      SpectralObject so;
      {  // encapsulating the heavy initialization of so
         bool enforceSymmetry=false;
         // our H2 is known to be symmetric
         bool test=false;
         // true would consume a lot of time but worked in my test cases
         // perfectly
```

```
         Z iterMax=128;
         R eps0=0;
         R eps1=1e-7,eps2=1e-6; // test values from the past
         so=ekin.spectralRepresentation(enforceSymmetry,
         test,iterMax,eps0,eps1,eps2); // this is a big beast
      }
      so.expi(-dt);
         // very efficient way of letting free Hamiltonian
         // dynamics act on the boosted state
      for (Z i=0; i<nSteps; ++i){
         R ip=i*nInv;
         cpmprogress("loop",ip,1);
         s1=so*s1; // notice that rounding errors acumulate
         if (cyc()||i==(nSteps-1)){
            s1.showAsCurves(gr);
            cpmwait(tWait);
         }
      }
   }
   cpmwait(tWaitWorkScreen_);
   CPM_MZ
}

//////////////////////////// CpmVQM2::App8 ////////////////////////////////

void CpmVQM2::App8::doTheWork()
// Initial states of particle 1 and 2 are Gaussian
// Made from App5 (2006-09-06)
// Covering a time interval with a few runs of different step size.
// Visualizing the scaling behavior.
{
   Z mL=1;
   Word loc("CpmVQM2::App8::doTheWork()");
   CPM_MA
//////////////////////////////////////////////////////////////////////
   Word sec("run control");
//////////////////////////////////////////////////////////////////////
   Z nSteps,viewingPeriod,nRuns,curveRes;
     //,iReversal,reversalMethod,nSmooth,nConcentrate,
     // docuPeriod,imagingPeriod,ipolMethod=1,optThr=0;
   R fac_dt=1,tWait=0,gamma=1;
   V<R> timeRange;
   cpmrh(nSteps);
   cpmrh(nRuns);
   cpmrh(viewingPeriod);
   cpmrh(fac_dt);
   cpmrh(tWait);
   cpmrh(gamma);
   cpmrh(timeRange);
   cpmrh(curveRes);
```

```
//////////////////////////////////////////////////////////////////////
   sec="system data";
//////////////////////////////////////////////////////////////////////
   Z d1=60,d2=60,stepMet=3;
   cpmrh(d1);
   cpmrh(d2);
   R size1=1.,size2=1.,massRel1,massRel2,vRel1,vRel2;
   R dtRel=0, xRel1=0, xRel2=0, wRel1=0, wRel2=0;
   cpmrh(xRel1);
   cpmrh(xRel2);
   cpmrh(wRel1);
   cpmrh(wRel2);
   bool useSI=false,noRel=false,crossway=true;
   bool autoStep=false;
   cpmrh(size1);
   cpmrh(size2);
   cpmrh(noRel);
   cpmrh(massRel1);
   cpmrh(massRel2);
   cpmrh(vRel1);
   cpmrh(vRel2);
   R alpha,beta,mu,eps,p;
   cpmrh(alpha);
   cpmrh(beta);
   cpmrh(mu);
   cpmrh(eps);
   cpmrh(p);
   cpmrh(useSI);
   cpmrh(dtRel);
   cpmrh(autoStep);
   cpmrh(crossway);
   cpmrh(stepMet);
// start working
   Word units=useSI ? "SI":"1";
   OFileStream out1(outFiles[1]);
   Greg date;
   date.now();
   out1()<<"// ";
   date.writeFormatted(out1());
   out1()<<"// output of CpmVQM2::App8::doTheWork()"<<endl;
   out1()<<"units="<<units;
   QMBase::setUnits(units);

   QMBase::setSize(size1);
      // to get the right lattice related data for particle 1
   R m1= noRel ? massRel1 : massRel1*QMBase::mMax(d1);

   QMBase::setSize(size2); // to get the right output from mMax
   R m2=noRel ? massRel2 : massRel2*QMBase::mMax(d2);
```

```
   State2Data data;
   data.m1=m1;
   data.m2=m2;
   data.size1=size1;
   data.size2=size2;
   data.alpha=alpha;
   data.beta=beta;
   data.mu=mu;
   data.eps=eps;
   data.p=p;
   data.crossway=crossway;

   State2::set_d(data);

   // initial state:
   bool unitHeg=true; // aha heg = height
   R_Func gpsi1=R_Func::Gauss(xRel1,wRel1,unitHeg);
   R_Func gpsi2=R_Func::Gauss(xRel2,wRel2,unitHeg);

   State s1(gpsi1,d1);
   State s2(gpsi2,d2);
// That boosting instances of State correctly, the class
// QMBase has to be adjusted, needs e.g. member mass
   QMBase::setMass(m1);
   QMBase::setSize(size1);
   R v1=noRel ? vRel1 : vRel1*0.5*QMBase::vMax(d1);
   out1()<<"v1="<<v1<<endl;
      // 0.5 inseted since without this vRel1=1
      // corresponds to a wave function mainly at
      // Nyquist frequency. Our representation of the Hamilton
      // operator is not very suitable for such strongly
      // varying wave functions
   s1=s1.boost(v1);
 //  s1.reg_(); // no significant influence
   Observable T1("eKinF",d1);
   R enr1=T1.expVal(s1);
   out1()<<"enr1="<<enr1<<endl;

   QMBase::setMass(m2);
   QMBase::setSize(size2);
   R v2=noRel ? vRel2 :  vRel2*0.5*QMBase::vMax(d2);
   out1()<<"v2="<<v2<<endl;
   s2=s2.boost(v2);
   Observable T2("eKinF",d2);
   R enr2=T2.expVal(s2);
   out1()<<"enr2="<<enr2<<endl;

   Graph gr;
   Z sym=1;
```

```
   cpmrh(sym);
   cpmrh(crossway);
      // 0 for non-identical particles
      // -1 for fermions
      // else for bosons
   if (d1!=d2){
      if (sym!=0){ // would result in index errors
         sym=0;
         cpmwarning("sym changed to 0 since d1!=d2");
         cpmwait(1);
      }
   }
   State2 s(s1,s2);
   if (sym!=0){
      State2 sSym(s2,s1);
      if (sym==-1) sSym*=-1;
      s+=sSym;
      s.nor_(1);
   }
   // now s1 and s2 are no longer needed as State-instances
   State2Trj st(s);
///////////////// reminder: CpmVQM2::App8 ////////////////////////////////
   R xx0=(st.x()|st.x()).real();
   cpmdebug(xx0);
   C xv0=(st.x()|st.v());
   R xvr0=xv0.real();
   R xvi0=xv0.imag();
   State2Trj st0(st);

   R dt0=noRel ? dtRel : dtRel*st.dtLimit();
   out1()<<"dt0="<<dt0<<endl;
   R tTot=nSteps*dt0;
   R turns=tTot*v1/size1;
   R tTotInv=1./tTot;
   out1()<<"turns="<<turns<<endl;
   st.stepMet_=stepMet;
   st.autoStep_=autoStep;

   R dt=dt0;
   V< F<R,R> > fxx(nRuns);
   V< F<R,R> > fxv(nRuns);
   V< F<R,R> > fxxxv(2*nRuns);

   // for each run (indexed by k) we create two
   // functions fxx[k] and fxv[k]. Creating data arrays
   // would not be usefull since finally we shoud have
   // the curves for different k shown over a common
   // time axis. The method to achieve this is elegant.
   R mult1=1, mult2=1; // scaling factor
```

```
for (Z k=1;k<=nRuns;++k){
   st=st0;
   st.set_dt_(dt);
   // the following vectors serve an elegant method to create a
   // common graphical representation of curces which differ
   // in the number of defining data points
   vector<R> tVal;
   vector<R> xxVal;
   vector<R> xvrVal;
   vector<R> xviVal;
   CyclicAlarm mark(viewingPeriod);
   for (R t=0; t<=tTot; t+=dt){
      R tp=t*tTotInv;
      cpmprogress("loop",tp,1);
      State2 chi=(st.x().dot()-st.v())*dt;
      R xx=(st.x()|st.x()).real()-xx0;
      C xv=(st.x()|st.v());
      R xvr=xv.real()-xvr0;
      R xvi=xv.imag()-xvi0;
      xx*=mult1;
      xvr*=mult2;
      xvi*=mult2;
      R showFac=100;
      xx*=showFac;
      tVal.push_back(t);
      xxVal.push_back(xx);
      xvrVal.push_back(xvr);
      xviVal.push_back(xvi);
      if (k==1){ // show the first nun and not the others
         if (mark()){
            st.x().mark(gr,0, gamma);
            gr.display();
            cpmwait(tWait);
         }
      }
      st.step_();
   }
   R_Vector t_val(tVal);
   R_Vector xx_val(xxVal);
   R_Vector xvr_val(xvrVal);
   R_Vector xvi_val(xviVal);
   R_Matrix rxx(2);
   R_Matrix rxv(2);
   rxx[1]=t_val;
   rxx[2]=xx_val;
   rxv[1]=t_val;
   rxv[2]=xvr_val;
   fxx[k]=R_Func(rxx);
   fxv[k]=R_Func(rxv);
   fxxxv[k]=R_Func(rxx);
```

```
      fxxxv[k+nRuns]=R_Func(rxv);
      dt*=fac_dt;
      mult1*=1./(fac_dt*fac_dt);
      mult2*=1./(fac_dt*fac_dt);
   }
   Iv tr(timeRange[1]*tTot,timeRange[2]*tTot);
   gr.setX(tr);

   gr.addText("fxx");
   gr.show(fxx,curveRes);
   cpmwait(tWaitWorkScreen_);

   gr.addText("fxv");
   gr.show(fxv,curveRes);
   cpmwait(tWaitWorkScreen_);

   gr.addText("fxxxv");
   gr.show(fxxxv,curveRes);
   cpmwait(tWaitWorkScreen_);
   CPM_MZ
}

//////////////////////////// CpmVQM2::App9 ////////////////////////////////
namespace{
   R pot2Func(R const& x, R const& alpha, R const& p, R const& mu, R\
    const&\
    eps)
   {
      R r=cpmabs(x);
      return alpha*::exp(-mu*r)*::pow(r+eps,-p-1);
   }
}

void CpmVQM2::App9::doTheWork()
// Single particle. Exact solution by spectral representation.
// studying the difference between the exact solution and the
// solution obtained by the direct midpoint integrator
// Made from App8 (2006-09-18)
{
   Z mL=1;
   Word loc("CpmVQM2::App9::doTheWork()");
   CPM_MA
//////////////////////////////////////////////////////////////////////
   Word sec("run control");
//////////////////////////////////////////////////////////////////////
   Z verboseQM,nSteps,viewingPeriod,fileData,fileStates,
      nRuns,curveRes,viewingModality;
   R tInspectPotential=0,tInspectCurves1=0,tInspectCurves2=0,
     tInspectSpectrum=0,tInspectEigenstates=0, fac_dt=1,tWait=0;
   bool createData=true;
```

```
   bool scaleCurves=true;
   bool showLog=true;
   bool showEps=false;
   V<R> timeRange;
/************** experiment, test *************************
   Vo<R> trO(timeRange);
   Vio<R> trIO(trO);
   cpmdebug(trIO);
************************************** works ! ************/
   cpmrh(verboseQM);
   cpmrh(nSteps);
   cpmrh(nRuns);
   cpmrh(viewingPeriod);
   cpmrh(createData);
   cpmrh(fileData);
   cpmrh(fileStates);
   cpmrh(fac_dt);
   cpmrh(tWait);
   cpmrh(tInspectPotential);
   cpmrh(tInspectCurves1);
   cpmrh(tInspectCurves2);
   cpmrh(tInspectSpectrum);
   cpmrh(tInspectEigenstates);
   cpmrh(scaleCurves);
   cpmrh(showLog);
   cpmrh(showEps);
   cpmrh(timeRange);
   cpmrh(curveRes);
   cpmrh(viewingModality);
//////////////////////////////////////////////////////////////////////
   sec="system data";
//////////////////////////////////////////////////////////////////////
   Z d1=60,stepMet=3;
   cpmrh(d1);
   R size1=1.,massRel1,vRel1;
   R dtRel=0, xRel1=0, wRel1=0;
   cpmrh(xRel1);
   cpmrh(wRel1);
   bool useSI=false, noRel=false, relToStabilityLimit=false;
   cpmrh(size1);
   cpmrh(noRel);
   cpmrh(relToStabilityLimit);
   cpmrh(massRel1);
   cpmrh(vRel1);
   R lambdaRel,alpha,mu,eps,p;
   cpmrh(lambdaRel);
   cpmrh(alpha);
   cpmrh(mu);
   cpmrh(eps);
   cpmrh(p);
```

```
   cpmrh(useSI);
   cpmrh(dtRel);
   cpmrh(stepMet);
// start working
   if (createData){
   QMBase::setVerbose(verboseQM);
   Word units=useSI ? "SI":"1";
   OFileStream out1(outFiles[1]);
   Greg date;
   date.now();
   out1()<<"// ";
   date.writeFormatted(out1());
   out1()<<"// output of CpmVQM2::App9::doTheWork()"<<endl;
   out1()<<"units="<<units;
   QMBase::setUnits(units);
   QMBase::setSize(size1);
      // to get the right lattice related data for particle 1
   R m1= noRel ? massRel1 : massRel1*QMBase::mMax(d1);
   QMBase::setMass(m1);
     // initial state:
   bool unitHeg=true; // aha heg = height
   R_Func gpsi1=R_Func::Gauss(xRel1,wRel1,unitHeg);
   State psi0(gpsi1,d1);
   R v1=noRel ? vRel1 : vRel1*0.5*QMBase::vMax(d1);
   out1()<<"v1="<<v1<<endl;
      // 0.5 inseted since without this vRel1=1
      // corresponds to a wave function mainly at
      // Nyquist frequency. Our representation of the Hamilton
      // operator is not very suitable for such strongly
      // varying wave functions
   psi0=psi0.boost(v1);
   R lambda=QMBase::lambdaOsc(d1)*lambdaRel;
   R_Func pot1=R_Func::x2*lambda; // potential of harmonic oscillator
   R_Func pot2=F4<R,R,R,R,R,R>(alpha,p,mu,eps)(pot2Func);
   R_Func pot=pot1+pot2;
   R lambdaAll=lambda==0 && alpha==0 ? 0. : 1.;
   Operator op=Operator::sinParHam(R2_Func(pot),d1,lambdaAll);
      // single particle Hamilton operator from pot
   Observable ob=op.toObservable(0.);
   SpectralObject so=ob.spectralRepresentation();
   so=so.order();
   so.writeSpectrum(out1());
   Frame fr0;
   Frames frs(fr0,1,2);
   Graph gr;
   if (tInspectSpectrum>0){
      so.showSpectrum(gr);
      cpmwait(tInspectSpectrum);
      gr.clr();
   }
```

```
   if (tInspectEigenstates>0){
      so.show(gr);
      cpmwait(tInspectEigenstates);
      gr.clr();
   }
   Observable dot=ob*QMBase::get_ihDiracInv();
   Observable dot2=dot*dot;
   State phi0=dot*psi0;
 //  phi0*=1.1;
      // initial condition is ready
///////////////// reminder: CpmVQM2::App9 ////////////////////////////////
   if (tInspectPotential>0){
      Iv ivx(-0.5,0.5);
      gr.setX(ivx);
      gr.addText("Potential");
      gr.show(pot);
      cpmwait(tInspectPotential);
   }
   gr.clr();
   Graph gr1(frs[1][1]);
   Graph gr2(frs[1][2]);

   R dt0=0;
   if (noRel) dt0=dtRel;
   else{
      if (relToStabilityLimit){
         R eLim=so.spectralRadius()*QMBase::get_hDiracInv();
         dt0=dtRel*2*cpminv(eLim);
      }
      else{
         dt0=dtRel*QMBase::dtLimit(d1);
      }
   }
   out1()<<"dt0="<<dt0<<endl;
   R tTot=nSteps*dt0;
   R tTotInv=1./tTot;
   V< F<R,R> > nDiffV(nRuns);
   V< F<R,R> > psiDiffV(nRuns);
   V<Z> sttCon(nRuns); // state count
   Z kRuns=1;
   R dt=dt0;
   R mult=1;
   if ( fileStates>0 && fac_dt!=0.5){
      // adjustment to get sufficiently many synchronous states
      cpmwarning(loc&": fac_dt set to 1/2");
      fac_dt=0.5;
   }
   R fac_mult0=1./(fac_dt*fac_dt);
   R fac_mult=fac_mult0;
   Word fn("states_of_run_");
```

```
   R dtFinal=dt;
   while (kRuns<=nRuns){
      OFileStream stf(fn&cpm(kRuns));
      cpmstatus("run "&cpm(kRuns)&" of "&cpm(nRuns),3);
      gr1.setAutoScale(1);
      gr2.setAutoScale(1);
      R t=0;
      R tau=0.5*dt;
      State psi=psi0;
      State phi=phi0;
      CyclicAlarm mark(viewingPeriod);
      vector<R> tVal;
      vector<R> nDiffVal;
      vector<R> psiDiffVal;
      Z frames=0;
      while (t<tTot){
         if (mark()){ // show and document the state if wanted
            frames++;
            SpectralObject sot(so);
            sot.expi(-t);
            State psiExc=sot*psi0; // psi exact
            State diff = showEps ? phi-dot*psi : psiExc-psi;
            R nDiff=psi.absVal()-1.;
            R psiDiff=diff.absVal();
            nDiff*=mult;
            psiDiff*=mult;
            tVal.push_back(t);
            nDiffVal.push_back(nDiff);
            psiDiffVal.push_back(psiDiff);
            if (fileData>0){
               out1()<<t<<", "<<nDiff<<", "<<psiDiff<<endl;
            }
            if (fileStates>0){
               R1 tPsi(t);
               tPsi.prnOn(stf());
               psi.prnOn(stf());
            }
///////////////// reminder: CpmVQM2::App9 ////////////////////////////////
            if (viewingModality>0 && kRuns==nRuns){
               // the behavior of the last run can be shown
               if (viewingModality==1){ // three curves
                  // absolute values only in one frame
                  // very informativ
                  V<State> vst("",psiExc,psi,diff);
                  State::show(gr,vst);
               }
               else if (viewingModality==2){
                  Z nTog=20;
                  psi.showAsCurves(gr1);
                  //psiExc.showAsCurves(gr1);
```

```
                  diff.showAsCurves(gr2);
                  if (viewingModality==2){
                     if (frames==nTog){
                        frames=0;
                        gr1.setAutoScale(1);
                        gr2.setAutoScale(1);
                     }
                     else{
                        gr1.setAutoScale(0);
                        gr2.setAutoScale(0);
                     }
                  }
               }
               else ;
               cpmwait(tWait);
            } // if (viewingModality>0 && kRuns==nRuns)
         } // if (mark())
// advancing the state dynamically
         if (stepMet==3){ // direct midpoint method
            State psiMid=psi+phi*tau;
            t+=tau;
            phi+=(dot2*psiMid)*dt;
            psi=psiMid+phi*tau;
            t+=tau;
         }
         else if (stepMet==2){ // inferior method
            State psiMid=psi+phi*tau;
            t+=tau;
            phi=dot*psiMid;
            psi+=phi*dt;
            t+=tau;
         }
         else if (stepMet==1){ // Euler method, only pedagogical
            phi=dot*psi;
            psi+=phi*dt;
            t+=dt;
         }
         else if (stepMet==4){ // pedagogical variation of the
            // direct midpoint method: re-initialization
            phi=dot*psi; // don't use phi from the last step
            State psiMid=psi+phi*tau;
            t+=tau;
            phi+=(dot2*psiMid)*dt;
            psi=psiMid+phi*tau;
            t+=tau;
         }
         else if (stepMet==5){ // pedagogical variation of the
            // direct midpoint method: normalization and
            // re-initialization
            psi.nor_();
```

```
      phi=dot*psi; // don't use phi from the last step
      State psiMid=psi+phi*tau;
      t+=tau;
      phi+=(dot2*psiMid)*dt;
      psi=psiMid+phi*tau;
      t+=tau;
   }
   else if (stepMet==6){ // Askar Cakmak method
      t+=tau;
      psi+=phi*tau;
      phi=dot*psi*2-phi;
      psi+=phi*tau;
      t+=tau;
   }
   else if (stepMet==7){ // midpoint method with smooting
      // no combination of the various conceivable actions of
      // smooting and re-normalization found that looks
      // like working so-la-la beyond the stability limit
      // (2007-06-30)
      // Mere smoothing on ooccasion does not prevent explosion.
      // Smoothing and re-normaizing on occasion freezes the
      // motion.
      State psiMid=psi+phi*tau;
      t+=tau;
      phi+=(dot2*psiMid)*dt;
      psi=psiMid+phi*tau;
      t+=tau;
      R nPsi=psi.absVal()-1;
      R tol=1e-2;
      if (nPsi>tol){
         psi.smt_();
         phi.smt_();
      }
   }
   else cpmerror(loc&" got unvalid value "&cpm(stepMet)&" for\
    stepMet");
   cpmprogress("loop",t*tTotInv,2);
}// time loop
R_Vector t_val(tVal);
R_Vector nDiff_val(nDiffVal);
R_Vector psiDiff_val(psiDiffVal);
R_Matrix nDiffR(2);
R_Matrix psiDiffR(2);
nDiffR[1]=t_val;
psiDiffR[1]=t_val;
nDiffR[2]=nDiff_val;
psiDiffR[2]=psiDiff_val;
nDiffV[kRuns]=R_Func(nDiffR);
psiDiffV[kRuns]=R_Func(psiDiffR);
sttCon[kRuns]=t_val.dim();
```

```
      dt*=fac_dt;
      dtFinal=dt;
      if (scaleCurves) mult*=fac_mult;
      if (fileStates>0) viewingPeriod*=2;
      kRuns++;
   } // run loop
   if (tInspectCurves1>0){
      Word txt=scaleCurves ?
         "scaled difference of norm from 1" : "difference of norm from\
          1"\
         ;
      Iv tr(timeRange[1]*tTot,timeRange[2]*tTot);
      gr.setX(tr);
      gr.addText(txt);
      gr.show(nDiffV,curveRes,rainBow,showLog);
      cpmwait(tInspectCurves1);
   }
   if (tInspectCurves2>0){
      Iv tr(timeRange[1]*tTot,timeRange[2]*tTot);
      gr.setX(tr);
      Word txt0=showEps ? " epsilon" : " error";
      Word txt=scaleCurves ?
         "scaled norm of"&txt0 : "norm of"&txt0;
      gr.addText(txt);
      gr.show(psiDiffV,curveRes,rainBow,showLog);
      Word img("ip_");
      gr.write(img&getRunName()&"_"&cpm(stepMet));
      cpmwait(tInspectCurves2);
   }

///////////////// reminder: CpmVQM2::App9 ////////////////////////////////

   if (fileStates==1){ // reading synchronous states from files
      // and visulizing their difference
      R scf=1;
       V< V<R2> > cur(nRuns-1);
      for (kRuns=1;kRuns<nRuns;kRuns++){
         IFileStream if1(fn&cpm(kRuns));
         IFileStream if2(fn&cpm(kRuns+1));
         cpmstatus("replay: "&cpm(kRuns)&" of "&cpm(nRuns-1),3);
         Z n1=sttCon[kRuns];
         Z n2=sttCon[kRuns+1];
         cpmassert(n2==n1,loc);
         n1=cpminf<Z>(n1,n2);
         R n1Inv=1./n1;
         R dtLimit=0.25*dtFinal; // factor 100 preliminary
         vector<R2> vtd;
         for (Z j=1;j<=n1;++j){
            R1 tPsi1,tPsi2;
            State psi1,psi2;
```

```
         tPsi1.scanFrom(if1());
         psi1.scanFrom(if1());
         tPsi2.scanFrom(if2());
         psi2.scanFrom(if2());
         R tDiff=cpmabs(tPsi1-tPsi2);
         if (tDiff<dtLimit){
            R psi12a=(psi1-psi2).absVal();
            if (scaleCurves) psi12a*=scf;
            if (showLog){
               if (psi12a>0) psi12a=::log10(psi12a);
            }
            R tp1=tPsi1;
            if (showLog && psi12a==0) ;
            else vtd.push_back(R2(tp1,psi12a));
         }
         else{
            cpmwarning(loc&": tDiff="&cpm(tDiff)&" fails to be\
             synchronous");
         }
         cpmprogress("read loop",j*n1Inv,2);
      } // j loop
      cur[kRuns]=V<R2>(vtd);
      scf*=4;
   }// kRuns loop
   gr.clr();
   gr.setAntiAliasLines(1);
   gr.mark(cur);
   Word txt1=showLog ? "log10 of " : "";
   Word txt2=scaleCurves ? "scaled " : "";
   Word txt3="norm of difference between synchronous states";
   gr.addText(txt1&txt2&txt3);
   gr.display();
   cpmwait(tWaitWorkScreen_);
} // if (fileStates>0)
} // createData
else { // thus !createData
   Graph gr;
   gr.clr();
   R aFac=0.25;
   V< V<R2> > cur(nRuns-1);
   Word fn("states_of_run_");
   cpmdebug(nRuns);
   for (Z k=1;k<nRuns;k++){
      R scf=pow(aFac,-nRuns)/(pow(aFac,-k)-1);
      IFileStream if1(fn&cpm(nRuns));
      IFileStream if2(fn&cpm(nRuns-k));
      cpmstatus("reading: "&cpm(k)&" of "&cpm(nRuns-1),3);
      R dtLimit=1e-9;
      vector<R2> vtd;
      Z frameCounter=0;
```

```
        bool fail=false;
        for(;;) { // reading from file
           R1 tPsi1,tPsi2;
           State psi1,psi2;
           fail=!tPsi1.scanFrom(if1());
           if (fail) break;
           fail=!psi1.scanFrom(if1());
           if (fail) break;
           fail=!tPsi2.scanFrom(if2());
           if (fail) break;
           fail=!psi2.scanFrom(if2());
           if (fail) break;
           frameCounter++;
           R tDiff=cpmabs(tPsi1-tPsi2);
        //   cpmassert(tDiff<dtLimit,loc);
           if (tDiff<dtLimit){
              State psi12=psi1-psi2;
              R psi12a=psi12.absVal();
              psi12.showAsCurves(gr);
              if (scaleCurves) psi12a*=scf;
              if (showLog){
                 if (psi12a>0) psi12a=::log10(psi12a);
              }
              R tp1=tPsi1;
              if (showLog && psi12a==0) ;
              else vtd.push_back(R2(tp1,psi12a));
           }
           else{
              cpmwarning(loc&": tDiff="&cpm(tDiff)&" fails to be\
               synchronous");
           }
        } // for (;;)
        Z nFrames=(Z)vtd.size();
        cpmdebug(nFrames);
        cur[k]=V<R2>(vtd);
     }// kRuns loop
  //    gr.clr();
     gr.mark(cur);
     Word txt1=showLog ? "log10 of " : "";
     Word txt2=scaleCurves ? "scaled " : "";
     Word txt3="norm of difference between synchronous states";
     gr.addText(txt1&txt2&txt3);
     gr.display();
     cpmwait(tWaitWorkScreen_);
   }
   CPM_MZ
}

//////////////////////////////////////////////////////////////////////////
```

```
//////////////////////////// CpmVQM2::App10 ////////////////////////////////

namespace{
   R log10X(R x, R ex)
   {
      R y=cpmabs(x);
      if (y==0.) return -ex;
      else return log10(y);
   }
}

void CpmVQM2::App10::doTheWork()
// 1-dimensional case: Euler formla exp(-i omega t)
// studying the difference between the exact solution and the
// solution obtained by the direct midpoint integrator
// Made from App9 (2006-10-13)
{
   Z mL=1;
   Word loc("CpmVQM2::App10::doTheWork()");
   CPM_MA
//////////////////////////////////////////////////////////////////////
   Word sec("run control");
//////////////////////////////////////////////////////////////////////
   Z verboseQM,nSteps,viewingPeriod,fileData,
      nRuns,curveRes,viewingModality;
   R tInspectPotential=0,tInspectCurves1=0,tInspectCurves2=0,
      fac_dt=1,tWait=0,logEx;
   bool createData=true;
   bool scaleCurves=true;
   bool showLog=true;
   bool showEps=false;
   V<R> timeRange;
   cpmrh(verboseQM);
   cpmrh(nSteps);
   cpmrh(nRuns);
   cpmrh(viewingPeriod);
   cpmrh(createData);
   cpmrh(fileData);
   cpmrh(fac_dt);
   cpmrh(tWait);
   cpmrh(tInspectCurves1);
   cpmrh(tInspectCurves2);
   cpmrh(scaleCurves);
   cpmrh(showLog);
   cpmrh(logEx);
   cpmrh(showEps);
   cpmrh(timeRange);
   cpmrh(curveRes);
   cpmrh(viewingModality);
//////////////////////////////////////////////////////////////////////
```

```
   sec="system data";
//////////////////////////////////////////////////////////////////////
   Z d1=1,stepMet=3;
   R omega=1,dt0;
   cpmrh(omega);
   cpmrh(dt0);
   cpmrh(stepMet);
// start working
   QMBase::setVerbose(verboseQM);
   OFileStream out1(outFiles[1]);
   Greg date;
   date.now();
   out1()<<"// ";
   date.writeFormatted(out1());
   out1()<<"// output of CpmVQM2::App10::doTheWork()"<<endl;
   QMBase::setUnits("1");
      // to get the right lattice related data for particle 1
   State psi0(d1,0);
   Observable ob(1.,d1);
   SpectralObject so=ob.spectralRepresentation();
   Observable dot=ob*QMBase::get_ihDiracInv();
   Observable dot2=dot*dot;
   State phi0=dot*psi0;
      // initial condition is ready
/////////////// reminder: CpmVQM2::App10 ////////////////////////////////
   Frame fr0;
   Frames frs(fr0,1,2);
   Graph gr;
   gr.clr();
   Graph gr1(frs[1][1]);
   Graph gr2(frs[1][2]);
   out1()<<"dt0="<<dt0<<endl;
   R tTot=nSteps*dt0;
   R tTotInv=1./tTot;
   V< F<R,R> > nDiffV(nRuns);
   V< F<R,R> > psiDiffV(nRuns);
   V<Z> sttCon(nRuns); // state count
   Z kRuns=1;
   R dt=dt0;
   R mult=1;
   R fac_mult0=1./(fac_dt*fac_dt);
   R fac_mult=fac_mult0;
   Word fn("states_of_run_");
   R dtFinal=dt;
   while (kRuns<=nRuns){
      OFileStream stf(fn&cpm(kRuns));
      cpmstatus("run "&cpm(kRuns)&" of "&cpm(nRuns),3);
      gr1.setAutoScale(1);
      gr2.setAutoScale(1);
      R t=0;
```

```
      R tau=0.5*dt;
      State psi=psi0;
      State phi=phi0;
      CyclicAlarm mark(viewingPeriod);
      vector<R> tVal;
      vector<R> nDiffVal;
      vector<R> psiDiffVal;
      Z frames=0;
      while (t<tTot){
         if (mark()){ // show and document the state if wanted
            frames++;
            SpectralObject sot(so);
            sot.expi(-t);
            State psiExc=sot*psi0; // psi exact
            State diff = showEps ? phi-dot*psi : psiExc-psi;
            //R nDiff=psi.absVal()-1.;
            //R nDiff=psi[0].imag();
            R nDiff=psi[0].real();
            R psiDiff=diff.absVal();
            nDiff*=mult;
            psiDiff*=mult;
            if (showLog){
               nDiff=log10X(nDiff,logEx);
               psiDiff=log10X(psiDiff,logEx);
            }
            tVal.push_back(t);
            nDiffVal.push_back(nDiff);
            psiDiffVal.push_back(psiDiff);
            if (fileData>0){
               out1()<<t<<", "<<nDiff<<", "<<psiDiff<<endl;
            }
/////////////// reminder: CpmVQM2::App10 ////////////////////////////////
            if (viewingModality>0 && kRuns==nRuns){
               if (viewingModality==1){ // three curves
                  // absolute values only in one frame
                  // very informativ
                  V<State> vst("",psiExc,psi,diff);
                  gr.setAutoScale(1); //experiment
                  State::show(gr,vst);
               }
               else if (viewingModality==2){
                  Z nTog=10;
               //psi.showAsCurves(gr1);
                  psiExc.showAsCurves(gr1);
                  diff.showAsCurves(gr2);
                  if (viewingModality==2){
                     if (frames==nTog){
                        frames=0;
                        gr1.setAutoScale(1);
                        gr2.setAutoScale(1);
```

```
                    }
                    else{
                       gr1.setAutoScale(0);
                       gr2.setAutoScale(0);
                    }
                 }
              }
              else ;
              cpmwait(tWait);
           } // if (viewingModality>0 && kRuns==nRuns)
        } // if (mark())
// advancing the state dynamically
        if (stepMet==3){ // direct midpoint method
           State psiMid=psi+phi*tau;
           t+=tau;
           phi+=(dot2*psiMid)*dt;
           psi=psiMid+phi*tau;
           t+=tau;
        }
        else if (stepMet==2){ // inferior method
           State psiMid=psi+phi*tau;
           t+=tau;
           phi=dot*psiMid;
           psi+=phi*dt;
           t+=tau;
        }
        else if (stepMet==1){ // Euler method, only pedagogical
           phi=dot*psi;
           psi+=phi*dt;
           t+=dt;
        }
        else if (stepMet==4){ // pedagogical variation of the
           // direct midpoint method: re-initialization
           phi=dot*psi; // don't use phi from the last step
           State psiMid=psi+phi*tau;
           t+=tau;
           phi+=(dot2*psiMid)*dt;
           psi=psiMid+phi*tau;
           t+=tau;
        }
        else if (stepMet==5){ // pedagogical variation of the
           // direct midpoint method: normalization and
           // re-initialization
           psi.nor_();
           phi=dot*psi; // don't use phi from the last step
           State psiMid=psi+phi*tau;
           t+=tau;
           phi+=(dot2*psiMid)*dt;
           psi=psiMid+phi*tau;
           t+=tau;
```

```
         }
         else cpmerror(loc&" got unvalid value "&cpm(stepMet)&" for\
          stepMet");
         cpmprogress("loop",t*tTotInv,2);
      }// time loop
      R_Vector t_val(tVal);
      R_Vector nDiff_val(nDiffVal);
      R_Vector psiDiff_val(psiDiffVal);
      R_Matrix nDiffR(2);
      R_Matrix psiDiffR(2);
      nDiffR[1]=t_val;
      psiDiffR[1]=t_val;
      nDiffR[2]=nDiff_val;
      psiDiffR[2]=psiDiff_val;
      nDiffV[kRuns]=R_Func(nDiffR);
      psiDiffV[kRuns]=R_Func(psiDiffR);
      sttCon[kRuns]=t_val.dim();
      dt*=fac_dt;
      dtFinal=dt;
      if (scaleCurves) mult*=fac_mult;
      kRuns++;
   } // run loop
   Word txtF=showLog ? "log10 of " : " ";
   if (tInspectCurves1>0){
      Word txt=txtF;
      if (scaleCurves)
         txt&="scaled difference of norm from 1";
      else txt&="difference of norm from 1";
      Iv tr(timeRange[1]*tTot,timeRange[2]*tTot);
      gr.setX(tr);
      gr.addText(txt);
      gr.show(nDiffV,curveRes);
      cpmwait(tInspectCurves1);
   }
   if (tInspectCurves2>0){
      Iv tr(timeRange[1]*tTot,timeRange[2]*tTot);
      gr.setX(tr);
      Word txt=txtF;
      Word txt0=showEps ? " epsilon" : " error";
      if (scaleCurves) txt&="scaled norm of"&txt0;
      else txt&="norm of"&txt0;
      gr.addText(txt);
      gr.show(psiDiffV,curveRes);
      cpmwait(tInspectCurves2);
   }
   CPM_MZ
}

///////////////// void CpmVQM2::function(void) //////////////
```

```
// functions of signature void *f(void) for final association with a menu
//
// selection


void CpmVQM2::app1()
{
   cpmmessage("app1() started");
   App1 a("app1");
   a.run();
}

void CpmVQM2::app2()
{
   cpmmessage("app2() started");
   App2 a("app2");
   a.run();
}

void CpmVQM2::app3()
{
   cpmmessage("app3() started");
   App3 a("app3");
   a.run();
}

void CpmVQM2::app4()
{
   cpmmessage("app4() started");
   App4 a("app4");
   a.run();
}

void CpmVQM2::app4b()
{
   cpmmessage("app4b() started");
   App4b a("app4b");
   a.run();
}

void CpmVQM2::app5()
{
   cpmmessage("app5() started");
   App5 a("app5");
   a.run();
}

void CpmVQM2::app6()
{
   cpmmessage("app6() started");
```

```
   App6 a("app6");
   a.run();
}

void CpmVQM2::app7()
{
   cpmmessage("app7() started");
   App7 a("app7");
   a.run();
}

void CpmVQM2::app8()
{
   cpmmessage("app8() started");
   App8 a("app8");
   a.run();
}

void CpmVQM2::app9()
{
   cpmmessage("app9() started");
   App9 a("app9");
   a.run();
}

void CpmVQM2::app10()
{
   cpmmessage("app10() started");
   App10 a("app10");
   a.run();
}
```

242 vqm2app2.cpp

```
//? vqm2app2.cpp
//? Status of work 2008-10-25.
//?
/*********************************************************************
   vqm2app2.cpp
*********************************************************************/
#include <vqm2app2.h>
#include <vqm2.h>

#include <cpmframes.h>
#include <cpmgreg.h>
#include <cpmrfuncrec.h>
#include <cpmoperator.h>
#include <cpmtrj.h>
#include <cpmfieldoperator.h>
#include <cpmstate2.h>
#include <cpmfft.h>
#include <cpmmaxw1d.h>
#include <cpmqmtemplates.h>

using namespace CpmSystem;
using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmArrays;
using namespace CpmFunctions;
using namespace CpmGraphics;
using namespace CpmImaging;
using namespace CpmQuantumMechanics;

using CpmAlgorithms::CyclicAlarm;
using CpmTime::Greg;
using CpmWaves::TEM;
using CpmWaves::TEM2;

//////////////////////// App11 ////////////////////////////////

void CpmVQM2::App11::doTheWork()
// 1-dimensional case: Euler formla exp(-i omega t)
// studying the difference between the exact solution and the
// solution obtained by the direct midpoint integrator
// visualizing splining
// Made from App10 (2006-10-13)
{
   Z mL=1;
   Word loc("CpmVQM2::App11::doTheWork()");
   CPM_MA
//////////////////////////////////////////////////////////////////////
```

```
   Word sec("run control");
//////////////////////////////////////////////////////////////////////
   Z nSteps,nSubSteps;
   R tInspectCurves0,tInspectCurves1,tInspectCurves2,tInspectCurves3;
   bool antiAlias=false;
   cpmrh(nSteps);
   cpmrh(nSubSteps);
   cpmrh(tInspectCurves0);
   cpmrh(tInspectCurves1);
   cpmrh(tInspectCurves2);
   cpmrh(tInspectCurves3);
   cpmrh(antiAlias);
 ////////////////////////////////////////////////////////////////////
   sec="system data";
//////////////////////////////////////////////////////////////////////
   R omega=1,dt,phase0,chi_x,chi_y;
   cpmrh(omega);
   cpmrh(chi_x);
   cpmrh(chi_y);
   cpmrh(dt);
   cpmrh(phase0);
 // start working
   OFileStream out1(outFiles[1]);
   Greg date;
   date.now();
   out1()<<"// ";
   date.writeFormatted(out1());
   out1()<<"// output of CpmVQM2::App11::doTheWork()"<<endl;
   C psi0=C::expi(phase0);
   C psi=psi0;
   C dot=-C::I*omega;
   C dot2=dot*dot;
   C phi=dot*psi;
   phi+=C(chi_x,chi_y);
   R tau=0.5*dt;
   C psiMid=psi+phi*tau;
   C alpha=dot*dot*psiMid;
      // initial condition is ready
   Graph gr(0.05,0.05,0.9,0.9);
   gr.clr();
   gr.setAntiAliasLines(antiAlias);

   R t=0;
   R tSteps=0;

   R dtSub=dt/nSubSteps;
   vector<R> tVal;
   vector<R> rpsiEx;
   vector<R> ipsiEx;
   vector<R> rpsiRep;
```

```
vector<R> ipsiRep;
vector<R> rspEx;
vector<R> ispEx;
vector<R> rspRep;
vector<R> ispRep;
vector<R> da;
R tTot=nSteps*dt;
while (t<tTot){
   R phase=-omega*t;
   C psiEx=C::expi(phase)*psi0;
   C phiEx=dot*psiEx;
   R aPsiEx=psiEx.absVal();
   R rPsiEx=psiEx.real();
   R iPsiEx=psiEx.imag();
   C spEx=phiEx.con()*psiEx;
   R rSpEx=spEx.real();
   R iSpEx=spEx.imag();
   rpsiEx.push_back(rPsiEx);
   ipsiEx.push_back(iPsiEx);
   rspEx.push_back(rSpEx);
   ispEx.push_back(iSpEx);

   R h=t-tSteps;
   if (h>=dt){
      psiMid=psi+phi*tau;
    //  alpha=dot2*psiMid; // alpha has valid value yet
      phi+=alpha*dt;
      psi=psiMid+phi*tau;
      tSteps+=dt;
      h=t-tSteps;
      psiMid=psi+phi*tau;
      alpha=dot2*psiMid; // this is alpha to be
      // used in the next statements
   }
   C psiRep=psi+phi*h+alpha*(h*h*0.5);
   C phiRep=phi+alpha*h;
   C spRep=phiRep.con()*psiRep;
   R aPsiRep=psiRep.absVal();
   R rPsiRep=psiRep.real();
   R iPsiRep=psiRep.imag();
   rpsiRep.push_back(rPsiRep);
   ipsiRep.push_back(iPsiRep);
   R rSpRep=spRep.real();
   R iSpRep=spRep.imag();
   rspRep.push_back(rSpRep);
   ispRep.push_back(iSpRep);
   da.push_back(aPsiRep-aPsiEx);
   tVal.push_back(t);
   t+=dtSub;
}// time loop
```

```
R_Vector t_val(tVal);
R_Vector rpsiEx_(rpsiEx);
R_Vector ipsiEx_(ipsiEx);
R_Vector rpsiRep_(rpsiRep);
R_Vector ipsiRep_(ipsiRep);
R_Vector rspEx_(rspEx);
R_Vector ispEx_(ispEx);
R_Vector rspRep_(rspRep);
R_Vector ispRep_(ispRep);
R_Vector da_(da);

if (tInspectCurves0>0){
   R_Matrix show(3);
   show[1]=t_val;
   show[2]=rpsiEx_;
   show[3]=rpsiRep_;
   gr.addText("psiExact Real, psiRep Real");
   gr.show(show);
   cpmwait(tInspectCurves0);
}
if (tInspectCurves1>0){
   R_Matrix show(5);
   show[1]=t_val;
   show[2]=rpsiEx_;
   show[3]=ipsiEx_;
   show[4]=rpsiRep_;
   show[5]=ipsiRep_;
   gr.addText("psiExact Real and Img, psiRep Real and Img");
   gr.show(show);
   cpmwait(tInspectCurves1);
}
if (tInspectCurves2>0){
   R_Matrix show(5);
   show[1]=t_val;
   show[2]=rspEx_;
   show[3]=ispEx_;
   show[4]=rspRep_;
   show[5]=ispRep_;
   gr.addText("scalar product phi times psi, Real Img Exact Rep");
   gr.show(show);
   cpmwait(tInspectCurves2);
}
if (tInspectCurves3>0){
   R_Matrix show(2);
   show[1]=t_val;
   show[2]=da_;
   gr.addText("norm Rep - norm Exact");
   gr.show(show);
   cpmwait(tInspectCurves3);
}
```

```
   CPM_MZ
}

//////////////////////// App12 ////////////////////////////////

void CpmVQM2::App12::doTheWork()
// 1-dimensional case: Euler formla exp(-i omega t)
// studying the difference between the exact solution and the
// solution obtained by the direct midpoint integrator
// Made from App11 (2006-11-15)
{
   Z mL=1;
   Word loc("CpmVQM2::App12::doTheWork()");
   CPM_MA
//////////////////////////////////////////////////////////////////////
   Word sec("run control");
//////////////////////////////////////////////////////////////////////
   R nPeriods;
   R tInspectCurves0,tInspectCurves1,tInspectCurves2,tInspectCurves3;
   bool antiAlias=false;
   cpmrh(nPeriods);
   cpmrh(tInspectCurves0);
   cpmrh(tInspectCurves1);
   cpmrh(tInspectCurves2);
   cpmrh(tInspectCurves3);
   cpmrh(antiAlias);
 ////////////////////////////////////////////////////////////////////
   sec="system data";
//////////////////////////////////////////////////////////////////////
   R omega=1,dt,phase0;
   cpmrh(omega);
   cpmrh(dt);
   cpmrh(phase0);
 // start working
   OFileStream out1(outFiles[1]);
   Greg date;
   date.now();
   out1()<<"// ";
   date.writeFormatted(out1());
   out1()<<"// output of CpmVQM2::App11::doTheWork()"<<endl;
   C psi0=C::expi(phase0);
   C dot=-C::I*omega;
   C dot2=dot*dot;
   C phi0=dot*psi0;

   C psiEx=psi0;
   C phiEx=phi0;

   C psiDM=psi0;
   C phiDM=phi0;
```

```
Graph gr(0.05,0.05,0.9,0.9);
gr.clr();
gr.setAntiAliasLines(antiAlias);

R t=0;
R tau=0.5*dt;
R tSteps=0;

vector<R> tVal;
vector<C> vPsiEx;
   // Ex: exact
vector<C> vPsiDM;
   // DM: direct midpoint
vector<C> vPhiEx;
vector<C> vPhiDM;
vector<R> vnu;

R tTot=nPeriods*2*CpmRoot::Pi/omega;
while (t<tTot){
   tVal.push_back(t);
   vPsiEx.push_back(psiEx);
   vPhiEx.push_back(phiEx);
   vPsiDM.push_back(psiDM);
   vPhiDM.push_back(phiDM);
   R vu=psiDM.absSqr()-1;
   vnu.push_back(vu);
   t+=dt;
   R phase=-omega*t;
   psiEx=C::expi(phase)*psi0;
   phiEx=dot*psiEx;

   psiDM+=phiDM*tau;
   C alpha=dot2*psiDM;
   phiDM+=alpha*dt;
   psiDM+=phiDM*tau;

}// time loop
R_Vector t_val(tVal);
R_Vector v_nu(vnu);
V<C> v_psiDM(vPsiDM);
V<C> v_psiEx(vPsiEx);
V<C> v_phiDM(vPhiDM);
V<C> v_phiEx(vPhiEx);
Z i,n=t_val.dim();
R_Vector vrpsiDM(n);
R_Vector vrpsiEx(n);
R_Vector vrphiDM(n);
R_Vector vrphiEx(n);
for (i=1;i<=n;++i){
```

```
      vrpsiDM[i]=v_psiDM[i][1];
      vrpsiEx[i]=v_psiEx[i][1];
      vrphiDM[i]=v_phiDM[i][1];
      vrphiEx[i]=v_phiEx[i][1];
   }
   /*R_Vector rpsiEx_(rpsiEx);
   R_Vector ipsiEx_(ipsiEx);
   R_Vector rpsiRep_(rpsiRep);
   R_Vector ipsiRep_(ipsiRep);
   R_Vector rspEx_(rspEx);
   R_Vector ispEx_(ispEx);
   R_Vector rspRep_(rspRep);
   R_Vector ispRep_(ispRep);
   R_Vector da_(da);*/

   if (tInspectCurves0>0){
      R_Matrix show(2);
      show[1]=t_val;
      show[2]=v_nu;
    //  show[3]=rpsiRep_;
      gr.addText("nu");
      gr.show(show);
      cpmwait(tInspectCurves0);
   }
   if (tInspectCurves1>0){
      R_Matrix show(3);
      show[1]=t_val;
      show[2]=vrpsiEx;
      show[3]=vrpsiDM;
      gr.addText("psiExact Real, psiDM Real");
      gr.show(show);
      cpmwait(tInspectCurves1);
   }
   if (tInspectCurves2>0){
      R_Matrix show(3);
      show[1]=t_val;
      show[2]=vrphiEx;
      show[3]=vrphiDM;
      gr.addText("phiExact Real, phiDM Real");
      gr.show(show);
      cpmwait(tInspectCurves2);
   }
/***************
   if (tInspectCurves2>0){
      R_Matrix show(5);
      show[1]=t_val;
      show[2]=rspEx_;
      show[3]=ispEx_;
      show[4]=rspRep_;
      show[5]=ispRep_;
```

```
      gr.addText("scalar product phi times psi, Real Img Exact Rep");
      gr.show(show);
      cpmwait(tInspectCurves2);
   }
   if (tInspectCurves3>0){
      R_Matrix show(2);
      show[1]=t_val;
      show[2]=da_;
      gr.addText("norm Rep - norm Exact");
      gr.show(show);
      cpmwait(tInspectCurves3);
   }
******************************/
   CPM_MZ
}


//////////////////////// App13 ////////////////////////////

C fp(R const& x){
   R y=6*x;
   return C(sin(y),cos(y))*exp(-y*y)*exp(-1./(0.25-x*x));
}

/*******************

void CpmVQM2::App13::doTheWork()
{
   Z mL=1;
   Word loc("CpmVQM2::App13::doTheWork()");
   CPM_MA
   static R fac=-QMBase::get_hDiracInv();
   Z d=128;
   Z i,n=4;
   R tWait=60;
   R tWaitFrame=0.;
  // F<R,C> f_psi(fp);
  // State psi(f_psi,d);
   Observable ob("eKinF",d);
   SpectralObject so=ob.spectralRepresentation();
   Z im=115;
   State psi_1=so[im];
   State psi_2=so[im+3]*C(0.5);
   State psi_3=so[im-5]*C(0,0.8);

   State psi=psi_1+psi_2+psi_3;
   psi.normalize_();
 //  State stF=so.represent(psi);
   R dt=1e-3;
   so.expi(dt*fac);
```

```
   Graph g;
   Z nf=256;
   psi.showAsCurves(g);
   g.setAutoScale(0);
   State psi_t=psi;
   V<C> v(nf);
   R nfInv=1./(nf-1);
   for (i=1;i<=nf;++i){
      cpmprogress("i-loop",(i-1)*nfInv);
      psi_t=so*psi_t;
      v[i]=(psi|psi_t);
      psi_t.showAsCurves(g);
      cpmwait(tWaitFrame);
   }
   V<C> e=CpmFourier::fft(v,-1);
   State st("",e);
 //  State st("",v);
   g.setAutoScale(1);
   st.showAsCurves(g);
   cpmwait(tWait);
  // stF.showAsCurves(g);
 //  cpmwait(tWait);
   CPM_MZ
}
*********************************/

void CpmVQM2::App13::doTheWork()
{
   Z mL=1;
   Word loc("CpmVQM2::App13::doTheWork()");
   CPM_MA
   static R fac=-QMBase::get_hDiracInv();
   Z d=128;
   Z i,n=4;
   R tWait=60;
   R tWaitFrame=0.;
  // F<R,C> f_psi(fp);
  // State psi(f_psi,d);
   Observable ob("eKinF",d);
   SpectralObject so=ob.spectralRepresentation();
   Z im=115;
   State psi_1=so[im];
   State psi_2=so[im+3]*C(0.5);
   State psi_3=so[im-20]*C(0,0.8);
   State psi_4=so[im-30];

   State psi=psi_1+psi_2+psi_3+psi_4;
   psi.normalize_();

   R dt=1e-1;
```

```
   R dx=dt*fac;
   so.expi(dx);
   Graph g;
   Z nf=500;
   psi.showAsCurves(g);
   g.setAutoScale(0);
   State psi_t=psi;
   R nfInv=1./(nf-1);
   R beta=0;
   R alpha=(psi_t|ob*psi_t).real();
   R_Vector va(nf+1);
   Z j0=200;
   va[1]=alpha;
   for (i=1;i<=nf;++i){
      cpmprogress("i-loop",(i-1)*nfInv);
      State psi_d=so*psi_t;
      State psi_next=psi_d*C::expi(-dx*alpha);
      R r=CpmRoot::randomR(j0++);
      Z sig=r<0.5 ? 1 : -1;
      beta=(psi_next-psi_t).absSqr()*0.25*sig;
      R delta=::sqrt(cpmabs(psi_d.absSqr()-alpha*alpha));
      psi_t=psi_next;
      alpha+=beta*delta;
     // alpha=(psi_t|ob*psi_t).real();
      va[i+1]=alpha;
      psi_t.showAsCurves(g);
     // cpmwait(tWaitFrame);
   }
   psi_t.showAsCurves(g);
   cpmwait(10);
   g.setAutoScale(1);
   g.show(va);
   cpmwait(tWait);
   CPM_MZ
}


//////////////////////// App14 ////////////////////////////////
// Maxwell 1D system, inspired by Hans De Raedt

namespace{
   R gauss14(R const& x, R const& x0, R const& sigma)
   {
      R a=(x-x0)/sigma;
      return ::exp(-a*a);
   }
}

//#undef USE_TEM2
#define USE_TEM2
```

```
void CpmVQM2::App14::doTheWork()
{
#if defined(USE_TEM2)
   typedef TEM2 AppSys;
   Word app("TEM2");
#else
   typedef TEM AppSys;
   Word app("TEM");
#endif
   Z mL=2;
   Word loc("CpmVQM2::App14::doTheWork()");
   CPM_MA
// reading control data from a text file
   Word sec("system definition");
   Z n, nTrigE=1, nTrigH=1;
   Word typeOfIniState="gauss";
   R L,x0,sigma,xJ,tiJ,tfJ,omegaJ;
   Z nSmoothJ;
   cpmrh(n);
   cpmrh(L);
   cpmrh(x0);
   cpmrh(xJ);
   cpmrh(tiJ);
   cpmrh(tfJ);
   cpmrh(omegaJ);
   cpmrh(nSmoothJ);
   cpmrh(sigma);
   cpmrh(nTrigE);
   cpmrh(nTrigH);
   cpmrh(typeOfIniState);
//////////////////////////////////////
   sec="run control";
   Z nSteps=10,viewingPeriod=1,nRuns=1;
   R dt,tWait=0.5;
   R xViewRelLower=0, xViewRelUpper=1, yView=-1;
   B autoScale=B(true);
   cpmrh(nSteps);
   cpmrh(dt);
   cpmrh(nRuns);
   cpmrh(viewingPeriod);
   cpmrh(tWait);
   cpmrh(autoScale);
   cpmrh(xViewRelLower);
   cpmrh(xViewRelUpper);
   cpmrh(yView);
// Now all data known and we build the initial state
   R_Func fE,fH;
   R sca=1./(dt*dt);
   AppSys tem;
```

```
   if (typeOfIniState=="gauss"){
      // proposed by Hans De Raedt
      R Linv=1./L;
      x0*=Linv;
      sigma*=Linv;
      fE=F2<R,R,R,R>(x0,sigma)(gauss14);
      tem=AppSys(n,L,fE,fH);
   }
   else if (typeOfIniState=="trigonometric"){
      // simple smooth functions, satisfying the boundary
      // conditions. Not proposed by Hans De Raedt
      R sE=1./(Pi*nTrigE), sH=1./(Pi*nTrigH);
      fE=R_Func(sin);
      fH=R_Func(cos);
      fE.stretch_(sE);
      fH.stretch_(sH);
      tem=AppSys(n,L,fE,fH);
   }
   else if (typeOfIniState=="stochastic")
      // proposed by Hans De Raedt
      tem=AppSys(n,L,Iv(-1,1));
#if defined(USE_TEM2)
   else if (typeOfIniState=="current"){
      // proposed by Hans De Raedt
      tem=TEM2(n, L, xJ, Iv(tiJ,tfJ),omegaJ,nSmoothJ);
   }
#endif
   else
      cpmerror("bad selection for typeOfIniState");
   AppSys tem0=tem;
   R en0=tem.energy();
   R dis0=tem.dissipation();
   R nStepsInv=1./nSteps;
   R t0=0,t;
   Z i,j;
// creating resources for storing results from nRuns
// consecutive program runs. These runs cover a given span of
// time with more and more time steps. Actually, from run
// to run the number of time steps doubles.
   V< R_Vector > vRes(nRuns);
   R_Vector vdt(nRuns);
   V< V<R2> > vE(nRuns);
   V< V<R2> > vH(nRuns);
   V< V<R2> > nu_scaled(nRuns);
   V< V<R2> > eps_scaled(nRuns);
   Graph g;
   Z order=2;
   Z facSteps=2;
   R facStepsInv=1./facSteps;
   R facSca=Z1(facSteps).pow(order);
```

```
   CyclicAlarm show(viewingPeriod);
   for (j=1;j<=nRuns;++j){ // loop of runs
      tem=tem0;
      if (j==1){
         g.setAutoScale(1);
         tem.showAsCurves(g);
         g.write(Word("initialFrame_")&app);
      }
      t=t0; // each run starts with the same field and the same model
      // time
      cpmvalue("run index ",j,3);
      nu_scaled[j]=V<R2>(nSteps);
      eps_scaled[j]=V<R2>(nSteps);
      vdt[j]=dt;
      for (i=1;i<=nSteps;++i){ // loop of steps
// performing a time step
         tem.step_(dt);
         t+=dt;
      //   R nu=(tem.energy()-en0)*sca;
      //   R nu=(tem.energy()-en0)/en0;
         R nu=tem.energy();
         R eps=(tem.dissipation()-dis0)*sca;
         //  R nu=tem.energy();
         // R eps=tem.dissipation();
// storing data concerning the state reached by the time step
         nu_scaled[j][i]=R2(t,nu);
         eps_scaled[j][i]=R2(t,eps);
         R L=tem.getL();
         R xL=L*xViewRelLower;
         R xU=L*xViewRelUpper;
         Iv yRange(-yView,yView);
         Iv xRange(xL,xU);
         if (j==1){
            if (show()||i==nSteps){ // visualizing the evolution
               if (autoScale) g.setAutoScale(1);
               else g.setXY(xRange,yRange);
               tem.showAsCurves(g);
            }
            if (i==nSteps){
               g.write(Word("finalFrame_")&app);
               cpmwait(tWait);
            }
         }
         cpmprogress("loop",i*nStepsInv);
      } // loop of steps
// storing results from run j
      vRes[j]=tem.getXr();
      V< V<R2> > vvr=tem.getEH();
      vE[j]=vvr[1];
      vH[j]=vvr[2];
```

```
// setting the parameters for run j+1
      nSteps*=facSteps;
      nStepsInv*=facStepsInv;
      dt*=facStepsInv;
      sca*=facSca;
   } // loop of runs
// putting the results from the various runs together

   V<R_Vector> diff(nRuns);

   for (j=1;j<=nRuns;++j){
      diff[j]=vRes[j]-vRes[nRuns];
   }

   R_Matrix diffs(nRuns);

   if (nRuns>1){
      Z dimx=tem.getxr().dim();
      Z dimy=diff[1].dim();
      diffs[1] = dimx==dimy ? tem.getxr() : tem.getxr()&tem.getxr();
      R fac=1;
      for (j=1;j<nRuns;++j){
         diffs[j+1]=diff[j]*fac;
         fac*=facSca;
      }
   }

// showing documentation on screen

   g.setAutoScale(1);
   g.mark(nu_scaled);
   g.addText("nu_scaled");
   g.display();
   cpmwait(tWait);

   g.setAutoScale(1);
   g.mark(eps_scaled);
   g.addText("eps_scaled");
  // g.display();
  // cpmwait(tWait);

   g.clear();
   g.setAutoScale(1);
   g.mark(vE);
   g.addText("final E");
  // g.display();
 //  cpmwait(tWait);

   g.clear();
   g.setAutoScale(1);
```

```
   g.mark(vH);
   g.addText("final H");
 //  g.display();
  // cpmwait(tWait);

   if (nRuns>1){
      g.clear();
      g.setAutoScale(1);
      g.addText("scaled differences between runs");
      g.show(diffs);
      cpmwait(2*nRuns*tWait);
   }

// filing documentation data

   R normFinal=vRes[nRuns].absVal();
   R normInv=1./normFinal;
   R_Vector res(nRuns);
   for (i=1;i<=nRuns;i++){
      res[i]=diff[i].absVal()*normInv;
   }

   OFileStream of1(outFiles[1]);
   of1()<<"Type of initial state is "<<typeOfIniState<<endl;
   of1()<<"Normalized distance dj from final state of run j to final\
    state\
    of final run:"<<endl;
   for (j=1;j<=nRuns;j++){
      of1()<<"j="<<j<<" dt="<<vdt[j]<<" dj="<<res[j]<<endl;
   }

   OFileStream of2(outFiles[2]);
   Z nr=nu_scaled[1].dim();
   for (i=1;i<=nr;++i){
      of2()<<nu_scaled[1][i][1]<<" "<<nu_scaled[1][i][2]<<"\
       "<<eps_scaled[1][i][2]<<endl;
   }
   CPM_MZ
}

//////////////////////// App15 ////////////////////////////////

void CpmVQM2::App15::doTheWork()
// Derived from App7, now aiming at testing the Chebyshev-method
{
   Z mL=2;
   Word loc("CpmVQM2::App15::doTheWork()");
   CPM_MA
//////////////////////////////////////////////////////////////////////
   Word sec("run control");
```

```
//////////////////////////////////////////////////////////////////////
   Z viewingPeriod;
   R travelRel, tWait=0;
   cpmrh(travelRel);
   cpmrh(viewingPeriod);
   cpmrh(tWait);

//////////////////////////////////////////////////////////////////////
   sec="system data";
//////////////////////////////////////////////////////////////////////

   Z d1;
   R size1=1.,massRel1,vRel1;
   R dtRel=0, xRel1=0, wRel1=0;
   bool useSI=false,noRel=false;

   cpmrh(d1);
   cpmrh(size1);
   cpmrh(noRel);
   cpmrh(massRel1);
   cpmrh(xRel1);
   cpmrh(wRel1);
   cpmrh(vRel1);
   cpmrh(useSI);
   cpmrh(dtRel);
// start working
   Word units=useSI ? "SI":"1";
   OFileStream out1(outFiles[1]);
   Greg date;
   date.now();
   out1()<<"// ";
   date.writeFormatted(out1());
   out1()<<"// output of CpmVQM2::App7::doTheWork()"<<endl;
   out1()<<"units="<<units;
   QMBase::setUnits(units);
   QMBase::setSize(size1);
      // to get the right lattice related data for particle 1
   R m1= noRel ? massRel1 : massRel1*QMBase::mMax(d1);
   // That boosting instances of State correctly, the class
// QMBase has to be adjusted, needs e.g. member mass
   QMBase::setMass(m1);
   out1()<<"d1="<<d1<<endl;
   out1()<<"size1="<<size1<<endl;
   out1()<<"m1="<<m1<<endl;

// kinetic energy as spectral object, which is the most efficient
// type of object for implementing dynamics

   bool ipt=CpmRootX::Z1(d1).isPowOfTwo();
// two Hamiltonians to select (indirectly via d1)
```

```
   Observable ekin= ipt? Observable("eKin",d1) : Observable("eKinF",d1);
// initial state:
   bool unitHeg=true; // aha heg = height
   R_Func gpsi1=R_Func::Gauss(xRel1,wRel1,unitHeg);
   State s1(gpsi1,d1);
   R enr=(s1|ekin*s1).real();
   out1()<<"Kinetic energy befor boost: "<<enr<<endl;
   R v1=noRel ? vRel1 : vRel1*0.5*QMBase::vMax(d1);
   out1()<<"v1="<<v1<<endl;
      // 0.5 inserted since without this vRel1=1
      // corresponds to a wave function mainly at
      // Nyquist frequency. Our representation of the Hamilton
      // operator is not very suitable for such strongly
      // varying wave functions
   s1=s1.boost(v1);
   enr=(s1|ekin*s1).real();
   out1()<<"Kinetic energy after boost: "<<enr<<endl;
   enr=0.5*m1*v1*v1;
   out1()<<"Kinetic energy from mass and velocity: "<<enr<<endl;
   Graph gr;
   R dt=noRel ? dtRel : dtRel*QMBase::tRef(d1);
   out1()<<"dt direct ="<<dt<<endl;
   R travel=noRel ? travelRel : travelRel*size1;
   cpmassert(v1!=0.,loc);
   R tTot=travel/v1;
   Z nSteps=cpmround(tTot/dt);
   out1()<<"nSteps ="<<nSteps<<endl;
   s1.showAsCurves(gr);
   cpmwait(1+tWait);
      // inspecting the initial state
   if (nSteps>0){
      dt=tTot/nSteps;
      out1()<<"dt adjusted ="<<dt<<endl;
      CyclicAlarm cyc(viewingPeriod);
      R nInv=1./nSteps;
      SpectralObject so;
      {  // encapsulating the heavy initialization of so
         bool enforceSymmetry=false;
         // our H2 is known to be symmetric
         bool test=false;
         // true would consume a lot of time but worked in my test cases
         // perfectly
         Z iterMax=128;
         R eps0=0;
         R eps1=1e-7,eps2=1e-6; // test values from the past
         so=ekin.spectralRepresentation(enforceSymmetry,
         test,iterMax,eps0,eps1,eps2); // this is a big beast
      }
   //   so.expi(-dt);
         // very efficient way of letting free Hamiltonian
```

```
         // dynamics act on the boosted state
      R t=0; // exp
      Chebyshev<Observable,State> ch(ekin*C::I,s1);
      for (Z i=0; i<nSteps; ++i){
         t+=dt; // exp
         R ip=i*nInv;
         cpmprogress("loop",ip,1);
         SpectralObject sot=so;
         sot.expi(-t);
         State sSpec=sot*s1;
         State sCheb=ch.step(-t);
         State diff=sSpec-sCheb;

         if (cyc()||i==(nSteps-1)){
            R acc=1e-10;
            gr.setY(Iv(-acc,acc));
         //   s1.showAsCurves(gr);
          //  sCheb.showAsCurves(gr);
            diff.showAsCurves(gr);
            cpmwait(tWait);
         }
      }
   }
   cpmwait(tWaitWorkScreen_);
   CPM_MZ
}


//////////////////////// App16 ////////////////////////////


namespace{
//   const Z dHam=100;
//   R omegaHam=2000; // 50,2000
   R wobble(R const& t, R const& omega)
   {
      R y=::cos(omega*t);
      return y*y;
   }


/******************************

   State dFunc(X2<State,R> const& x, Z const& n, R const& omega)
   {
      R lambda=QMBase::lambdaOsc(n);
      R_Func f2=F1<R,R,R>(omega)(wobble);
      R2_Func pot(R_Func::x2,f2);
      State psi=x.c1();
      R t=x.c2();
```

```
      return psi.sinParHam(pot,t,lambda)*C(0,-1);
   }
******************************/
}


class Ham{ // Hamiltonian
// independent data
   Z n_;
   R omega_;
// dependent data
   R lambda_;
   R2_Func pot_;
public:
   Ham(Z n, R omega):n_(n),omega_(omega),lambda_(QMBase::lambdaOsc(n_)),
      pot_(R_Func::x2,F1<R,R,R>(omega_)(wobble)){}

   State operator()(State const& psi, R t)const
      // acting as D not as H
   {
      return psi.sinParHam(pot_,t,lambda_)*C(0,-1);
   }

   bool timeIndependent()const{ return omega_==0.;}
   bool enforceNormalization()const{ return false;}
};

class Inh{
   State st_;
   R omega_;
   R ampl_;
   R phase_;
public:
   Inh(State const& st, R omega, R ampl, R phase):
      st_(st),omega_(omega),ampl_(ampl),phase_(phase){}
   State operator()(R t)const{
      R x=ampl_*cos(omega_*t+phase_);
      return st_*x;
   }
};


void CpmVQM2::App16::doTheWork()
{
   Z mL=2;
   Word loc("CpmVQM2::App16::doTheWork()");
   CPM_MA
   Z d=100;
   R omega=2000,omega0=4000,ampl0=20,phase0=0.1,x=-0.2,sigma=0.05;
   R dtRel=4e-2;
```

```
Z nSteps=20000;
Z nRunsForOrder=0;
Z viewingPeriod=40;
Z integrationMethod=1;
R tWait=10;
R tWaitTest1=3;
R tWaitTest2=6;
R tWaitTest3=0;
R tWaitTest4=0;
R fracRun=0.5;
bool symmetric=true;

Word sec("system data");
cpmrh(d);
cpmrh(omega0);
cpmrh(ampl0);
cpmrh(phase0);

cpmrh(omega);
cpmrh(x);
cpmrh(sigma);
cpmrh(symmetric);

sec="run control";
cpmrh(dtRel);
cpmrh(nSteps);
cpmrh(nRunsForOrder);
cpmrh(integrationMethod);
cpmrh(fracRun);
cpmrh(viewingPeriod);
cpmrh(tWait);
cpmrh(tWaitTest1);
cpmrh(tWaitTest2);
cpmrh(tWaitTest3);
cpmrh(tWaitTest4);


State psi0(d);
psi0=psi0.ranVal();
psi0=psi0.smooth();

R_Func f=R_Func::Gauss(x,sigma);
if (symmetric) f+=R_Func::Gauss(-x,sigma);
State psi(f,d);
R t=0;

Inh inh(psi0,omega0,ampl0,phase0);
Ham ham(d,omega);

GMPM<Ham,Inh,State> mp(ham,inh,psi,t);
```

```
   mp.setMethod(integrationMethod);

   R dt=QMBase::dtStabilityLimit(d)*dtRel;
   R tSpan=nSteps*dt;
   R t1=0;
   R t2=t1+tSpan*0.45;
   R t3=t1+tSpan;
   R errorSum=0;

   if (tWaitTest1>0){
      R error=GMPM<Ham,Inh,State>::testOfOrder(ham,inh,psi,t1,t2,nSteps\
      ,nRunsForOrder);
      errorSum+=error;
      cpmvalue("errComposition",error,1);
      cpmwait(tWaitTest1);
   }
/**********************

   if (tWaitTest1>0){
      R error=GMPM<Ham,Inh,State>::testOfComposition(ham,inh,psi,t1,t2,t3\
      ,nSteps,nSteps);
      errorSum+=error;
      cpmvalue("errComposition",error,1);
      cpmwait(tWaitTest1);
   }
   if (tWaitTest2>0){
      R error=GMPM<Ham,Inh,State>::testOfReversibility(ham,inh,psi,t1,t3\
      ,2*nSteps);
      errorSum+=error;
      cpmvalue("errReversibility",error,1);
      cpmwait(tWaitTest2);
   }
   if (tWaitTest3>0){
      R error=GMPM<Ham,Inh,State>::testOfNormConservation(ham,inh,psi,t1\
      ,t3\
      ,4*nSteps);
      errorSum+=error;
      cpmvalue("errNormConservation",error,1);
      cpmwait(tWaitTest3);
   }
   if (tWaitTest4>0){
      R error=GMPM<Ham,Inh,State>::testOfConservation(ham,inh,psi,t1,t3\
      ,4*nSteps);
      errorSum+=error;
      cpmvalue("errConservation",error,1);
      cpmwait(tWaitTest4);
   }
   if (errorSum!=0){
      cpmvalue("err",errorSum,4);
   }
```

```
****************************************/
   if (fracRun>0){
      R nFrac=fracRun*nSteps;
      nSteps=cpmround(nFrac);
      Graph gr;
      R_Vector times(nSteps);
  // R_Matrix norms(3,nSteps);
      R_Matrix norms(2,nSteps);
      CyclicAlarm show(viewingPeriod);
      R nInv=1./nSteps;
      C ene0=mp.getEnergy();
      R ea=ene0[2];
      R eb=ene0[1];
      R facInv=-1./ea;
      mp.getPsi().showAsCurves(gr);
      cpmwait(1);
      for (Z i=1;i<=nSteps;++i){
         mp.step(dt);
         R t=mp.getTime();
         times[i]=t;
         norms[1][i]=mp.getNorm()-1.;
         C ene=(mp.getEnergy()-ene0)*facInv;
     // C ene=mp.getEnergy();
         norms[2][i]=ene[1];
    //  norms[3][i]=ene[2];

         if (show()){
            mp.getPsi().showAsCurves(gr);
            cpmprogress("loop",i*nInv);
         }
      }
      gr.clr();
      gr.showGL(times,norms,rainBow,false);
      cpmwait(tWait);
   }
   CPM_MZ
}

///////////////// void CpmVQM2::function(void) /////////////

// functions of signature void *f(void) for final association with a menu
//
// selection


void CpmVQM2::app11()
{
   cpmmessage("app11() started");
   App11 a("app11");
   a.run();
```

```
}

void CpmVQM2::app12()
{
   cpmmessage("app12() started");
   App12 a("app12");
   a.run();
}

void CpmVQM2::app13()
{
   cpmmessage("app13() started");
   App13 a("app13");
   a.run();
}

void CpmVQM2::app14()
{
   cpmmessage("app14() started");
   App14 a("app14");
   a.run();
}

void CpmVQM2::app15()
{
   cpmmessage("app15() started");
   App15 a("app15");
   a.run();
}

void CpmVQM2::app16()
{
   cpmmessage("app16() started");
   App16 a("app16");
   a.run();
}
```

243 vqm2app3.cpp

```
//? vqm2app3.cpp
//? Status of work 2008-10-25.
//?
////////////////////////////////////////////////////////////////////////
//   vqm2app3.cpp
////////////////////////////////////////////////////////////////////////

#include <vqm2app3.h>
#include <vqm2.h>
#include <cpmqm.h>
#include <cpmvectormore.h>
#include <cpmmultiimage.h>
#include <cpmfile.h>

using namespace CpmSystem;
using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmArrays;
using namespace CpmFunctions;
using namespace CpmGraphics;
using namespace CpmImaging;
using CpmAlgorithms::CyclicAlarm;

//////////////////////////////// App17 ////////////////////////////////////

void CpmVQM2::App17::doTheWork()
// Experimenting with two and three separated, weakly interacting
// particles. Also tool for creating the publication data.
// All publication data made with sel==8.
{
   Z mL=1;
   Word loc("CpmVQM2::App17::doTheWork()");
   CPM_MA
   using namespace CpmQM;
   V<OFile> vout(outFiles.dim());
   for (Z it=vout.b();it<=vout.e();++it){ // open OFiles
      vout[it]=OFile(outFiles[it]);
   }
   Z d1=50,d2=50,d3=50,d4=3;
   Z n1=0,n2=0,n3=0;
   R gap=0;

   Word sec="system data";
   cpmrh(gap);
   cpmrh(d1);
   cpmrh(d2);
   cpmrh(d3);
```

```
cpmrhf(d4);

cpmrh(n1);
cpmrh(n2);
cpmrh(n3);

Iv iv1(0,1);
Iv iv2(1,2);
Iv iv3(2,3);
iv2+=gap;
iv3+=gap*2;

Lat1 lat1(iv1,d1);
Lat1 lat2(iv2,d2);
Lat1 lat3(iv3,d3);

if (vout.valInd(1)){
    vout[1]()<<"lat1="<<lat1<<endl;
    vout[1]()<<"lat2="<<lat2<<endl;
    vout[1]()<<"lat3="<<lat3<<endl;
}

R m1=lat1.mMax();
R m2=lat2.mMax();
R m3=lat3.mMax();

R q1=1,q2=1,q3=1;

R lambda1=lat1.lamOsc(m1);
R lambda2=lat2.lamOsc(m2);
R lambda3=lat3.lamOsc(m3);

R p1=2,p2=2,p3=2;
R x1Rel=0.5, x2Rel=0.5, x3Rel=0.5;
R lambda1Rel=1, lambda2Rel=1, lambda3Rel=1;
R m1Rel=0.5, m2Rel=0.5, m3Rel=0.5;
R fac1=1, fac2=1, fac3=1;
R beta=0;
Z out1=0,out2=0,out3=0;
bool homIniOff=true;
bool simplifyFun3=false;
// method parameter in Fun3::step_
R epsilonFun3=0;
// method parameter in Fun3::step_

cpmrh(x1Rel);
cpmrh(x2Rel);
cpmrh(x3Rel);

cpmrh(m1Rel);
```

```
cpmrh(m2Rel);
cpmrh(m3Rel);

cpmrh(q1);
cpmrh(q2);
cpmrh(q3);

cpmrh(lambda1Rel);
cpmrh(lambda2Rel);
cpmrh(lambda3Rel);

cpmrh(p1);
cpmrh(p2);
cpmrh(p3);

cpmrhf(out1); // needs not to be there
   // since 0 is OK
cpmrhf(out2);
cpmrhf(out3);

cpmrh(fac1);
cpmrh(fac2);
cpmrh(fac3);

cpmrh(beta);
cpmrhf(homIniOff);

cpmrhf(simplifyFun3);
cpmrhf(epsilonFun3);

Fun3::eps()=epsilonFun3;
Fun3::sim()=simplifyFun3;


R x1=iv1.x(x1Rel);
R x2=iv2.x(x2Rel);
R x3=iv3.x(x3Rel);

m1*=m1Rel;
m2*=m2Rel;
m3*=m3Rel;

lambda1*=lambda1Rel;
lambda2*=lambda2Rel;
lambda3*=lambda3Rel;

DynDat dd1(m1,q1,lambda1,x1,p1,out1);
DynDat dd2(m2,q2,lambda2,x2,p2,out2);
DynDat dd3(m3,q3,lambda3,x3,p3,out3);
```

```
   sec="interaction data";
   IntActDat dd12(rch,sec);

   if (vout.valInd(1)){
       vout[1]()<<"dd1="<<dd1<<endl;
       vout[1]()<<"dd2="<<dd2<<endl;
       vout[1]()<<"dd3="<<dd3<<endl;
       vout[1]()<<"dd12="<<dd12<<endl;
   }

   R alpha=dd12.alpha();

   if (homIniOff)  DynDat::beta()=0;
   else DynDat::beta()=beta;

   SysLat1 s1(lat1,dd1,n1);
  // SysLat1 s1e(lat1,dd1,n1+1);
   SysLat1 s2(lat2,dd2,n2);
 //  SysLat1 s2e(lat2,dd2,n2+1);
   SysLat1 s3(lat3,dd3,n3);
//   SysLat1 s3e(lat3,dd3,n3+1);
      // eigenstates of one-particle hamiltonian
      // here no homogeneous fiels should be in action,
      // since particle shift (polarization) due to the
      // field should only occur during a dynamical run.

   DynDat::beta()=beta;
      // now DynDat::pot includes a homogeneous field

   SysLat2 s12(s1,s2);

   sec="run control";
   Z nAlpha=1,nBeta=1;
   bool fun3=false;
   cpmrhf(nAlpha);
   cpmrhf(nBeta);
   cpmrhf(fun3);
   RunCon rc(rch,sec);
   R tWaitFinal=6;
   R tWaitFinal2=6;
   cpmrh(tWaitFinal);
   cpmrhf(tWaitFinal2); // needs not to be there
   if (tWaitFinal>0 && rc.tWaitRes<=0) rc.tWaitRes=0.5;
   if (tWaitFinal>0 && rc.tWaitDocu<=0) rc.tWaitDocu=0.5;
      // otherwise res will not be created

   sec="display control";
   DisCon dc(rch,sec);

   dd1.mulLam_(fac1);
```

```
dd2.mulLam_(fac2);
dd3.mulLam_(fac3);
   // changing the coupling constant after
   // having formed the states

Z sel=rc.sel;

Graph g;
if (sel==0){
    TrjLat1(dd1,s1).run_(g,rc,dc);
}
else if (sel==1){
   if (dc.ipolMet<0) dc.ipolMet=0;
   TrjLat2(dd1,dd2,dd12,s12).run_(g,rc,dc);
}
else if (sel==2){
   ; // free
}
else if (sel==3){
   Sep2(dd1,dd2,dd12,s1,s2).run_(g,rc,dc);
}
else if (sel==4){
   using namespace CpmImaging;
   DisCon dcMem=dc;
   if (dc.ipolMet<0) dc.ipolMet=0;
   V< V<R2> > res1=TrjLat2(dd1,dd2,dd12,s12).run_(g,rc,dc);
   dc.ipolMet=dcMem.ipolMet;
   V< V<R2> > res2=Sep2(dd1,dd2,dd12,s1,s2).run_(g,rc,dc);
   V< V<R2> > res=res1&res2;
   res=homDataRan(res);
   g.clr_();
   g.setAutoScl_(true);
   V<Color> vc(3);
   vc[1]=BLUE;
   vc[2]=GREEN;
   vc[3]=RED;

   g.mark(res,vc);
   g.addText("correlations from TrjLat2 and Sep2");
   g.dis();
   cpmwait(tWaitFinal,2);
}
else if (sel==5){
   using namespace CpmImaging;
   DisCon dcMem=dc;
   if (dc.ipolMet<0) dc.ipolMet=0;
   x2vv x1=Sep3(dd1,dd2,dd3,dd12,s1,s2,s3).run_(g,rc,dc);
   V<V<R2> > res1=x1.c1();
   dc.ipolMet=dcMem.ipolMet;
   x2vv x2=Fun3(dd1,dd2,dd3,dd12,s1,s2,s3).run_(g,rc,dc);
```

```
      V<V<R2> > res2=x2.c1();
      V< V<R2> > res=res1&res2;
      res=homDataRan(res);
      g.clr_();
      g.setAutoScl_(true);
      V<Color> vc(3);
      vc[1]=BLUE;
      vc[2]=GREEN;
      vc[3]=RED;

      g.mark(res,vc);
      g.addText("correlations from TrjLat2 and Sep2");
      g.dis();
      cpmwait(tWaitFinal,2);
   }
   else if (sel==6){
      if (dc.ipolMet<0) dc.ipolMet=0;
      SysLat3 s123(s1,s2,s3);
      TrjLat3(dd1,dd2,dd3,dd12,s123).run_(g,rc,dc);
   }
   else if (sel==7){
      if (fun3)  Fun3(dd1,dd2,dd3,dd12,s1,s2,s3).run_(g,rc,dc);
      else Sep3(dd1,dd2,dd3,dd12,s1,s2,s3).run_(g,rc,dc);
   }
   else if (sel==8){ // exact and separated dynamics for
      // three particles. Comparison
      using namespace CpmImaging;
      DisCon dcMem=dc;
      Z i;
      if (dc.ipolMet<0) dc.ipolMet=0;
  //    SysLat1 s2a=s2+s2e;
  //    s2a.nor_();
      SysLat3 s123(s1,s2,s3);
      s123.nor_();
      x2vv x1=TrjLat3(dd1,dd2,dd3,dd12,s123).run_(g,rc,dc);
         // exact dynamics, implemented by the direct midpoint integrator
         // may change rc by automatic time step adjustment, so that the
         // adjusted time step will be used by Sep3::run_ without further
         // adjustment
      if (vout.valInd(1)) vout[1]()<<"rc.dt="<<rc.dt<<endl;
      V<V<R2> > res1=x1.c1();
      V<R2> stExc=res1[8];
      dc.ipolMet=dcMem.ipolMet;
      x2vv x2=fun3 ? Fun3(dd1,dd2,dd3,dd12,s1,s2,s3).run_(g,rc,dc)
               : Sep3(dd1,dd2,dd3,dd12,s1,s2,s3).run_(g,rc,dc);
            // separated dynamics
      V<V<R2> > res2=x2.c1();
// comparing polarization (i.e. total particle shift)
// as a function of time
      V<R2> stSep=res2[8];
```

```
      V<V<R2> > res("",stExc,stSep);
      if (dc.showCls){
          V<R2> stCls=res2[16];
          res&=stCls;
      }
      if (vout.valInd(2)) toGnuPlot(vout[2](),res);
      g.clr_();
      g.setAutoScl_(true);
      g.setSclMem_(false);
      g.mark(res);
      Word mes1("TrjLat3, ");
      Word mes2=fun3 ? "Fun3" : "Sep3";
      Word mes3=": polarization from exact and separated ";
      Word mes4=dc.showCls ? "and classical dynamics" : "dynamics";
      Word mes=mes1&mes2&mes3&mes4;
      g.addText(mes);
      g.dis();
      cpmwait(tWaitFinal,2);
// comparing also quantum correlations from both methods
      V<V<R2> > resCor=res1.resize(7)&res2.resize(7);
      resCor=homDataRan(resCor);
      if (vout.valInd(3)) toGnuPlot(vout[3](),resCor);
      bool homFirst=false; // data already homogenized
      resCor=difOfCorData(resCor,homFirst);
      if (vout.valInd(4)) toGnuPlot(vout[4](),resCor);
      V<Color> vc1("",RED,GREEN,BLUE,ORANGE,MAGENTA,CYAN,DARKGRAY);
      V<Color> vc2("",LIGHTRED,LIGHTGREEN,LIGHTBLUE,YELLOW,
         LIGHTMAGENTA,LIGHTCYAN,LIGHTGRAY);
      g.clr_();
      g.setAutoScl_(true);
      g.setSclMem_(false);
      g.mark(resCor,vc1);
      mes=mes1&mes2&
      ": difference of correlations for exact and separated dynamics";
      g.addText(mes);
      g.dis();
      cpmmessage("talata: came here");
      cpmwait(tWaitFinal2,2);
      V<Word> vw1=x1.c2();
      V<Word> vw2=x2.c2();
      Z dw1=vw1.dim();
      Z dw2=vw2.dim();
      cpmassert(dw1==dw2,loc);
      Word fn("twins_");
      if (dw2>0){ // only then there are images to be processed
         Z nf=dw1;
         for (i=1;i<=nf;++i){
            Word fn1=vw1[i];
            Word fn2=vw2[i];
            Img24 img_i=Img24(fn1).pair(Img24(fn2));
```

```
            img_i.write(fn&cpm(i,3));
         }
      }
   }
   else if (sel==9){
      // should create data which show the dependence of polarization
      // on alpha and beta. For this purpose we interprete the input
      // values as maximum value which, in a linear manner get finally
      // reduced to 0 in nAlpha and nBeta steps.
      using namespace CpmImaging;
      OFileStream ofs1(outFiles[1]);
      DisCon dcMem=dc;
      if (dc.ipolMet<0) dc.ipolMet=0;
      SysLat3 s123(s1,s2,s3);
      V< V<R2> > dat;
      V< Color > colors;
      R_Vector alphas(R2(alpha,0),nAlpha);
      Z nA=alphas.dim();
      R_Vector betas(R2(beta,0),nBeta);
      betas=betas.cut(1); // we don't beta=0, since polarizability is
         // shift/beta
      Z nB=betas.dim();
      R_Vector levels(R2(0,180),nA); // one level for each
         // alpha value
      // n equidistant components starting with p[1] and ending
      for (Z i=1;i<=nA;++i){
         R alphai=alphas[i];
         dd12.alpha()=alphai;
 //        betaj=beta;
         V<R2> dat1,dat2,dat3;
         for (Z j=1;j<=nB;++j){
            R betaj=betas[j];
            if (!homIniOff){ // then the initial states depend on
               // beta and have to be constructed anew
               DynDat::beta()=betaj;
               s1=SysLat1(lat1,dd1,n1);
               s2=SysLat1(lat2,dd2,n2);
               s3=SysLat1(lat3,dd3,n3);
               s123=SysLat3(s1,s2,s3);
            }
            DynDat::beta()=betaj;
            cpmvalues2("alpha,beta",alphai,betaj,3);
            x2vv x1=TrjLat3(dd1,dd2,dd3,dd12,s123).run_(g,rc,dc);
            V< V<R2> > res1=x1.c1();
         // exact dynamics, implemented by the direct midpoint integrator
         // may change rc by automatic time step adjustment, so that the
         // adjusted time step will be used by Sep3::run_ without further
         // adjustment. Note that the time step adjustment depends on
         // alpha and beta
            Z k;
```

```
Z r11d=res1[1].dim();
Z r12d=res1[2].dim();
Z r13d=res1[3].dim();
cpmassert(r11d==r12d,loc);
cpmassert(r12d==r13d,loc);
R_Vector pol1(r11d);
for (k=1;k<=r11d;++k){
   R2 ri=res1[1][k]+res1[2][k]+res1[3][k];
   pol1[k]=ri[2];
}
dc.ipolMet=dcMem.ipolMet;
x2vv x2=fun3 ? Fun3(dd1,dd2,dd3,dd12,s1,s2,s3).
   run_(g,rc,dc)
   : Sep3(dd1,dd2,dd3,dd12,s1,s2,s3).run_(g,rc,dc);
V< V<R2> > res2=x2.c1();
// separated dynamics
Z r21d=res2[1].dim();
Z r22d=res2[2].dim();
Z r23d=res2[3].dim();

Z rc21d=res2[8].dim();
Z rc22d=res2[9].dim();
Z rc23d=res2[10].dim();

cpmassert(r21d==r22d,loc);
cpmassert(r22d==r23d,loc);
cpmassert(r23d==rc21d,loc);
cpmassert(rc21d==rc22d,loc);
cpmassert(rc22d==rc23d,loc);

R_Vector pol2(r21d);
for (k=1;k<=r21d;++k){
   R2 ri=res2[1][k]+res2[2][k]+res2[3][k];
   pol2[k]=ri[2];
}
R_Vector pol2c(r21d);
for (k=1;k<=r21d;++k){
   R2 ri=res2[8][k]+res2[9][k]+res2[10][k];
   pol2c[k]=ri[2];
}
R betaInv=cpminv(betaj);
R p1=extMean(pol1)*betaInv;
R p2=extMean(pol2)*betaInv;
R p3=extMean(pol2c)*betaInv;
dat1<<R2(betaj,p1);
dat2<<R2(betaj,p2);
dat3<<R2(betaj,p3);
cpmnote(alphai);
cpmnote(betaj);
cpmnote(p1);
```

```
            cpmnote(p2);
            cpmnote(p3);
         }// j-loop
         dat<<dat1;
         R level=levels[i];
         colors<<Color(255.,level,level);
         dat<<dat2;
         colors<<Color(level,255.,level);
         dat<<dat3;
         colors<<Color(level,level,255.);
      }//i-loop
      g.clr_();
      g.setAutoScl_(true);
      g.mark(dat,colors);
      g.addText("polarization curves, homIniOff="
         &cpm(homIniOff)&", alphaMax="&cpm(alpha));
      g.dis(true);
      ofs1()<<dat;
      cpmwait(tWaitFinal,2);
   }// sel==9
   else if (sel==10){
      testEC(d1,d2,d3);
   }
   else if (sel==11){
      testEC4(d1,d2,d3,d4);
   }
   else if (sel==12){  // exact and separated dynamics for
      // three particles. Comparison. Form usin Record input
      SysLat3 s123(s1,s2,s3);
      s123.nor_();
      x2vv x1=TrjLat3(dd1,dd2,dd3,dd12,s123).run_(g,rch);
      x2vv x2=Sep3(dd1,dd2,dd3,dd12,s1,s2,s3).run_(g,rch);
   }
   else{
      cpmmessage("unvalid selection");
   }
   CPM_MZ
}

//////////////////////////////// App18 ////////////////////////////////////

/****************
void CpmVQM2::App18::doTheWork()
// Experimenting with a chain of spin 1/2 particles with Heisenberg model
// like interaction
{
   Z mL=1;
   Word loc("CpmVQM2::App18::doTheWork()");
   CPM_MA
   using namespace CpmQM;
```

```
Z np=50;
R h=0, q=0, eps=0, fracRim=0.1;
bool full=true;
R phi0=0.1;
R theta0=0.2;
R flipProbability=0.1;

Word sec="system data";
cpmrh(np);
cpmrh(h);
cpmrh(q);
cpmrhf(full);
cpmrh(eps);
cpmrh(fracRim);
cpmrh(phi0);
cpmrh(theta0);
cpmrh(flipProbability);

sec="run control";
RunCon rc(rch,sec);
R tWaitFinal=6;
cpmrh(tWaitFinal);

sec="display control";
DisCon dc(rch,sec);

Vl<C2> vx(np,C2(theta0,phi0));
Z dRim=cpmround(np*fracRim);
IvZ iv(dRim,np-1-dRim);
for (Z i=iv.b();i<=iv.e();++i){
   if (randomR()<flipProbability){
      vx[i].swap_();
   }
}
Z sel=rc.sel;

Graph g;
if (sel==0){
   SpinN sn(vx,h,q,full);
   sn.setEps_(eps);
   sn.run_(g,rc,dc);
}
if (sel==1){
   R err=C2::blcTest(10000);
   cpmvalue("err",err);
   cpmwait(6);
}
if (sel==2){
   vector<bool> v(4,false);
```

```
      while (CpmAlgorithms::next_1(v)){
         cpmcerr<<endl;
         vector<bool>::const_iterator i;
         for (i=v.begin();i!=v.end();++i){
            cpmcerr<<*i<<" ";
         }
         cpmcerr<<endl;
      }
   }
   if (sel==3){
      Qreg qr(vx);
      QregPar qrp(h,q,full);
      TrjQreg tq(qrp,qr);
      tq.run_(g,rc,dc);

      SpinN sn(vx,h,q,full);
      sn.run_(g,rc,dc);
   }
   else{
      cpmmessage("unvalid selection");
   }
   CPM_MZ
}
*****************/

void CpmVQM2::App18::doTheWork()
// Experimenting with a chain of spin 1/2 particles with Heisenberg model
// like interaction
// and also with areal lattices (images).
{
   Z mL=1;
   Word loc("CpmVQM2::App18::doTheWork()");
   CPM_MA
   using namespace CpmQM;

   Z mp=10,np=10;
   R q=0, eps=0, fracRim=0.1;
   V<R> hAux;
   bool full=true;
   bool reversed=false;
   R phi0=0.1;
   R theta0=0.2;
   R flipProbability=0.1;
   Word image;

   Word sec="system data";
   cpmrhf(mp);
   cpmrhf(np);
   cpmrh(hAux);
   cpmrh(q);
```

```
cpmrhf(full);
cpmrhf(eps);
cpmrhf(fracRim);
cpmrhf(phi0);
cpmrhf(theta0);
cpmrhf(flipProbability);
R3 h(hAux);

sec="run control";
RunCon rc(rch,sec);
cpmrhf(image);

sec="display control";
DisCon dc(rch,sec);

Z sel=rc.sel;
Graph g;
Ia ia(h,q,full);
if (sel==-2){ // selection for publication
   Img24 im(image);
   Frame fr(im,0.1,0.2,false);
   g=Graph(fr);
   SpinN sn(im,ia);
   sn.mod_dt_(rc.dt);
   sn.runRevTest_(g,rc,dc);
}
else if (sel==-1){
   Img24 im(image);
   Frame fr(im);
   g=Graph(fr);
   SpinN sn(im,ia);
   sn.mod_dt_(rc.dt);
   sn.setEps_(eps);
   sn.run_(g,rc,dc);
   if (reversed){
      sn.rev_();
      g.clr_();
      sn.run_(g,rc,dc);
   }
}
else if (sel==0){
   Img24 im(image);
   Frame fr(im);
   g=Graph(fr); // g adjusted to im. No need to
    // adjust cpmconfig.ini
   SpinN sn(im,ia);
   sn.mod_dt_(rc.dt);
   sn.setEps_(eps);
   sn.runSim_(g,rc,dc);
   if (reversed){
```

```
      sn.rev_();
      g.clr_();
      sn.runSim_(g,rc,dc);
   }
}
else if (sel==1){
   Z mn=mp*np;
   C2 c0(theta0,phi0);
   Vl<C2> vx(mn,c0);
   for (Z i=vx.b();i<=vx.e();++i){
      C2 ci=c0.ranVal(i+1); // arg should not be 0
      ci.nor_();
      vx[i]=ci;
   }
   SpinN sn(mp,vx,ia);
   sn.setEps_(eps);
   sn.run_(g,rc,dc);
   R dt=sn.get_dt();
   sn.set_dt_(dt);
   sn.rev_();
   g.clr_();
   sn.runSim_(g,rc,dc);
}
else if (sel==2){
   Z mn=mp*np;
   C2 c0(theta0,phi0);
   Vl<C2> vx(mn,c0);
   for (Z i=vx.b();i<=vx.e();++i){
      C2 ci=c0.ranVal(i+1); // arg should not be 0
      ci.nor_();
      vx[i]=ci;
   }
   SpinN sn(mp,vx,ia);
   sn.setEps_(eps);
   sn.runRevTest_(g,rc,dc);
}
else if (sel==3){
   R err=C2::codTest(1000);
   cpmvalue("err",err,3);
   cpmwait(4,2);
}
else if (sel==4){
   V<Word> imageList;
   sec="run control";
   cpmrh(imageList);
   Z nL=imageList.dim();
   bool doAll=true;
   cpmrhf(doAll);
   R_Vector xV(nL);
   R_Matrix yV(1,nL);
```

```
      R_Matrix eDen(1,nL);
      for (Z i=1;i<=nL;++i){
         image=imageList[i];
         Img24 im(image);
         Frame fr(im);
         g=Graph(fr); //
         SpinN sn(im,ia);
         sn.mod_dt_(rc.dt);
         sn.setEps_(eps);
         xV[i]=sn.hw();
         eDen[1][i]=sn.eTot()[1]/xV[i];
         if (doAll){
            sn.run_(g,rc,dc);
            yV[1][i]=sn.dtMean();
         }
      }
      cpmdebug(xV);
      if (doAll) cpmdebug(yV);
      cpmdebug(eDen);
      Graph gr;
      gr.setWithOrigin(0);
      if (doAll){
         gr.show(xV,yV,V<Color>("",RED),"pixel number dependence of mean\
          time step",0.4,false,false);
         gr.dis(true);
         cpmwait(10,2);
         g.clr_();
      }
      gr.show(xV,eDen,V<Color>("",GREEN),"pixel number dependence of\
       energy density",0.4,false,false);
      gr.dis(true);
      cpmwait(60,2);
   }
   else{
      cpmmessage("unvalid selection");
   }
   CPM_MZ
}

//////////////// void CpmVQM2::function(void) ////////////////////////

// functions of signature void *f(void) for final association with a menu
// selection

void CpmVQM2::app17()
{
   cpmmessage("app17() started");
   App17 a("app17");
   a.run();
}
```

```
void CpmVQM2::app18()
{
   cpmmessage("app18() started");
   App18 a("app18");
   a.run();
}
```

244 survey.txt

```
2008-10-25   output by Ruby class CpmSurvey

         C+- higher-level names:
 Class/struct content of namespaces, together with
 the header files in which these are declared

Directories under consideration are:
 /cygdrive/c/e/cpm
        /sourcepublishing

 Number of files considered: 240
 Number of program lines considered: 119141
 Number of classes and structs found: 466


   CpmAlgorithms  introduced in file  cpmalgorithms.h
   CpmApplication  introduced in file  cpmapplication.h
   CpmArrays  introduced in file  cpmm.h
   CpmCamera2  introduced in file  cpmcamera2.h
   CpmCamera3  introduced in file  cpmcamera3.h
   CpmColCam  introduced in file  cpmcamera.h
   CpmDim2  introduced in file  cpmdim.h
   CpmDim3  introduced in file  cpmdim.h
   CpmDimx  introduced in file  cpmdimx.h
   CpmDynSys  introduced in file  cpmdynaux.h
   CpmFields  introduced in file  cpmfield.h
   CpmFonts  introduced in file  cpmfontgeneration.h
   CpmFourier  introduced in file  cpmfft.h
   CpmFunctions  introduced in file  cpmf.h
   CpmGeo  introduced in file  cpmangle.h
   CpmGraphics  introduced in file  cpmviewport.h
   CpmImaging  introduced in file  cpmhistogr.h
   CpmLinAlg  introduced in file  cpmclinalg.h
   CpmMPI  introduced in file  cpmmpi.h
   CpmMathFound  introduced in file  cpmlogic1.h
   CpmPala1_3  introduced in file  cpmrigbodydyn.h
   CpmPala2_2  introduced in file  cpmsys2aps.h
   CpmPala2_3  introduced in file  cpmsys2aps.h
   CpmPala2_x  introduced in file  cpmextractorx.h
   CpmPalaAction  introduced in file  cpmactionprinciple.h
   CpmPalaAux  introduced in file  cpmsysviewer.h
   CpmPhysics  introduced in file  cpmconstphys.h
   CpmPhysicsx  introduced in file  cpmbodyx.h
   CpmProperties  introduced in file  cpmproperties.h
   CpmQM  introduced in file  cpmqm.h
   CpmQuantumMechanics  introduced in file  cpmfieldoperator.h
   CpmRigidBodyx  introduced in file  cpmcomparx.h
```

```
   CpmRoot  introduced in file  cpmbasictypes.h
   CpmRootX  introduced in file  cpmtypes.h
   CpmStd  introduced in file  cpmnumbers.h
   CpmSystem  introduced in file  cpmsystem.h
   CpmTOP  introduced in file  cpmsys2datx.h
   CpmTests  introduced in file  cpmtestr.h
   CpmTime  introduced in file  cpmgreg.h
   CpmUnits  introduced in file  cpmunits.h
   CpmWaves  introduced in file  cpmmaxw1d.h

Classes and structs by namespace

 Namespace CpmAlgorithms:
   AveragingInterpolator   class in file   cpmalgorithms.h
      interpolation method for vector-valued functions on metric spaces

   BurstAlarm   class in file   cpmalgorithms.h
      double periodic counter

   CyclicAlarm   class in file   cpmalgorithms.h
      periodic counter

   FindRoot   class in file   cpmroot.h
      finds a root of a function

   IntQuant   class in file   cpmalgorithms.h
      Putting integer numbers into boxes

   LatticeBox   class in file   cpmalgorithms.h
      putting points into cells defined by a cartesian product of intervals

   LatticeEnumerator   class in file   cpmalgorithms.h
      enumerating the points in cubic lattices

   Maxi   class in file   cpmmini.h
      finds maximum of a function

   Mini   class in file   cpmmini.h
      finds minimum of a function

   NonCyclicAlarm   class in file   cpmalgorithms.h
      cog-based state machine

   Percentilizer   class in file   cpmalgorithms.h
      handling quartiles (hinges)

   PeriodicAlarm   class in file   cpmalgorithms.h
      periodic counter

   Place   class in file   cpmalgorithms.h
```

```
      creating X-valued lists, satisfying specified conditions

   Quantizer   class in file   cpmalgorithms.h
      Putting real numbers into boxes

   RandomIntegration   class in file   cpmalgorithms.h
      calculating expectation values based on a X-valued list

   SingleAlarm   class in file   cpmalgorithms.h
      1-step counter


 Namespace CpmApplication:
   IniFileBasedApplication   class in file   cpminifilebasapp.h
      creating file-based applications which document their input and output

   Task   class in file   cpminifilebasapp.h
      tool for implementing a menu-like application structure


 Namespace CpmArrays:
   C_Vector   class in file   cpmcvector.h
      class of vectors with C-valued components.

   DisR   class in file   cpmdistribution.h
      distribution of reals

   Distribution   class in file   cpmdistribution.h
      general framework for distributions ( = histograms)

   IpolControl   class in file   cpmsmoothlocalinterpol.h
      interpolation control

   IvZ   class in file   cpmzinterval.h
      'intervals of integers', i.e. contiguous finite subsets of Z

   M   class in file   cpmm.h
      map, associative array, dictionary, hash

   P   class in file   cpmuc.h
      'P' as in 'pointer', constant smart pointer

   Perm   class in file   cpmperm.h
      group of permutations

   Po   class in file   cpmp.h
      Adding order operations to Pp<T>

   Pp   class in file   cpmp.h
      polymorphic smart pointers
```

```
R_Matrix   class in file   cpmrmatrix.h
   real matrices, rich interface

R_Vector   class in file   cpmrvector.h
   vectors with R-valued components, rich interface including arithmetics

S   class in file   cpms.h
   sets of homotypic elements

SVD   class in file   cpmrmatrix.h
   singular value decomposition of real (m,n)-matrices

Set   class in file   cpmset.h
   represents general finite sets

Sr   class in file   cpmsr.h
   S with a rich interface

UseCount   class in file   cpmuc.h
   supporting copy on write

V   class in file   cpmv.h
   vector template, indexing starts with 1

VC   class in file   cpmvc.h
   vectors with C-valued components, rich interface including arithmetics

VV   class in file   cpmv.h
   matrices

VVV   class in file   cpmv.h
   tensors of rank 3

VVVV   class in file   cpmv.h
   tensors of rank 4

VVVVa   class in file   cpmva.h
   version of VVVVo with arithmetic operations

VVVVo   class in file   cpmvo.h
   version of VVVV with order-related operations

VVVa   class in file   cpmva.h
   version of VVVo with arithmetic operations

VVVo   class in file   cpmvo.h
   version of VVV with order-related operations

VVa   class in file   cpmva.h
```

```
      version of VVo with arithmetic operations

   VVo   class in file   cpmvo.h
      version of VV with order-related operations

   Va   class in file   cpmva.h
      version of Vo with arithmetic operations

   Vl   class in file   cpmvl.h
      vector template, indexing starts with 0

   Vo   class in file   cpmvo.h
      version of V with order-related operations

   Vp   class in file   cpmp.h
      polymorphic V template

   Vr   class in file   cpmvr.h
      version of Va with a rich interface

   X2   class in file   cpmx.h
      heterotypic pairs

   X3   class in file   cpmx.h
      heterotypic triplets

   X4   class in file   cpmx.h
      heterotypic 4-tuples

   X5   class in file   cpmx.h
      heterotypic 5-tuples

   X6   class in file   cpmx.h
      heterotypic 6-tuples

   X7   class in file   cpmx.h
      heterotypic 7-tuples

   X8   class in file   cpmx.h
      heterotypic 8-tuples


Namespace CpmCamera2:
   Camera   class in file   cpmcamera2.h
      sees a rectangular part of infinite 2D object space


Namespace CpmCamera3:
   Camera   class in file   cpmcamera3.h
      maps the 3D object space onto a 2D image rectangle
```

```
Namespace CpmDim2:
  AxVec   class in file   cpmdim2.h
     axial vectors, rotation descriptors, 1-dimensional in this case

  Group   class in file   cpmdim2.h
     group of active Euclidian transformations (translations and rotations)

  InTens   class in file   cpmdim2.h
     tensor of inertia, reduces in 2D to a scalar, the moment of inertia

  Plane   class in file   cpmdim2.h
     oriented affine hyperplane, which in this case is a line

  RefSys   class in file   cpmdim2.h
     orthonormal, positively oriented reference system in space

  Spc   class in file   cpmdim2.h
     affine Euclidian space, instances are points and not vectors

  SubsetOfPlane   class in file   cpmdim2.h
     subset of a Plane  defined via an indicator function

  Vec   class in file   cpmdim2.h
     space of translations and velocities


Namespace CpmDim3:
  AxVec   class in file   cpmdim3.h
     axial vectors, rotation descriptors

  Group   class in file   cpmdim3.h
     group of active Euclidean transformations (translations and rotations)

  InTens   class in file   cpmdim3.h
     tensors of inertia

  Plane   class in file   cpmdim3.h
     oriented affine plane

  RefSys   class in file   cpmdim3.h
     orthonormal, positively oriented reference system in space

  Spc   class in file   cpmdim3.h
     affine Euclidian space, instances are points and not vectors

  SubsetOfPlane   class in file   cpmdim3.h
     subset of a Plane defined via an indicator function
```

```
  Vec   class in file   cpmdim3.h
     vectors, space of point differences and velocities


Namespace CpmDimx:
  BndSph   class in file   cpmdimx.h
     BndSph = Bounding Sphere

  Box   class in file   cpmshapesx.h
     rectangular parallelepipeds

  BoxDat   class in file   cpmshapesx.h
     data which determine a box

  Boxoid   class in file   cpmshapesx.h
     objects between box and ellipsoid

  BoxoidDat   class in file   cpmshapesx.h
     data of which determine a boxoid

  Cylinder   class in file   cpmshapesx.h
     cylinder

  CylinderDat   class in file   cpmshapesx.h
     data which determine a cylinder

  DisPar   class in file   cpmdimx.h
     discretization parameters

  Ellipsoid   class in file   cpmshapesx.h
     ellipsoids

  EllipsoidDat   class in file   cpmshapesx.h
     data which determine an ellipsoid

  Form   class in file   cpmshapesx.h
     shape determining function objects

  IndCon   class in file   cpmlineartx.h
     indexing control

  LineArt   class in file   cpmlineartx.h
     a variety of graphical objects in combination

  ParOfSpc   class in file   cpmparofspcx.h
     partition of space for parallelization of particle dynamics

  Path   class in file   cpmdimx.h
     array of points as a discrete representation of a curve
```

```
  Shp   class in file   cpmshapesx.h
     shapes

  Skeleton   class in file   cpmdimx.h
     directed graph of points

  Sphere   class in file   cpmshapesx.h
     spheres

  SphereDat   class in file   cpmshapesx.h
     data which determine a sphere

  SubsetOfSpace   class in file   cpmdimx.h
     subsets of space with boolean operations

  Texture   class in file   cpmdimx.h
     textures

  Triangle   class in file   cpmdimx.h
     triangle, few methods

  WireFrame   class in file   cpmdimx.h
     array of Skeleton objects

  Wrt   class in file   cpmlineartx.h
     Wrt=writing, actually printable symbols


Namespace CpmDynSys:
  CPM_CN   class in file   cpmpardynsys.h
     = ParDynSys dynamical system of particles, granular matter

  StepControl   class in file   cpmdynaux.h
     a data collection providing I/O and initialization


Namespace CpmFields:
  Field   class in file   cpmfield.h
     fields on which a group acts

  ScalarField   class in file   cpmfield.h
     scalar fields


Namespace CpmFonts:
  Font   class in file   cpmfontgeneration.h
     a font is a list of letters

  Let   class in file   cpmfontgeneration.h
     letter, where letter means a printable symbol
```

```
  PrlFont   class in file   cpmfontgeneration.h
     preliminary font is a list of preliminary letters

  PrlLet   class in file   cpmfontgeneration.h
     preliminary concept of a letter


Namespace CpmFourier:
  FD1   class in file   cpmfftmore.h
     frequency domain 1D

  FD2   class in file   cpmfftmore.h
     frequency domain 2D

  FD3   class in file   cpmfftmore.h
     frequency domain 3D

  SD1   class in file   cpmfftmore.h
     space domain 1D

  SD2   class in file   cpmfftmore.h
     space domain 2D

  SD3   class in file   cpmfftmore.h
     space domain 3D


Namespace CpmFunctions:
  Bind1   class in file   cpmf.h
     binding the first parameter

  Bind2   class in file   cpmf.h
     binding the second parameter

  ConvertDomain   class in file   cpmf.h
     converting the function domain

  ConvertRange   class in file   cpmf.h
     converting the function range

  F   class in file   cpmfl.h
     functions as a class

  F1   class in file   cpmfl.h
     functions with one parameter

  F2   class in file   cpmfl.h
     functions with two parameters
```

```
  F3   class in file   cpmfl.h
     functions with three parameters

  F4   class in file   cpmfl.h
     functions with four parameters

  F5   class in file   cpmf.h
     functions with five parameters

  F6   class in file   cpmf.h
     functions with six parameters

  F_2   class in file   cpmf.h
     functions of two variables

  Fa   class in file   cpmfa.h
     version of Fo with arithmetics operations

  FncObj   class in file   cpmfl.h
     function objects

  Fo   class in file   cpmfo.h
     version of F with order-related operations

  Fr   class in file   cpmfr.h
     version of Fa with rich interface

  R2_Func   class in file   cpmr2func.h
     R-valued functions on RR with arithmetics

  R_Func   class in file   cpmrfunc.h
     functions R to R

  R_FuncRec   class in file   cpmrfuncrec.h
     functions as recordable objects

  Rn   class in file   cpmrnfunc.h
     R_Vector with specified dimension

  Rn_Func   class in file   cpmrnfunc.h
     R1-valued functions on Rn<n>

  RomCon   class in file   cpmrfunc.h
     control of Romberg integration


Namespace CpmGeo:
  Angle   class in file   cpmangle.h
     angles
```

```
  DiGraph   class in file   cpmdigraph.h
     directed graph

  DiIv   class in file   cpminterval.h
     directed interval in R

  DisField   class in file   cpmdiscretespace.h
     discrete field

  DisSpc   class in file   cpmdiscretespace.h
     directed graph with position descriptors attached to vertices

  GroupGraph   class in file   cpmgroupgraph.h
     directed graph with group elements attached

  Iv   class in file   cpminterval.h
     open intervals in R


Namespace CpmGraphics:
  ColRef   class in file   cpmviewport.h
     lean 24-bit color-values

  Fig   class in file   cpmgraph.h
     relevant for making Latex figures

  Figure   class in file   cpmvectorgraphics.h
     the analogon of a screen window as a 'paper window', related to LaTex

  Font   class in file   cpmviewport.h
     only Helvetica 12 needed from this font system of GLUT

  Frame   class in file   cpmframe.h
     rectangular subsets of the screen with pixel structure

  Frames   class in file   cpmframes.h
     array of Frame objects

  FrmText   class in file   cpmframe.h
     Frame text

  Graph   class in file   cpmgraph.h
     Frame as rectangular part of object space

  Img24   class in file   cpmimg24.h
     class of 24-bit (R,G,B)-images

  Rec   class in file   cpmviewport.h
     pixel rectangle which always fits Viewport::win()
```

```
  TriAng   class in file   cpmimg24.h
     triangles in the plane, geometry only as needed for graphics

  VecGraph   class in file   cpmvectorgraphics.h
     vector graphics, directed graph with vertices of type C

  VecGraphs   class in file   cpmvectorgraphics.h
     array of VecGraph objects

  Viewport   class in file   cpmviewport.h
     lean interface to the system's graphical capabilities

  rgb   class in file   cpmviewport.h
     Z-valued red green blue

  xy   class in file   cpmviewport.h
     graphical points

  xyxy   class in file   cpmviewport.h
     graphical rectangles


Namespace CpmImaging:
  Color   class in file   cpmimagingtools.h
     data of CpmGraphics::rgb + gamma, aritmetics

  Histogram   class in file   cpmhistogr.h
     histograms: generation, visualization, analysis

  Image   class in file   cpmimage.h
     real-valued image, gray image with high dynamic range

  ImagingTools   class in file   cpmimagingtools.h
     collection of helper functions

  Mask   class in file   cpmimagingtools.h
     tool for defining convolution

  MultiImage   class in file   cpmmultiimage.h
     real-valued multi-plane image,  typedef MultiImage<R> ImgR

  Rect   class in file   cpmimagingtools.h
     rectangular sub-area of the unit-square

  Response   class in file   cpmimagingtools.h
     versatile conversion from R to screen color

  Roi   class in file   cpmimagingtools.h
     region of interest
```

```
  ShowControl   class in file   cpmimagingtools.h
     collection of display control parameters


Namespace CpmLinAlg:
  C2   class in file   cpmclinalg.h
     tuples of two complex elements

  C3   class in file   cpmclinalg.h
     tuples of 3 complex elements

  C4   class in file   cpmclinalg.h
     tuples of 4 complex elements

  CC2   class in file   cpmclinalg.h
     complex 2*2 matrices

  CC3   class in file   cpmclinalg.h
     complex 3*3 matrices

  CC4   class in file   cpmclinalg.h
     complex 4*4 matrices

  CCC2   class in file   cpmclinalg.h
     pairs of CC2

  CCCC2   class in file   cpmclinalg.h
     pairs of CCC2, for implementation

  H   class in file   cpmh.h
     H  quaternions

  R2   class in file   cpmlinalg.h
     tuples of two real elements

  R3   class in file   cpmlinalg.h
     tuples of 3 real elements

  R4   class in file   cpmlinalg.h
     tuples of 4 real elements

  RR1   class in file   cpmlinalg.h
     real 1*1 matrices

  RR2   class in file   cpmlinalg.h
     real 2*2 matrices

  RR3   class in file   cpmlinalg.h
     real 3*3 matrices
```

```
  RR4   class in file   cpmlinalg.h
     real 4*4 matrices

  Z2   class in file   cpmlinalg.h
     tuples of two integer elements

  Z3   class in file   cpmlinalg.h
     tuples of three integer elements

  Z4   class in file   cpmlinalg.h
     tuples of 4 integer elements


Namespace CpmMPI:
  Com   class in file   cpmmpi.h
     class version of MPI's communicator concept for parallel computing

  Com   class in file   cpmmpi.h
     trivial implementation of Com


Namespace CpmMathFound:
  Binary_node   class in file   cpmlogic1.h
     binary nodes in expression trees

  Expr_node   class in file   cpmlogic1.h
     expression nodes

  Leaf   class in file   cpmlogic1.h
     leafs in expression trees

  NatNum   class in file   cpmmathfound.h
     natural numbers {1,2,3, ...} in 'stone age representation'

  Nr   class in file   cpmnr.h
     natural numbers, based on recursion

  Prop   class in file   cpmlogic1.h
     atomic propositions

  PropLog   class in file   cpmlogic1.h
     propositional logic

  Unary_node   class in file   cpmlogic1.h
     unary nodes in expression trees

  V_bool   class in file   cpmlogic1.h
     array of bool-values, helper class
```

```
Namespace CpmPala1_3:
  App1RigBodyDyn   class in file   cpmrigbodydyn.h
     appl

  DynCon   class in file   cpmrigbodydyn.h
     dynamics control

  RigBodyDyn   class in file   cpmrigbodydyn.h
     rigid body dynamics

  TypQun   class in file   cpmrigbodydyn.h
     'typical quantities'


Namespace CpmPala2_2:
  Sys2Appl   class in file   cpmsys2aps.h
     Sys2 application 2D


Namespace CpmPala2_3:
  Sys2Appl   class in file   cpmsys2aps.h
     Sys2 application 3D


Namespace CpmPala2_x:
  ParExt   class in file   cpmextractorx.h
     particle extractor


Namespace CpmPalaAction:
  ActionAppl   class in file   cpmactionprinciple.h
     action application


Namespace CpmPalaAux:
  SysViewerAppl   class in file   cpmsysviewer.h
     viewer application


Namespace CpmPhysics:
  ACCELERATION   class in file   cpmsi.h
     m*s^-2

  ANGLE   class in file   cpmsi.h
     unit rad = radian = 1

  AREA   class in file   cpmsi.h
     m^2

  CHARGE   class in file   cpmsi.h
```

```
   electrical charge == Coulomb [C] = s*A

CURRENT   class in file   cpmsi.h
   CURRENT() = A

DENSITY   class in file   cpmsi.h
   kg*m^-3

ENERGY   class in file   cpmsi.h
   J = N*m == m^2 * kg * s^-2

FORCE   class in file   cpmsi.h
   N == m * kg * s^-2

INTENSITY   class in file   cpmsi.h
   INTENSITY = cd

LENGTH   class in file   cpmsi.h
   LENGTH() = m

MASS   class in file   cpmsi.h
   MASS() = kg

ParticleViewer   class in file   cpmparticleviewer.h
   particle viewer

RESISTANCE   class in file   cpmsi.h
   electrical resistance Ohm = V/A

SOLID_ANGLE   class in file   cpmsi.h
   unit sterrad = steradian = 1

SUBSTANCE   class in file   cpmsi.h
   SUBSTANCE() = mol

Si   class in file   cpmsi.h
   class for description of physical quantities

TEMPERATURE   class in file   cpmsi.h
   TEMPERATURE() = K

TIME   class in file   cpmsi.h
   TIME() = s

VELOCITY   class in file   cpmsi.h
   m*s^-1

VOLTAGE   class in file   cpmsi.h
   V
```

```
  VOLUME   class in file   cpmsi.h
     m^3


Namespace CpmPhysicsx:
  Body   class in file   cpmbodyx.h
     external bodies

  Drum   class in file   cpmbodyx.h
     a rotating rough cylindrical cavity

  DrumData   class in file   cpmbodyx.h
     data of a rotating drum

  ElcDipole   class in file   cpmforcesx.h
     electric dipoles

  ExtSys   class in file   cpmbodyx.h
     external systems

  Fen   class in file   cpmbodyx.h
     separator (fence) that divides the pot

  FenData   class in file   cpmbodyx.h
     data for a grid-like separator (fence) of the pot

  FieldE   class in file   cpmforcesx.h
     E-field with group action

  FieldSpace   class in file   cpmforcesx.h
     electric, magnetic, gravitational fieldstrengths into one linear space

  ForTor   class in file   cpmforcesx.h
     force and torque combined

  Force   class in file   cpmforcesx.h
     force

  HomFil   class in file   cpmbodyx.h
     homogeneous field

  HomFilData   class in file   cpmbodyx.h
     data for a homogeneous field

  Lid   class in file   cpmbodyx.h
     a cover of the pot

  LidData   class in file   cpmbodyx.h
     data of a lid for the container vessel
```

```
  MagDipole   class in file   cpmforcesx.h
     magnetic dipoles

  MagPole   class in file   cpmforcesx.h
     magnetic point charges

  PatchCharge   class in file   cpmforcesx.h
     charged planar rectangle (patch)

  PointCharge   class in file   cpmforcesx.h
     point charges

  PointMass   class in file   cpmforcesx.h
     point masses

  Pot   class in file   cpmbodyx.h
     box-shaped vessel

  PotData   class in file   cpmbodyx.h
     data for a container vessel

  ScalarField   class in file   cpmforcesx.h
     scalar field with group action

  SourceOfVectorField   class in file   cpmforcesx.h
     Sources of static electromagnetic and gravitational fields.

  SphTank   class in file   cpmbodyx.h
     spherical tank

  SphTankData   class in file   cpmbodyx.h
     data for a spherical tank

  VecBH   class in file   cpmforcesx.h
     magnetic induction B and magnetic field strength H combined

  VecE   class in file   cpmforcesx.h
     electrical field strength E

  VecG   class in file   cpmforcesx.h
     gravitational field strength

  VectorField   class in file   cpmforcesx.h
     FieldSpace-valued field with group action

  VibData   class in file   cpmbodyx.h
     vibration data


Namespace CpmProperties:
```

```
  BodyProperties   class in file   cpmproperties.h
     prop. surrounding bodies

  ContactProperties   class in file   cpmproperties.h
     data describing particle contact

  DumbbellData   class in file   cpmproperties.h
     data of dumbbell-shaped grains

  NuggetData   class in file   cpmproperties.h
     data of nugget-like grains

  PairData   class in file   cpmproperties.h
     data of pair-shaped grains

  ParEnsPrp   class in file   cpmproperties.h
     Particle Ensemble Properties

  ParticleProperties   class in file   cpmproperties.h
     particle properties

  SysControl   class in file   cpmproperties.h
     system control parameters

  SysControlState   class in file   cpmproperties.h
     system control parameters as non-static data


Namespace CpmQM:
  CM3   class in file   cpmqm.h
     classical model of 3 particles

  DisCon   class in file   cpmqm.h
     display control

  DynDat   class in file   cpmqm.h
     dynamical data

  Fun3   class in file   cpmqm.h
     fundamental version of Sep3

  Ia   class in file   cpmqm.h
     interaction descriptor for spin one half

  IntActDat   class in file   cpmqm.h
     interaction data

  Ky   class in file   cpmqm.h
     lean class of keys, for forming map<Ky,C>
```

```
  Lat1   class in file   cpmqm.h
     lattice in one dimension

  Qreg   class in file   cpmqm.h
     array of qubits

  QregPar   class in file   cpmqm.h
     parameters needed in TrjQreg

  RunCon   class in file   cpmqm.h
     run control

  Sep2   class in file   cpmqm.h
     quantum system made of two separated subsystems

  Sep3   class in file   cpmqm.h
     quantum system made of three separated subsystems

  SpinN   class in file   cpmqm.h
     spin lattice quantum system

  SysLat1   class in file   cpmqm.h
     quantum system on a one-dimensional lattice

  SysLat2   class in file   cpmqm.h
     quantum system on a two-dimensional lattice

  SysLat3   class in file   cpmqm.h
     quantum system on a three-dimensional lattice

  TrjLat1   class in file   cpmqm.h
     trajectories of Lat1 systems

  TrjLat2   class in file   cpmqm.h
     trajectories of Lat2 systems

  TrjLat3   class in file   cpmqm.h
     trajectories of Lat3 systems

  TrjQreg   class in file   cpmqm.h
     Trajectory of a Qreg system


Namespace CpmQuantumMechanics:
  Chebyshev   class in file   cpmqmtemplates.h
     The Chebyshev one-step integraton method

  FieldOperator   class in file   cpmfieldoperator.h
     field operators as functions FieldState x R -->  FieldState
```

```
FieldState   class in file   cpmfieldstate.h
   state space in occupation number representation

GMPM   class in file   cpmqmtemplates.h
   general midpoint method

Ky   class in file   cpmfieldstate.h
   index set for an orthonormal base of FieldSpace

MidPoint   class in file   cpmqmtemplates.h
   direct midPoint integraton method

Observable   class in file   cpmobservable.h
   observable as matrix

Operator   class in file   cpmoperator.h
   observables as functions

QMBase   class in file   cpmstate.h
   quantum mechnics on a linear lattice, basics

SpectralMeasure   class in file   cpmobservable.h
   eigenvalues and spectral projectors

SpectralObject   class in file   cpmobservable.h
   eigenvectors/values in combination

State   class in file   cpmstate.h
   finite-dimensional QM state space

State2   class in file   cpmstate2.h
   fixed-dimensional Quantum mechanical 2-particle state space

State2Data   class in file   cpmstate2.h
   data that define a State2 object

State2Trj   class in file   cpmstate2.h
   State2 trajectory

State2Trj2   class in file   cpmstate2.h
   State2 trajectory 2

Trj2   class in file   cpmtrj.h
   array of trajectories from various methods for comparison

Trj2DMI   class in file   cpmtrj.h
   adds velocity and hamiltonian to TrjT

TrjBase   class in file   cpmtrj.h
   basis for trajectory classes
```

```
  TrjCon   class in file   cpmtrj.h
     trajectory control

  TrjDMI   class in file   cpmtrj.h
     adding velocity to TrjT

  TrjDMI_FilOpr   class in file   cpmfieldoperator.h
     trajectory method: DMI

  TrjDMI_Obs   class in file   cpmtrj.h
     trajectory from observable

  TrjDMI_Opr   class in file   cpmoperator.h
     trajectories det. by operators

  TrjSR   class in file   cpmtrj.h
     trajectory from a SpectralObject

  TrjT   class in file   cpmtrj.h
     adds to TrjBase the state descriptor x_


Namespace CpmRigidBodyx:
  ComPar   class in file   cpmcomparx.h
     composite particles

  ComParC   class in file   cpmcomparx.h
     constant quantities of composite particles

  ComParV   class in file   cpmcomparx.h
     variable quantities of composite particles

  ParEns   class in file   cpmparensx.h
     particle ensemble

  PhyParPrp   class in file   cpmcomparx.h
     physical particle properties

  PointPar   class in file   cpmcomparx.h
     point particles

  PointParC   class in file   cpmcomparx.h
     constant quantities of point particles

  PointParV   class in file   cpmcomparx.h
     variable quantities of point particles

  RigidObject   class in file   cpmcomparx.h
     describes the kinematical state of a rigid body
```

```
Namespace CpmRoot:
  AbsSqr   class in file   cpmnumbers.h
     absoute (value) squared

  AbsSqr<L>   class in file   cpmnumbers.h
     absoute (value) squared

  AbsSqr<N>   class in file   cpmnumbers.h
     absoute (value) squared

  AbsSqr<R>   class in file   cpmnumbers.h
     absoute (value) squared

  AbsSqr<Rh>   class in file   cpmnumbers.h
     absoute (value) squared

  AbsSqr<Z>   class in file   cpmnumbers.h
     absoute (value) squared

  AbsSqr<bool>   class in file   cpmnumbers.h
     absoute (value) squared

  AbsSqr<string>   class in file   cpmnumbers.h
     here string is interpreted as an array of L's

  AbsVal   class in file   cpmnumbers.h
     absoute value

  AbsVal<L>   class in file   cpmnumbers.h
     absoute value

  AbsVal<N>   class in file   cpmnumbers.h
     absoute value

  AbsVal<R>   class in file   cpmnumbers.h
     absoute value

  AbsVal<Rh>   class in file   cpmnumbers.h
     absoute value

  AbsVal<Z>   class in file   cpmnumbers.h
     absoute value

  AbsVal<bool>   class in file   cpmnumbers.h
     absoute value

  AbsVal<string>   class in file   cpmnumbers.h
     here string is interpreted as an array of L's
```

```
C   class in file   cpmc.h
   complex numbers

Comp   class in file   cpmnumbers.h
   compare

Comp<L>   class in file   cpmnumbers.h
Comp<N>   class in file   cpmnumbers.h
Comp<R>   class in file   cpmnumbers.h
Comp<Rh>   class in file   cpmnumbers.h
Comp<Z>   class in file   cpmnumbers.h
Comp<bool>   class in file   cpmnumbers.h
Comp<string>   class in file   cpmnumbers.h
Conj   class in file   cpmnumbers.h
   conjugation

Conj<L>   class in file   cpmnumbers.h
Conj<N>   class in file   cpmnumbers.h
Conj<R>   class in file   cpmnumbers.h
Conj<Rh>   class in file   cpmnumbers.h
Conj<Z>   class in file   cpmnumbers.h
Conj<bool>   class in file   cpmnumbers.h
Conj<string>   class in file   cpmnumbers.h
DisVal   class in file   cpmnumbers.h
DisVal<L>   class in file   cpmnumbers.h
DisVal<N>   class in file   cpmnumbers.h
DisVal<R>   class in file   cpmnumbers.h
DisVal<Rh>   class in file   cpmnumbers.h
DisVal<Z>   class in file   cpmnumbers.h
DisVal<bool>   class in file   cpmnumbers.h
DisVal<string>   class in file   cpmnumbers.h
HashVal   class in file   cpmnumbers.h
   hash value

HashVal<L>   class in file   cpmnumbers.h
HashVal<N>   class in file   cpmnumbers.h
HashVal<R>   class in file   cpmnumbers.h
HashVal<Rh>   class in file   cpmnumbers.h
HashVal<Z>   class in file   cpmnumbers.h
HashVal<bool>   class in file   cpmnumbers.h
HashVal<string>   class in file   cpmnumbers.h
IO   class in file   cpmnumbers.h
   input output class template

IO<L>   class in file   cpmnumbers.h
   specialization of IO

IO<N>   class in file   cpmnumbers.h
   specialization of IO
```

```
IO<R>   class in file   cpmnumbers.h
   specialization of IO

IO<Rh>   class in file   cpmnumbers.h
   specialization of IO

IO<Z>   class in file   cpmnumbers.h
   specialization of IO

IO<bool>   class in file   cpmnumbers.h
   specialization of IO

IO<string>   class in file   cpmnumbers.h
   specialization of IO

Inv   class in file   cpmnumbers.h
   inverse

Inv<L>   class in file   cpmnumbers.h
   inverse

Inv<N>   class in file   cpmnumbers.h
   inverse

Inv<R>   class in file   cpmnumbers.h
   inverse

Inv<Rh>   class in file   cpmnumbers.h
   inverse

Inv<Z>   class in file   cpmnumbers.h
   inverse

Inv<bool>   class in file   cpmnumbers.h
   inverse

Inv<string>   class in file   cpmnumbers.h
   reverse string

Name   class in file   cpmword.h
   tool class template for defining the nameOf function

Name<CpmRootX::fpVoidToVoid>   class in file   cpmtypes.h
   also this type needs a name

Name<CpmRootX::fpWordToVoid>   class in file   cpmtypes.h
   also this type needs a name

Name<L>   class in file   cpmword.h
```

```
   L: letter

Name<N>   class in file   cpmword.h
   N: natural numbers

Name<R>   class in file   cpmword.h
   R: real numbers

Name<Rh>   class in file   cpmword.h
   Rh: real number of reduced accuracy

Name<Z>   class in file   cpmword.h
   Z: integers

Name<bool>   class in file   cpmword.h
   specialization

Name<char>   class in file   cpmword.h
   specialization

Name<string>   class in file   cpmword.h
   specialization

Neutrals   class in file   cpmnumbers.h
   neutral elements

Neutrals<L>   class in file   cpmnumbers.h
Neutrals<N>   class in file   cpmnumbers.h
Neutrals<R>   class in file   cpmnumbers.h
Neutrals<Rh>   class in file   cpmnumbers.h
Neutrals<Z>   class in file   cpmnumbers.h
Neutrals<bool>   class in file   cpmnumbers.h
Neutrals<string>   class in file   cpmnumbers.h
Q   class in file   cpmq.h
   rational numbers

RanVal   class in file   cpmnumbers.h
   random value

RanVal<L>   class in file   cpmnumbers.h
RanVal<N>   class in file   cpmnumbers.h
RanVal<R>   class in file   cpmnumbers.h
RanVal<Rh>   class in file   cpmnumbers.h
RanVal<Z>   class in file   cpmnumbers.h
RanVal<bool>   class in file   cpmnumbers.h
RanVal<string>   class in file   cpmnumbers.h
TestVal   class in file   cpmnumbers.h
   test value

TestVal<L>   class in file   cpmnumbers.h
```

```
  TestVal<N>   class in file   cpmnumbers.h
  TestVal<R>   class in file   cpmnumbers.h
  TestVal<Rh>   class in file   cpmnumbers.h
  TestVal<Z>   class in file   cpmnumbers.h
  TestVal<bool>   class in file   cpmnumbers.h
  TestVal<string>   class in file   cpmnumbers.h
  ToWord   class in file   cpmword.h
  ToWord<L>   class in file   cpmword.h
  ToWord<N>   class in file   cpmword.h
  ToWord<R>   class in file   cpmword.h
  ToWord<Rh>   class in file   cpmword.h
  ToWord<Z>   class in file   cpmword.h
  ToWord<bool>   class in file   cpmword.h
  ToWord<string>   class in file   cpmword.h
  Word   class in file   cpmword.h
     wrapping charcter strings, rich functionaliy


Namespace CpmRootX:
  B   class in file   cpmtypes.h
     boolean values as a class

  R1   class in file   cpmtypes.h
     real numbers as a class

  Record   class in file   cpmrecord.h
     data structure for communication with ini-files

  RecordHandler   class in file   cpmrecordhandler.h
     tool for reading and writing a Record

  Recordable   class in file   cpmrecord.h
     deriving from this class provides IO for complex data

  Text   class in file   cpmrecord.h
     texts as lists of Words

  Z1   class in file   cpmtypes.h
     integer numbers as a class


Namespace CpmSystem:
  Error   class in file   cpmsystem.h
     used if C+- classes throw errors

  Exception   class in file   cpmsystem.h
     simple debugging tool

  IFile   class in file   cpmfile.h
     input file
```

```
  IFileStream   class in file   cpmsystem.h
     input file stream

  Message   class in file   cpmsystem.h
     provides basic output (unidirectional communication)

  OFile   class in file   cpmfile.h
     output file

  OFileStream   class in file   cpmsystem.h
     output file stream

  Timer   class in file   cpmsystem.h
     implements e.g. getSecondsLeft()


Namespace CpmTOP:
  Sys2Dat   class in file   cpmsys2datx.h
     sys2 data


Namespace CpmTests:
  Assignment   class in file   cpmtestr.h
     testing consistency of assignement

  CopyConstructor   class in file   cpmtestr.h
     testing consistency of copy constructor

  DefaultConstructor   class in file   cpmtestr.h
     testing consistency of default constructor

  PolymorphicMulti   class in file   cpmtestr.h
     testing polymorphic containers

  PolymorphicSingle   class in file   cpmtestr.h
     testing polymorphic containers

  StrictAssignment   class in file   cpmtestr.h
     testing strict consistency of assignement

  Sym   class in file   cpmtestr.h
     testing symmetry of equality

  TestBase   class in file   cpmtestr.h
     base class for classes which test that class T

  TestOfPolymorphism   class in file   cpmtestr.h
     testing consistent polymorphic behavior
```

```
  Test_F   class in file   cpmtestr.h
     Testing the chaining operation of F

  Test_Vp   class in file   cpmtestr.h
     testing behavior of Vp<>

  Test_c   class in file   cpmtestr.h
     testing complex classes

  Test_lin   class in file   cpmtestr.h
     class for testing classes that

  Test_logic   class in file   cpmtestr.h
     test models of propositional logic

  Test_r   class in file   cpmtestr.h
     testing the r-interface

  Test_set   class in file   cpmtestr.h
     testing behavior of sets

  Test_sv   class in file   cpmtestr.h
     testing the strict value interface

  Test_v   class in file   cpmtestr.h
     testing the value interface

  Trans   class in file   cpmtestr.h
     testing transitivity of equality


Namespace CpmTime:
  Greg   class in file   cpmgreg.h
     time, date, and Gregorian Calendar

  TimeStyle   class in file   cpmgreg.h
     how to interprete time with respect to the globe


Namespace CpmWaves:
  HarmOsc   class in file   cpmpsi.h
     1-dimensional harmonic oscillator

  Psi   class in file   cpmpsi.h
     1-dimensional discrete wave function

  RV2   class in file   cpmmaxw1d.h
     E and H field combined, tool class

  TEM   class in file   cpmmaxw1d.h
```

```
   electromagnetic field in TEM mode

TEM2   class in file   cpmmaxw1d.h
   TEM in C+- style

Wave   class in file   cpmwaves.h
   states of a discrete string

Waves   class in file   cpmwaves.h
   array of waves on strata with one-point overlap
```

245 headerdependencies.txt

```
2008-10-25    output of Ruby class UM1::HeaderFileHierarchy
The directories under consideration are:

        /cygdrive/c/e/cpm/sourcepublishing
The 134 header files under consideration are in alphabetic order:
    cpm0listing.out
    cpmactionprinciple.h
    cpmalgorithms.h
    cpmangle.h
    cpmapplication.h
    cpmbas.h
    cpmbasicinterfaces.h
    cpmbasictypes.h
    cpmbody.h
    cpmbodyx.h
    cpmc.h
    cpmcamera.h
    cpmcamera2.h
    cpmcamera3.h
    cpmcamerax.h
    cpmclinalg.h
    cpmcompar.h
    cpmcomparx.h
    cpmconstphys.h
    cpmcvector.h
    cpmdefinitions.h
    cpmdigraph.h
    cpmdim.h
    cpmdim2.h
    cpmdim3.h
    cpmdimdef.h
    cpmdimx.h
    cpmdiscretespace.h
    cpmdistribution.h
    cpmdynaux.h
    cpmextractor.h
    cpmextractorx.h
    cpmf.h
    cpmfa.h
    cpmfft.h
    cpmfftmore.h
    cpmfield.h
    cpmfieldoperator.h
    cpmfieldstate.h
    cpmfile.h
    cpmfl.h
    cpmfo.h
```

```
cpmfontgeneration.h
cpmforces.h
cpmforcesx.h
cpmfr.h
cpmframe.h
cpmframes.h
cpmgraph.h
cpmgreg.h
cpmgroupgraph.h
cpmh.h
cpmhistogr.h
cpmimage.h
cpmimagingtools.h
cpmimg24.h
cpminifilebasapp.h
cpminterfaces.h
cpminterval.h
cpmlinalg.h
cpmlineart.h
cpmlineartx.h
cpmlisting.out
cpmlogic1.h
cpmm.h
cpmmacros.h
cpmmathfound.h
cpmmaxw1d.h
cpmmini.h
cpmmpi.h
cpmmultiimage.h
cpmnr.h
cpmnumbers.h
cpmobservable.h
cpmoperator.h
cpmp.h
cpmpardynsys.h
cpmparens.h
cpmparensx.h
cpmparofspc.h
cpmparofspcx.h
cpmparticleviewer.h
cpmperm.h
cpmproperties.h
cpmpsi.h
cpmq.h
cpmqm.h
cpmqmtemplates.h
cpmr2func.h
cpmrecord.h
cpmrecordhandler.h
cpmrfunc.h
```

```
    cpmrfuncrec.h
    cpmrigbodydyn.h
    cpmrmateig.h
    cpmrmatrix.h
    cpmrnfunc.h
    cpmroot.h
    cpmrvector.h
    cpms.h
    cpmset.h
    cpmshapes.h
    cpmshapesx.h
    cpmsi.h
    cpmsmoothlocalinterpol.h
    cpmsr.h
    cpmstate.h
    cpmstate2.h
    cpmsys2.h
    cpmsys2aps.h
    cpmsys2dat.h
    cpmsys2datx.h
    cpmsys2x.h
    cpmsystem.h
    cpmsystemdependencies.h
    cpmsysviewer.h
    cpmtestr.h
    cpmtrj.h
    cpmtypes.h
    cpmuc.h
    cpmunits.h
    cpmv.h
    cpmva.h
    cpmvc.h
    cpmvectorgraphics.h
    cpmvectormore.h
    cpmviewport.h
    cpmvl.h
    cpmvo.h
    cpmvr.h
    cpmwaves.h
    cpmword.h
    cpmx.h
    cpmzinterval.h

 Which header files are included in a header file ?


cpm0listing.out includes:
  none of the files under consideration
cpmactionprinciple.h includes:
  cpminifilebasapp.h
```

```
cpmalgorithms.h includes:
  cpmf.h  cpminterval.h  cpmvo.h  cpmx.h
cpmangle.h includes:
  cpmc.h  cpmtypes.h  cpmv.h
cpmapplication.h includes:
  cpmv.h
cpmbas.h includes:
  cpmangle.h  cpmc.h  cpmfr.h  cpmgreg.h
  cpmm.h  cpmp.h  cpmsr.h  cpmtypes.h
  cpmvr.h
cpmbasicinterfaces.h includes:
  cpmbasictypes.h
cpmbasictypes.h includes:
  cpmdefinitions.h
cpmbody.h includes:
  cpmbodyx.h  cpmcompar.h  cpmframes.h  cpmv.h
  cpmvectormore.h
cpmbodyx.h includes:
  none of the files under consideration
cpmc.h includes:
  cpmword.h
cpmcamera.h includes:
  cpmcamera2.h  cpmcamera3.h  cpmdim.h  cpmgraph.h
  cpmimagingtools.h  cpmrecordhandler.h  cpmvectorgraphics.h
cpmcamera2.h includes:
  cpmcamerax.h
cpmcamera3.h includes:
  cpmcamerax.h
cpmcamerax.h includes:
  none of the files under consideration
cpmclinalg.h includes:
  cpmlinalg.h
cpmcompar.h includes:
  cpmcamera.h  cpmcomparx.h  cpmdim.h  cpmdimdef.h
  cpmdistribution.h  cpmforces.h  cpmmacros.h  cpmpardef.h
  cpmrmatrix.h  cpmshapes.h
cpmcomparx.h includes:
  none of the files under consideration
cpmconstphys.h includes:
  cpmnumbers.h
cpmcvector.h includes:
  cpmc.h  cpmrvector.h
cpmdefinitions.h includes:
  none of the files under consideration
cpmdigraph.h includes:
  cpmgraph.h  cpmlinalg.h  cpms.h
cpmdim.h includes:
  cpmangle.h  cpmdim2.h  cpmdim3.h  cpmdimx.h
  cpmfield.h  cpminterval.h  cpmlinalg.h  cpmr2func.h
  cpmvectorgraphics.h  cpmvo.h  cpmx.h
```

```
cpmdim2.h includes:
  none of the files under consideration
cpmdim3.h includes:
  none of the files under consideration
cpmdimdef.h includes:
  none of the files under consideration
cpmdimx.h includes:
  none of the files under consideration
cpmdiscretespace.h includes:
  cpmdigraph.h  cpmf.h
cpmdistribution.h includes:
  cpmf.h  cpmgraph.h  cpminterfaces.h  cpminterval.h
  cpmlinalg.h  cpmtypes.h  cpmv.h
cpmdynaux.h includes:
  cpmdimdef.h  cpminterval.h  cpmrecord.h  cpmtypes.h
cpmextractor.h includes:
  cpmbody.h  cpmextractorx.h  cpmfile.h  cpmpardynsys.h
  cpmsys2dat.h
cpmextractorx.h includes:
  none of the files under consideration
cpmf.h includes:
  cpmv.h  cpmx.h
cpmfa.h includes:
  cpmfo.h  cpmtypes.h
cpmfft.h includes:
  cpmc.h  cpmv.h
cpmfftmore.h includes:
  cpmfft.h  cpmva.h
cpmfield.h includes:
  cpmf.h
cpmfieldoperator.h includes:
  cpmfieldstate.h  cpmoperator.h
cpmfieldstate.h includes:
  cpmoperator.h  cpmx.h
cpmfile.h includes:
  cpmv.h
cpmfl.h includes:
  cpmuc.h  cpmword.h
cpmfo.h includes:
  cpmf.h
cpmfontgeneration.h includes:
  cpmimg24.h  cpmm.h
cpmforces.h includes:
  cpmcamera.h  cpmconstphys.h  cpmdim.h  cpmdimdef.h
  cpmfield.h  cpmforcesx.h  cpmproperties.h
cpmforcesx.h includes:
  none of the files under consideration
cpmfr.h includes:
  cpmfa.h  cpmvo.h
cpmframe.h includes:
```

```
  cpmfontgeneration.h  cpmimagingtools.h  cpmimg24.h
cpmframes.h includes:
  cpmframe.h
cpmgraph.h includes:
  cpmframe.h  cpmr2func.h  cpmrmatrix.h
cpmgreg.h includes:
  cpmangle.h
cpmgroupgraph.h includes:
  cpmv.h
cpmh.h includes:
  cpmlinalg.h
cpmhistogr.h includes:
  cpmframe.h  cpmrfunc.h  cpmrmatrix.h
cpmimage.h includes:
  cpmc.h  cpmfa.h  cpmfftmore.h  cpmgraph.h
  cpmhistogr.h  cpmimagingtools.h  cpmva.h
cpmimagingtools.h includes:
  cpmalgorithms.h  cpmangle.h  cpmc.h  cpmf.h
  cpminterval.h  cpmva.h  cpmviewport.h
cpmimg24.h includes:
  cpminterval.h  cpmviewport.h
cpminifilebasapp.h includes:
  cpmapplication.h  cpmf.h  cpmp.h  cpmrecordhandler.h
cpminterfaces.h includes:
  cpmbasicinterfaces.h  cpmmpi.h  cpmnumbers.h
cpminterval.h includes:
  cpmlinalg.h  cpmx.h  cpmzinterval.h
cpmlinalg.h includes:
  cpmc.h  cpmtypes.h  cpmv.h
cpmlineart.h includes:
  cpmcamera.h  cpmdim.h  cpmdimdef.h  cpmlineartx.h
cpmlineartx.h includes:
  none of the files under consideration
cpmlisting.out includes:
  none of the files under consideration
cpmlogic1.h includes:
  cpms.h
cpmm.h includes:
  cpms.h
cpmmacros.h includes:
  none of the files under consideration
cpmmathfound.h includes:
  cpminterfaces.h
cpmmaxw1d.h includes:
  cpmgraph.h  cpmrfunc.h  cpmrvector.h
cpmmini.h includes:
  cpmfa.h  cpmrmatrix.h
cpmmpi.h includes:
  cpmdefinitions.h
cpmmultiimage.h includes:
```

```
  cpmdiscretespace.h  cpmframes.h  cpmimage.h
cpmnr.h includes:
  none of the files under consideration
cpmnumbers.h includes:
  cpmbasictypes.h
cpmobservable.h includes:
  cpmalgorithms.h  cpmangle.h  cpmf.h  cpmp.h
  cpmrecord.h  cpmrmatrix.h  cpmstate.h  cpmv.h
cpmoperator.h includes:
  cpmtrj.h
cpmp.h includes:
  cpmv.h
cpmpardynsys.h includes:
  cpmalgorithms.h  cpmdiscretespace.h  cpmdynaux.h  cpmgraph.h
  cpmimagingtools.h  cpmm.h  cpmmacros.h  cpmp.h
  cpmpardef.h  cpmparticleviewer.h  cpmproperties.h  cpmrecordhandler.h
  cpmrmatrix.h  cpmvectormore.h
cpmparens.h includes:
  cpmcompar.h  cpmparensx.h
cpmparensx.h includes:
  none of the files under consideration
cpmparofspc.h includes:
  cpmalgorithms.h  cpmangle.h  cpmdim.h  cpmdimdef.h
  cpmforces.h  cpmfr.h  cpmframe.h  cpmgroupgraph.h
  cpmimagingtools.h  cpmlinalg.h  cpmmini.h  cpmparofspcx.h
  cpmrecord.h  cpmshapes.h  cpmvectormore.h
cpmparofspcx.h includes:
  none of the files under consideration
cpmparticleviewer.h includes:
  cpmcamera.h  cpmimage.h  cpmimg24.h  cpminterval.h
  cpmlinalg.h  cpmmultiimage.h  cpmnumbers.h  cpmr2func.h
cpmperm.h includes:
  cpmvo.h
cpmproperties.h includes:
  cpmimagingtools.h  cpmrecordhandler.h  cpmrvector.h
cpmpsi.h includes:
  cpmconstphys.h  cpmcvector.h  cpmrfunc.h  cpmrmatrix.h
cpmq.h includes:
  cpmtypes.h
cpmqm.h includes:
  cpmc.h  cpmclinalg.h  cpmgraph.h  cpmrecordhandler.h
  cpmvl.h
cpmqmtemplates.h includes:
  cpmc.h  cpmf.h  cpmrvector.h
cpmr2func.h includes:
  cpmc.h  cpmrfunc.h
cpmrecord.h includes:
  cpmm.h  cpmtypes.h  cpmv.h  cpmword.h
  cpmx.h
cpmrecordhandler.h includes:
```

```
  cpmrecord.h
cpmrfunc.h includes:
  cpmfa.h  cpmrmatrix.h  cpmsmoothlocalinterpol.h
cpmrfuncrec.h includes:
  cpmrecord.h  cpmrfunc.h
cpmrigbodydyn.h includes:
  cpmcompar.h  cpminifilebasapp.h  cpmp.h
cpmrmateig.h includes:
  cpmrmatrix.h
cpmrmatrix.h includes:
  cpmf.h  cpmrvector.h
cpmrnfunc.h includes:
  cpmalgorithms.h  cpmfa.h  cpmrvector.h
cpmroot.h includes:
  cpmf.h  cpmtypes.h
cpmrvector.h includes:
  cpminterval.h  cpmlinalg.h  cpmperm.h  cpmvr.h
cpms.h includes:
  cpmtypes.h  cpmvo.h
cpmset.h includes:
  cpmf.h  cpmnumbers.h  cpmv.h  cpmword.h
cpmshapes.h includes:
  cpmdim.h  cpmshapesx.h
cpmshapesx.h includes:
  none of the files under consideration
cpmsi.h includes:
  cpmq.h  cpmtypes.h  cpmvr.h
cpmsmoothlocalinterpol.h includes:
  cpmtypes.h  cpmv.h
cpmsr.h includes:
  cpms.h
cpmstate.h includes:
  cpmc.h  cpmconstphys.h  cpmgraph.h  cpminterfaces.h
  cpmrfunc.h  cpmtypes.h  cpmvl.h
cpmstate2.h includes:
  cpmstate.h  cpmtrj.h
cpmsys2.h includes:
  cpmextractor.h  cpmpardynsys.h  cpmsys2dat.h  cpmsys2x.h
cpmsys2aps.h includes:
  cpminifilebasapp.h  cpmsys2.h
cpmsys2dat.h includes:
  cpmparofspc.h  cpmsys2datx.h
cpmsys2datx.h includes:
  none of the files under consideration
cpmsys2x.h includes:
  none of the files under consideration
cpmsystem.h includes:
  cpmword.h
cpmsystemdependencies.h includes:
  none of the files under consideration
```

```
cpmsysviewer.h includes:
  cpmcamera.h  cpmimagingtools.h  cpminifilebasapp.h
cpmtestr.h includes:
  cpmc.h  cpmfr.h  cpmp.h  cpmsr.h
  cpmtypes.h  cpmvr.h
cpmtrj.h includes:
  cpmobservable.h
cpmtypes.h includes:
  cpmmacros.h  cpmsystem.h  cpmx.h
cpmuc.h includes:
  cpmbasictypes.h
cpmunits.h includes:
  cpmnumbers.h
cpmv.h includes:
  cpmvl.h
cpmva.h includes:
  cpmtypes.h  cpmvo.h
cpmvc.h includes:
  cpmc.h  cpmvl.h
cpmvectorgraphics.h includes:
  cpmdigraph.h  cpmgraph.h  cpmrfunc.h
cpmvectormore.h includes:
  cpmalgorithms.h  cpmrmatrix.h
cpmviewport.h includes:
  cpmvl.h
cpmvl.h includes:
  cpmfl.h  cpmmacros.h  cpmzinterval.h
cpmvo.h includes:
  cpmv.h
cpmvr.h includes:
  cpmva.h
cpmwaves.h includes:
  cpmlinalg.h  cpmrfunc.h  cpmrmatrix.h  cpmv.h
cpmword.h includes:
  cpminterfaces.h
cpmx.h includes:
  cpmword.h
cpmzinterval.h includes:
  cpmsystem.h

 In which header files the header files are included ?


cpm0listing.out is included in:
  no header file
cpmactionprinciple.h is included in:
  no header file
cpmalgorithms.h is included in:
  cpmimagingtools.h  cpmobservable.h  cpmpardynsys.h  cpmparofspc.h
  cpmrnfunc.h  cpmvectormore.h
```

```
cpmangle.h is included in:
  cpmbas.h  cpmdim.h  cpmgreg.h  cpmimagingtools.h
  cpmobservable.h  cpmparofspc.h
cpmapplication.h is included in:
  cpminifilebasapp.h
cpmbas.h is included in:
  no header file
cpmbasicinterfaces.h is included in:
  cpminterfaces.h
cpmbasictypes.h is included in:
  cpmbasicinterfaces.h  cpmnumbers.h  cpmuc.h
cpmbody.h is included in:
  cpmextractor.h
cpmbodyx.h is included in:
  cpmbody.h
cpmc.h is included in:
  cpmangle.h  cpmbas.h  cpmcvector.h  cpmfft.h
  cpmimage.h  cpmimagingtools.h  cpmlinalg.h  cpmqm.h
  cpmqmtemplates.h  cpmr2func.h  cpmstate.h  cpmtestr.h
  cpmvc.h
cpmcamera.h is included in:
  cpmcompar.h  cpmforces.h  cpmlineart.h  cpmparticleviewer.h
  cpmsysviewer.h
cpmcamera2.h is included in:
  cpmcamera.h
cpmcamera3.h is included in:
  cpmcamera.h
cpmcamerax.h is included in:
  cpmcamera2.h  cpmcamera3.h
cpmclinalg.h is included in:
  cpmqm.h
cpmcompar.h is included in:
  cpmbody.h  cpmparens.h  cpmrigbodydyn.h
cpmcomparx.h is included in:
  cpmcompar.h
cpmconstphys.h is included in:
  cpmforces.h  cpmpsi.h  cpmstate.h
cpmcvector.h is included in:
  cpmpsi.h
cpmdefinitions.h is included in:
  cpmbasictypes.h  cpmmpi.h
cpmdigraph.h is included in:
  cpmdiscretespace.h  cpmvectorgraphics.h
cpmdim.h is included in:
  cpmcamera.h  cpmcompar.h  cpmforces.h  cpmlineart.h
  cpmparofspc.h  cpmshapes.h
cpmdim2.h is included in:
  cpmdim.h
cpmdim3.h is included in:
  cpmdim.h
```

```
cpmdimdef.h is included in:
  cpmcompar.h  cpmdynaux.h  cpmforces.h  cpmlineart.h
  cpmparofspc.h
cpmdimx.h is included in:
  cpmdim.h
cpmdiscretespace.h is included in:
  cpmmultiimage.h  cpmpardynsys.h
cpmdistribution.h is included in:
  cpmcompar.h
cpmdynaux.h is included in:
  cpmpardynsys.h
cpmextractor.h is included in:
  cpmsys2.h
cpmextractorx.h is included in:
  cpmextractor.h
cpmf.h is included in:
  cpmalgorithms.h  cpmdiscretespace.h  cpmdistribution.h  cpmfield.h
  cpmfo.h  cpmimagingtools.h  cpminifilebasapp.h  cpmobservable.h
  cpmqmtemplates.h  cpmrmatrix.h  cpmroot.h  cpmset.h
cpmfa.h is included in:
  cpmfr.h  cpmimage.h  cpmmini.h  cpmrfunc.h
  cpmrnfunc.h
cpmfft.h is included in:
  cpmfftmore.h
cpmfftmore.h is included in:
  cpmimage.h
cpmfield.h is included in:
  cpmdim.h  cpmforces.h
cpmfieldoperator.h is included in:
  no header file
cpmfieldstate.h is included in:
  cpmfieldoperator.h
cpmfile.h is included in:
  cpmextractor.h
cpmfl.h is included in:
  cpmvl.h
cpmfo.h is included in:
  cpmfa.h
cpmfontgeneration.h is included in:
  cpmframe.h
cpmforces.h is included in:
  cpmcompar.h  cpmparofspc.h
cpmforcesx.h is included in:
  cpmforces.h
cpmfr.h is included in:
  cpmbas.h  cpmparofspc.h  cpmtestr.h
cpmframe.h is included in:
  cpmframes.h  cpmgraph.h  cpmhistogr.h  cpmparofspc.h
cpmframes.h is included in:
  cpmbody.h  cpmmultiimage.h
```

```
cpmgraph.h is included in:
  cpmcamera.h  cpmdigraph.h  cpmdistribution.h  cpmimage.h
  cpmmaxw1d.h  cpmpardynsys.h  cpmqm.h  cpmstate.h
  cpmvectorgraphics.h
cpmgreg.h is included in:
  cpmbas.h
cpmgroupgraph.h is included in:
  cpmparofspc.h
cpmh.h is included in:
  no header file
cpmhistogr.h is included in:
  cpmimage.h
cpmimage.h is included in:
  cpmmultiimage.h  cpmparticleviewer.h
cpmimagingtools.h is included in:
  cpmcamera.h  cpmframe.h  cpmimage.h  cpmpardynsys.h
  cpmparofspc.h  cpmproperties.h  cpmsysviewer.h
cpmimg24.h is included in:
  cpmfontgeneration.h  cpmframe.h  cpmparticleviewer.h
cpminifilebasapp.h is included in:
  cpmactionprinciple.h  cpmrigbodydyn.h  cpmsys2aps.h  cpmsysviewer.h
cpminterfaces.h is included in:
  cpmdistribution.h  cpmmathfound.h  cpmstate.h  cpmword.h
cpminterval.h is included in:
  cpmalgorithms.h  cpmdim.h  cpmdistribution.h  cpmdynaux.h
  cpmimagingtools.h  cpmimg24.h  cpmparticleviewer.h  cpmrvector.h
cpmlinalg.h is included in:
  cpmclinalg.h  cpmdigraph.h  cpmdim.h  cpmdistribution.h
  cpmh.h  cpminterval.h  cpmparofspc.h  cpmparticleviewer.h
  cpmrvector.h  cpmwaves.h
cpmlineart.h is included in:
  no header file
cpmlineartx.h is included in:
  cpmlineart.h
cpmlisting.out is included in:
  no header file
cpmlogic1.h is included in:
  no header file
cpmm.h is included in:
  cpmbas.h  cpmfontgeneration.h  cpmpardynsys.h  cpmrecord.h
cpmmacros.h is included in:
  cpmcompar.h  cpmpardynsys.h  cpmtypes.h  cpmvl.h
cpmmathfound.h is included in:
  no header file
cpmmaxw1d.h is included in:
  no header file
cpmmini.h is included in:
  cpmparofspc.h
cpmmpi.h is included in:
  cpminterfaces.h
```

```
cpmmultiimage.h is included in:
  cpmparticleviewer.h
cpmnr.h is included in:
  no header file
cpmnumbers.h is included in:
  cpmconstphys.h  cpminterfaces.h  cpmparticleviewer.h  cpmset.h
  cpmunits.h
cpmobservable.h is included in:
  cpmtrj.h
cpmoperator.h is included in:
  cpmfieldoperator.h  cpmfieldstate.h
cpmp.h is included in:
  cpmbas.h  cpminifilebasapp.h  cpmobservable.h  cpmpardynsys.h
  cpmrigbodydyn.h  cpmtestr.h
cpmpardynsys.h is included in:
  cpmextractor.h  cpmsys2.h
cpmparens.h is included in:
  no header file
cpmparensx.h is included in:
  cpmparens.h
cpmparofspc.h is included in:
  cpmsys2dat.h
cpmparofspcx.h is included in:
  cpmparofspc.h
cpmparticleviewer.h is included in:
  cpmpardynsys.h
cpmperm.h is included in:
  cpmrvector.h
cpmproperties.h is included in:
  cpmforces.h  cpmpardynsys.h
cpmpsi.h is included in:
  no header file
cpmq.h is included in:
  cpmsi.h
cpmqm.h is included in:
  no header file
cpmqmtemplates.h is included in:
  no header file
cpmr2func.h is included in:
  cpmdim.h  cpmgraph.h  cpmparticleviewer.h
cpmrecord.h is included in:
  cpmdynaux.h  cpmobservable.h  cpmparofspc.h  cpmrecordhandler.h
  cpmrfuncrec.h
cpmrecordhandler.h is included in:
  cpmcamera.h  cpminifilebasapp.h  cpmpardynsys.h  cpmproperties.h
  cpmqm.h
cpmrfunc.h is included in:
  cpmhistogr.h  cpmmaxw1d.h  cpmpsi.h  cpmr2func.h
  cpmrfuncrec.h  cpmstate.h  cpmvectorgraphics.h  cpmwaves.h
cpmrfuncrec.h is included in:
```

```
  no header file
cpmrigbodydyn.h is included in:
  no header file
cpmrmateig.h is included in:
  no header file
cpmrmatrix.h is included in:
  cpmcompar.h  cpmgraph.h  cpmhistogr.h  cpmmini.h
  cpmobservable.h  cpmpardynsys.h  cpmpsi.h  cpmrfunc.h
  cpmrmateig.h  cpmvectormore.h  cpmwaves.h
cpmrnfunc.h is included in:
  no header file
cpmroot.h is included in:
  no header file
cpmrvector.h is included in:
  cpmcvector.h  cpmmaxw1d.h  cpmproperties.h  cpmqmtemplates.h
  cpmrmatrix.h  cpmrnfunc.h
cpms.h is included in:
  cpmdigraph.h  cpmlogic1.h  cpmm.h  cpmsr.h
cpmset.h is included in:
  no header file
cpmshapes.h is included in:
  cpmcompar.h  cpmparofspc.h
cpmshapesx.h is included in:
  cpmshapes.h
cpmsi.h is included in:
  no header file
cpmsmoothlocalinterpol.h is included in:
  cpmrfunc.h
cpmsr.h is included in:
  cpmbas.h  cpmtestr.h
cpmstate.h is included in:
  cpmobservable.h  cpmstate2.h
cpmstate2.h is included in:
  no header file
cpmsys2.h is included in:
  cpmsys2aps.h
cpmsys2aps.h is included in:
  no header file
cpmsys2dat.h is included in:
  cpmextractor.h  cpmsys2.h
cpmsys2datx.h is included in:
  cpmsys2dat.h
cpmsys2x.h is included in:
  cpmsys2.h
cpmsystem.h is included in:
  cpmtypes.h  cpmzinterval.h
cpmsystemdependencies.h is included in:
  no header file
cpmsysviewer.h is included in:
  no header file
```

```
cpmtestr.h is included in:
  no header file
cpmtrj.h is included in:
  cpmoperator.h  cpmstate2.h
cpmtypes.h is included in:
  cpmangle.h  cpmbas.h  cpmdistribution.h  cpmdynaux.h
  cpmfa.h  cpmlinalg.h  cpmq.h  cpmrecord.h
  cpmroot.h  cpms.h  cpmsi.h  cpmsmoothlocalinterpol.h
  cpmstate.h  cpmtestr.h  cpmva.h
cpmuc.h is included in:
  cpmfl.h
cpmunits.h is included in:
  no header file
cpmv.h is included in:
  cpmangle.h  cpmapplication.h  cpmbody.h  cpmdistribution.h
  cpmf.h  cpmfft.h  cpmfile.h  cpmgroupgraph.h
  cpmlinalg.h  cpmobservable.h  cpmp.h  cpmrecord.h
  cpmset.h  cpmsmoothlocalinterpol.h  cpmvo.h  cpmwaves.h
cpmva.h is included in:
  cpmfftmore.h  cpmimage.h  cpmimagingtools.h  cpmvr.h
cpmvc.h is included in:
  no header file
cpmvectorgraphics.h is included in:
  cpmcamera.h  cpmdim.h
cpmvectormore.h is included in:
  cpmbody.h  cpmpardynsys.h  cpmparofspc.h
cpmviewport.h is included in:
  cpmimagingtools.h  cpmimg24.h
cpmvl.h is included in:
  cpmqm.h  cpmstate.h  cpmv.h  cpmvc.h
  cpmviewport.h
cpmvo.h is included in:
  cpmalgorithms.h  cpmdim.h  cpmfr.h  cpmperm.h
  cpms.h  cpmva.h
cpmvr.h is included in:
  cpmbas.h  cpmrvector.h  cpmsi.h  cpmtestr.h
cpmwaves.h is included in:
  no header file
cpmword.h is included in:
  cpmc.h  cpmfl.h  cpmrecord.h  cpmset.h
  cpmsystem.h  cpmx.h
cpmx.h is included in:
  cpmalgorithms.h  cpmdim.h  cpmf.h  cpmfieldstate.h
  cpminterval.h  cpmrecord.h  cpmtypes.h
cpmzinterval.h is included in:
  cpminterval.h  cpmvl.h
```