

# The C+– Tutorial

Ulrich Mutze  
www.ulrichmutze.de

January 12, 2012

As explained in more detail in the article ‘On the C+– project’, [www.ulrichmutze.de/softwaredescriptions/cpmProject.pdf](http://www.ulrichmutze.de/softwaredescriptions/cpmProject.pdf) C+– programming *is* C++ programming. Therefore, with respect to compiling, linking, and program execution there are no new aspects to be considered.

The main feature of C+– is that it allows avoiding pointers, without lacking the functionality usually brought about by pointers. As a benefit, one is unable to create memory leaks or the kind of disaster resulting from bad deletions<sup>1</sup>

(Nearly) all C+– classes implement the *strict value interface*. C+– objects thus behave as *values* (and not as references).

This article explains the basic features of C+–, mainly by realistic and amply commented, code examples. These examples are not meant to be read line by line. Scanning over the stuff will let the eyes stick at the regions of interest.

Here comes the first example, which does not yet involve graphics.

## 1 Defining main without the help of the C+– application framework

This is an example of a console application and thus has as entry point the well-known function `int main(int argc, char* arv[])`. Of course, C+– classes — as any other C++ classes — can be used in the implementation of GUI applications. If the graphical capabilities of C+– classes are to be deployed for the generation of screen graphics one needs to link with OpenGL and GLUT. Even without such libraries, each screen image that C+– can create in the presence of these libraries, can be created without them as an image file in the portable pixmap format (\*.ppm) for any desired pixel number (i.e. not limited to what a computer screen can show).

---

<sup>1</sup> as we read in the ‘Annotated C++ Reference Manual’ by Ellis and Stroustrup, AT&T 1990, p. 63: ‘Bad deletions are usually not detected immediately, and programs containing them are therefore among the nastiest to debug. Almost any effort to avoid such bad deletions is worthwhile.’ Creating C+– is such an effort.

## 1.1 Code example tut1.cpp

```

/// tut1.cpp
/// C++ by Ulrich Mutze. Status of work 2012-01-12.
/// Copyright (c) 2012 Ulrich Mutze, www.ulrichmutze.de
/// All rights reserved

/*****
    Comparing std::vector and CpmArays::V with respect to speed
    of copy-construction and assignment.
*****/
#include <cpmbas.h>
/*****
1. C++ filenames start with 'cpm' and contain neither underscores nor
    blanks. Therefore, even projects employing files from many sources are
    not likely to run into problems with non-unique file names.

```

This directive includes all C++ header files needed here; 'bas' stands for 'basics'. Notice that we have not written `#include "cpmbas.h"` and thus have assumed that our C++ project - let it be named `tut1` - has a list of include directories defined. Who wants to use C++ for more than a single project, should hold C++ files in directories separate from his project files. The C++ files to be used in the project `tut1` are on my computer in `xxx/cpm/cpm0/include` and `xxx/cpm/cpm0/source`. The `tut1` project files are in `yyy/tut1/include` and `yyy/tut1/source`. In `yyy/tut1/include` there are the project specific headers among which there need to be two C++ related configuration files: `cpmdefinitions.h` and `cpmsystemdependencies.h`. Customizing these files allows us to control the behavior of the C++ classes in our project, as will be explained later. In `yyy/tut1/source` the present project-defining main source `tut1.cpp` is to be placed. Now we are in a position to define the file content of project `tut1`: Its include directories have to be set as `yyy/tut1/include` and `xxx/cpm/cpm0/include` and the files to be compiled (translation units) have to be chosen as `yyy/tut1/source/tut1.cpp` and `xxx/cpm/cpm0/source/cpmbas.cpp`.

1.1 Details (skip on first reading ?):

The file `cpmbas.cpp` is actually a collection of all other files in `xxx/cpm/cpm0/source`. Its content is

```

#include "cpmc.cpp"
#include "cpmangle.cpp"
#include "cpmv.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"
End of 1.1
*****/
using namespace CpmRoot;
/*****
2. C+- namespace names start with 'Cpm' followed by an identifier
   which starts with a capital letters.

```

The namespace CpmRoot is rather small, so that the present using directive should be applicable also in projects in which classes from various sources are being used.

3. The C+- names for integer, real, and complex numbers are CpmRoot::Z, CpmRoot::R, Cpmoot::C. CpmRoot::Word wraps std::string and adds functionality to it.

The main effect of this using directive is that we may use these names simply as Z, R, C, Word.

#### 3.1 Details (skip on first reading ?):

There are the less ubiquitous types L, Rh, and N in CpmRoot: L for unsigned characters ('L' for 'letter', after 'integer promotion' the values range from 0 to 255), Rh for 'R with half the storage size', N for 'natural numbers', i.e. unsigned integers. Types L,Z,N,R,Rh are typedefs for unsigned char, int, unsigned long int, double, float. If we add in file cpmdefinition.h the macro #define CPM\_LONG, we change Z to long int, and R to long double (which, unfortunately, is the same as double for many compilers). If in cpmdefinitions.h the macro CPM\_MP is defined, R has the meaning of mpreal as defined by the wrapper library MPFRC++ to the multiple precision library mpfr. See comments to CPM\_MP in file cpmdefinitionswrc.h.  
End of 3.1

There is a namespace CpmRootX for the 'less essential essentials':

4. There are classes CpmRootX::B, CpmRootX::Z1, CpmRootX::R1, for Boolean values, integer, and real numbers, which may replace the non-class types bool, Z, and R in situations where class functionality, such as automatic initialization, is indispensable.

#### 4.1 Details (skip on first reading ?):

Namespace CpmRoot in mainly layed out in file cpmnumbers.h and CpmRootX in cpmtypes.h. The vigilant reader may argue that we have not available the declarations of CpmRoot, since we have no #include <cpmnumbers.h> seen so far. Actually, it is there since file cpmbas.h is the following collection of include directives  
#ifndef CPM\_BAS\_H\_  
#define CPM\_BAS\_H\_

```

#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

```

Also here we don't see

```

#include <cpmnumbers.h>

```

but inspecting `cpmtypes.h`, we find the commented directive

```

#include <cpmsystem.h> // includes cpmwords.h and
// thus also cpmnumbers.h

```

which shows the steps which finally include `cpmnumbers.h`. Actually the directory `cpm/cpm0/include` contains 32 header files. This set of files (as any set of C++ header files) is a directed graph in a natural manner: There is an edge leading from `file1` to `file2` iff `file2` contains the directive `#include <file1>`.

The files that can be reached from `file1` along a path of that graph depend on `file1` and every translation unit which includes such a `file1`-dependent header has to be re-compiled upon some change in `file1`. So, any system capable of generating proper make files has to do this dependency analysis and thus has to work with this `dependency graph`. I made a tool consisting of a Ruby program for file analysis and a C++ program for graph analysis and representation which creates a pictorial representation of this dependency graph of an arbitrary collection of C/C++ header files. This tool turned out to be very useful.

It is an obvious logical requirement that the dependency graph is free of cycles. It is a non-trivial task to organize the header dependencies in a way that each translation unit gets the declarations which it needs to know by inclusion of only a few header files. For instance, `cpmword.cpp` and `cpmtypes.cpp` need only `cpmtypes.h`; `cpmc.cpp` needs `cpmc.h` and `cpmtypes.h`. The present header file hierarchy is the result of many simplifying re-organizations. However, many attempts of a simplification failed since they entailed unacceptable complications elsewhere. I'm not aware of tools which would automatize relevant parts of this organization process.

End of 4.1

```

*****/
using namespace CpmArrays;
/*****
5. The general-purpose array in C++ is the template class CpmArrays::V
Most constructors set the first valid index of a non-void V is 1 and
not 0 as for std::vector. There is, however, a member function which
shifts all indexes and so allows indexing to start with 0. This allows
identical code to be used for these two types of arrays.

```

5.1 Details on C++ namespaces (skip on first reading?):

The C++ class system introduces many namespaces. Their belonging

together under the Cpm banner is not expressed by making them sub-namespaces of a single Cpm-namespace. Instead, their names all start with 'Cpm' and continue with a capital letter.

If the name contains a digit, this is 2 or 3 and refers to the space dimension under consideration. So the namespace CpmDim2 contains the 2D ('flatland') analogs of the geometric classes defined in namespace CpmDim3.

The namespaces which are declared in the present scope by including cpmbas.h are

CpmRoot, CpmRootX, CpmSystem, CpmArrays, CpmFunctions, CpmGeo, CpmGraphics, CpmMPI, CpmStd, CpmTests, CpmTime.

End of 5.1

```

*****/

void info();
Z tutorial1(Z,Z);
    // Declaration of two functions, the definition of which will
    // determine the functionality of the program.

int main(int argc, char* argv[])
    // Traditional main function with types of arguments and return value
    // according to C/C++. Not all compilers accept using Z instead of
    // int here.
{
    Z m=200, n=20000; // default values to be used if no input from the
        // command line can be found
        // For Z m=400, n=40000;
        // under cygwin the function perf_V runs through but perf_v causes
        // stackdump without leaving any message. Visual C++ has no problem
        // in this case.
    V<Word> arg=comLine(argc,argv); // CpmRoot::Word is similar to
        // std::string. CpmArrays::V is the C+- array type. There are two
        // 'schools' about indexing arrays. The 'C school' lets valid
        // indexes start with 0 and the 'Fortran school' which lets valid
        // valid indexes start with 1. Till September 2010 C+- had a
        // 'C school'-array V1 (1 for 'lean') and a 'Fortran school'-array
        // V , where the implementation of V was such that any instance of
        // V had a data member of type V1. The type V1 was used mostly
        // internally in situation where efficiency was considered to be
        // of utmost importance. Finally it became clear that the decision
        // to have these two types of arrays was a mistake. It creates a
        // permanent uncertainty for the programmer whether he should use
        // the default type or should strive for utmost efficiency by
        // using the lean version. Now we have only a single array type V
        // for which the valid indexes may form any contiguous set of
        // integers. This index range is set to start with 1 in most
        // of the available constructors but can be very conveniently reset
        // to any other start index such as 0.
        // CpmArrays::comLine is a function which transforms the
        // traditional argument of main into the more functional data type
        // of C+- . The function name 'comLine' results from
        // the d e s c r i p t i v e f u l l n a m e (DFN)

```

```

// 'command line' by a simple deterministic rule: The rule is
// to leave unchanged all words with up to four letters and shorten
// all longer words to 3 by taking the first two letters as they are
// and take the first of the following consonants as the third
// letter. Thus 'oops' gets converted to 'oop'. If there is no
// such consonant, the vowel at place 3 has to be taken: 'hooiii'
// gets converted to 'hoo'. In a chain of words, the contracted
// components get concatenated in a style which is evident from
// the example that 'descriptive full name' goes to
// 'desFullName'. In most cases the names come out quite nice, but
// there are exceptions (not to the rules!): I consider it ugly to
// see 'field' abbreviated to 'fil' (notice that 'file', as having
// only four letters, remains 'file')
// The simplicity of the rule allows us to use the the function name
// fluently whenever we remember the full name correctly. Of course,
// the descriptive full name of a function has to be stated in the
// declaration of this function. In our case the declaration is in
// file cpmv.h:
// inline V<Word> comLine(int argc, char* argv[])
//     //: command line
// The colon hints at the fact that here a descriptive full name
// is being communicated.
if (arg.valInd(2)){ // valInd's DFN is 'valid index'
// C+- arrays judge the validity of an index directly without
// a need to ask for 'size' and making a comparison.
Word w2=arg[2];
if (w2=="?"){ // Thus a question mark as the second entry of the
// command line triggers a call to function info.
    info();
    return 0;
}
else{
    m=w2.toZ(); // Converting Word to Z, for getting a control
// parameter. There are also conversions toR(), toBool(),
// and toStr()
    if (arg.valInd(3)) n=arg[3].toZ();
}
}
return tutorial1(m,n);
}

using namespace std;

C cz(Z i) // an ad-hoc function Z --> C
// CpmRoot::C is the class of complex numbers
{
    R ir=i;
    return C(ir/(1+ir*ir),ir*cpmpi); // there is a CpmRoot::Pi = 3.14159...
// calling constructor C(R,R)
}

// Now we define the performance measuring functions perf_v and perf_V.

```

```

// These functions have to execute identical code for two different
// vector templates: std::vector in perf_v and Cpmrrays::V in perf_V.
// Since we want to iterate these templates, it would not be easy to
// achieve this code doubling by defining a template function.
// Such a template would need at least two template arguments and would
// enforce unnatural expressions.

// This function is designed as to maximize the benefit from reference
// counting and to report the time spent on copy-construction and
// assignment (not of the tedious verification part of the function).

#define Vec vector
V<R> perf_v(Z m, Z n)
{
    Z mL=1;
/*
    This line, together with the following
    Word loc(...);
    CPM_MA
    CPM_MZ
    constitute a convenient idiom for signaling
    entry to and exit from a function block to the log file
    cpmcerr.txt. This writing to the log file takes place only if
    mL is larger or equal to the static data member
    CpmSystem::Message::verbose. This bulky quantity has a
    macro alias 'cpmverbose' (i.e. we have, in file cpmsystem.h
        #define cpmverbose      CpmSystem::Message::verbose
    ) and it can be set by a statement like
        cpmverbose=10;
    Its initial value is 2.
    Soon we will encounter macros cpmtime, cpmwait, cpmdebug.
    See the remarks on the macro namespace in function perf_V.
*/
    Word loc("perf_v(Z,Z)"); // messages use this as name of the function
    CPM_MA // defined in cpmmacros.h, assumes that 'mL' and 'loc' are
        // defined
    R tl=cpmtime(); /* time of function call in seconds from some
        system-defined 'point-zero in time'.
        cpmtime is a short name for function CpmSystem::time defined
        in file cpmsystem.h as follows:
            #define cpmtime      CpmSystem::time
    */
    Z i,j;
    Vec< Vec<C> > vi,vf; // we use arrays, the components of which
        // are potentially large objects. This is the strong point of
        // C+- arrays and a weak one of std::vector and of std::valarray.
    {
        Vec<C> v1(m);
        for (i=0;i<m;++i) v1[i]=cz(i);
            // The preferred form of this statement in C+- is
            // for (i=v1.b();i<=v1.e();++i) v1[i]=cz(i);
            // Written in this form, it is also correct if v1 is of type V<C>,

```

```

    // the C++ array for which indexing starts with 1 instead of 0.

    Vec< Vec<C> > w(n,v1);
    // If we use valarray for Vec, one has to write w(v1,n) instead.
    // This is a inconvenience of the C++ standard library which is
    // due to their multi-source origin.
    Vec< Vec<C> > temp1=w; // copy constructor
    Vec< Vec<C> > temp2=temp1; // assignment
    Vec< Vec<C> > temp3=temp2; // assignment
    temp2=Vec< Vec<C> >(0);
    vi=w;
    vf=temp3;
}
R t2=cpmtime();
R t12=t2-t1; // time needed for copy and assignment
R err=0,sum=0;
for (i=0;i<n;++i){
    for (j=0;j<m;++j){ // sic!
        err+=(vi[i][j]-vf[i][j]).abs();
        sum+=(vf[i][j]).abs();
    }
} // making sure that copying and assignment worked correctly
V<R> res("",t12,err,sum);
/*
Convenient l i s t c o n s t r u c t o r .
The maximum number of entries is 10. The first entry ''''
is necessary for proper function call resolution.
This is much more convenient than the equivalent
V<R> res(3);
res[1]=t12; res[2]=err; res[3]=sum;
In C++ all concatenation operations have a '&' in their name. So it is
natural that the previous definition can also be writtern as:
V<R> res;
res&=t12&=err&=sum;
which is nearly as compact as the list constructor and has the
advantage to be lot limited in length.
*/
CPM_MZ
return res;
}
#undef Vec
#define Vec V
V<R> perf_V(Z m, Z n)
{
    Z mL=1;
    Word loc("perf_V(Z,Z)");
    // Here goes code which is identical to the one of the
    // previous function perf_v. In actual C++ code this
    // is never implemented by copying some piece of code
    // but always by defining the macro CPM_SC (SC for 'shared
    // code' (and undefining it after usage). C++ claims a
    // m a c r o n a m e s p a c e : all macros beginning

```

```

// with the four characters 'CPM_' and the three characters
// 'cpm'. The upper case macros stand in most cases for a few
// lines of code and should be recognized by the user as not
// being a simple identifier. They are thus written without a
// semi-colon at the end in code since the semi-colon is part
// of the macro's definition.
// The lower-case macros, by contrast, are in most cases
// #define-implemented abbreviations of quantities for which
// a normal name is available (e.g. CpmSystem::Message::verbose
// for cpmverbose). Whether these are variables or functions, they
// are used as normal identifiers in code and the user needs not to
// be aware of their macro nature. Hence, these macros are thus
// written
// in code with a semi-colon at the end. (The semi-colon is not part
// of the macro's definition.)
//
CPM_MA // macro not followed by a semi-colon
R t1=cpmtime(); // macro followed by a semi-colon
Z i,j;
Vec< Vec<C> > vi, vf;
{
  Vec<C> v1(m);
  v1.b_(0); // lets indexing of v1 start with 0
  for (i=0;i<m;++i) v1[i]=cz(i);
  Vec< Vec<C> > w(n,v1);
  w.b_(0); // lets indexing of w start with 0
  Vec< Vec<C> > temp1=w;
  Vec< Vec<C> > temp2=temp1;
  Vec< Vec<C> > temp3=temp2;
  temp2=Vec< Vec<C> >(0);
  vi=w;
  vf=temp3;
}
R t2=cpmtime();
R t12=t2-t1;
R err=0,sum=0;
for (i=0;i<n;++i){
  for (j=0;j<m;++j){
    err+=(vi[i][j]-vf[i][j]).abs();
    sum+=(vf[i][j]).abs();
  }
}
CPM_MZ
return V<R>("",t12,err,sum);
}
#undef Vec

Z tutorial1(Z m, Z n)
{
  Z mL=1;
  Word loc("tutorial1(Z,Z)");
  CPM_MA

```

```

V<R> res_V=perf_V(m,n), res_v=perf_v(m,n);
R t_V=res_V[1], t_v=res_v[1];
R err_V=res_V[2], err_v=res_v[2];
R s_V=res_V[3], s_v=res_v[3];
R fac=t_v*cpminv(t_V);
cout<<"t_v="<<t_v<<" t_V="<<t_V<<" fac="<<fac<<
    " err_v="<<err_v<<" err_V="<<err_V<<endl;
    // output of the result to the console
cpmdebug(m);
cpmdebug(n);
cpmdebug(t_V);
cpmdebug(t_v);
cpmdebug(fac);
    // convenient documentation of the main result on
    // the auto-generated log file cpmcerr.txt.
    // Defined in file cpmtypes.h as
// #define cpmdebug(X) cpmcerr<<endl<<"C+- debug: "<<#X "="<< X <<endl
// Here are the first lines of this file:
//
// 2008-06-10 16:26:20.0 Z
// counter = 1
// Message: CpmSystem::Message::ini(void): files cpmcerr.txt and
// cpmdata.txt successfully created,
// rank=1, size=1
// 0.062 s after program start
// ...
// After some other message we read:
//
// C+- debug: m=400
//
// C+- debug: n=40000
//
// C+- debug: t_V=0.032
//
// C+- debug: t_v=3.469
// Writing 'cpmnote' instead of 'cpmdebug' places similar lines
// in the auto-generated data file cpmdata.txt.
CPM_MZ
return 1;
}

void info(){
    cout<<endl<<"takes zero, one, or two integer argument";
    cout<<endl<<"compares the performance of std::vector and CpmArays::V";
    cout<<endl<<"in copy construction and assignment operations";
}
// end of tut1.cpp

```

## 1.2 Discussion

For the default data  $m = 400$  and  $n = 40000$  I got on a 2.4 GHz off-the-shelf laptop PC the computation time  $t_l = 0.032$  s and  $t_v = 3.469$  s so that the speed advantage for the C++ arrays results as a factor of 108 in this particular case. One should take into account, however, that this number is not accurate, since the function `cpmtime` is not good at properly measuring short times, say below one second. Unfortunately, using much larger numbers (e.g.  $m = n = 7000$ ) does not work, since `std::vector` than stops on throwing a `std::bad_alloc` exception (using MS Visual C++ 2008 express edition, and also with the Dev C++ 4.9.9.2 system). Replacing `std::vector` by `CpmArrays::V` — which cannot create an interesting result — makes the program finish regularly. Replacing it with `std::valarray` lets the program fail in the same way as with `std::vector`. All the tested variants work well with  $m = n = 5000$ . So it is not true that the standardized tools are the most powerful ones in all situations. There are, however, no mysteries involved:  $m = n = 10000$  lets fail `CpmArrays::V` in the same way as `std::vector` failed before. The difference is simply that, due to the reference counting mechanism, class `CpmArrays::V` avoids some allocations that `std::vector` has to perform (unless an optimizing compiler is clever enough to avoid them too) and thus reaches the limits a bit later.

All memory allocation for C++ objects is done by calling a constructor, so that C++ guarantees that a matching destructor call frees the memory in time. These internal allocation, de-allocation processes are done with the `new`, `delete` facility, as provided by the compiler. This lets debugging facilities make their choices according to their needs. Since we need no graphics libraries for this program we need to have the line

```
#define CPM_NOGRAPHICS
```

in file `cpmsystemdependencies.h`. By experimentally disabling (not recommended for permanent use) the lines

```
#define CPM_USECOUNT
```

```
#define CPM_RANGE_CHECK
```

in file `cpmdefinitions.h` we disable respectively the reference counting mechanism and the range check on array indexes. Of course, changes here change the code and thus require partial recompilation.

## 1.3 Options for compilation

Each program which uses C++ classes (e.g. by having `cpmbas.h` among its header files and `cpmbas.cpp` among its source files) has to hold in one of its include directories header files named `cpmdefinitions.h` and `cpmsystemdependencies.h` for controlling compilation. The various control options are explained in the following fully commented versions of these files. Most C++ code depends on the first of these files so that changes there require recompilation of virtually all sources. Of particular importance is the recently added possibility to employ multiple precision arithmetic which is explained at entry `CPM_MP`.

```

//? cpmdefinitionswrc.h
//? C+- by Ulrich Mutze. Status of work 2009-08-03.
//? Copyright (c) 2009 Ulrich Mutze, www.ulrichmutze.de
//? All rights reserved

#ifndef CPM_DEFINITIONS_H_
#define CPM_DEFINITIONS_H_
/*****
    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)
    - - - - -
*****/
// This file will never be included by a C+- program since its name is
// cpmdefinitionswrc.h instead of cpmdefinitions.h. Upon renaming, it can
// function as cpmdefinitions.h.
// The name appendix 'wrc' stands for 'with reference comments'.
// We don't use the more readable form cpmdefinitions_wrc.h in order
// not to disturb pdflatex-based Rugsy script listing generation.
// The role of this file is to hold valid and complete explanations of the
// macros which can be set in any file cpmdefinitions.h.
// A file with similar scope is cpmsystemdependencieswrc.h.

#define CPM_NAMEOF
    // If this is defined, the C+- naming scheme is used instead of
    // C++ typeid. Recommended since for template types, native
    // typeid creates inconveniently long names.

// #define CPM_LONG
    // If this is defined, the types Z and N are long int and unsigned
    // long int, otherwise simply int and unsigned int.
    // If this is defined and CPM_MP is not defined the type R is
    // long double (which, unfortunately, is not always longer than
    // double).
    // Was not found to make much of a difference.

#define CPM_USECOUNT
    // If this is defined, all arrays use reference counting and
    // copy on write.
    // Recommended, increases efficiency.

#define CPM_RANGE_CHECK
    // If this is defined, all indexed access to arrays is index-
    // checked.
    // Recommended, increases safety.

// #define CPM_USE_MPI
    // MPI = message passing interface (for parallel computing).
    // See cpmmmpi.h and cpmmmpi.cpp.
    // This functionality was virtually unused for 7 years and was
    // reanimated in 2009-05-27. After dislocation of a single

```

```

// b(road)cast statement in the main constructor of RecordHandler
// a 2D example with gravitating and elastic repelling particles
// worked fine and created a movie on which all particles were
// to be seen in a common field of view. The command for
// running was:
//   mpirun -localonly -np 4 palaMS.exe
// However, the C++ system favors a style of programming which
// abstracts from technical aspects of the computing environment.
// So questions of how to distribute a computational task over
// various processors or various computers are not considered
// of interest here. They were of interest in the industrial
// toning modeling project which contributed significantly to
// the experiences on which the C++ system is based.

// #define CPM_MP 32
// This macro enables usage of multiple precision
// mathematics and assumes that the libraries gmp and
// mpfr are available for linking (and, of course, their headers
// for inclusion). Also the files mpreal.cpp, mpreal.h (by Pavel
// Holoborodko) need to be available for compilation and for
// inclusion respectively.
// The number of decimal places may also be set in cpmconfig.ini.
// This line has to be invalid (by preceding it with '///', i.e. by
// commentarizing it) if normal IEEE 754 arithmetic is to be used
// or if the libraries gmp and mpfr are missing.
// Then the input in cpmconfig.ini concerning numerical precision
// does not become effective.

// #define CPM_FIGURE_FOR_PUBLICATION_1
// Selects particular code in cpmpardynsys.h, which is easily found
// by searching for 'CPM_FIGURE_FOR_PUBLICATION_1'.
// Ad hoc solution.
// No effect if file cpmpardynsys.h is not used.

#endif

```

Only a few files depend on the following file so that changes there require only minor recompilation activities.

```

//? cpmsystemdependencieswrc.h
//? C++ by Ulrich Mutze. Status of work 2011-03-30.
//? Copyright (c) 2011 Ulrich Mutze, www.ulrichmutze.de
//? All rights reserved

#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. on the access to a display or a file system.
These defines may also modify the way how 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.

```

*****/
// See file cpmdefinitionswrc.h for the meaning of the 'wrc'-appendix.

#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
    // comfortably set in cpmconfig.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_STEREOLEFT_R 255
#define CPM_STEREOLEFT_G 0
#define CPM_STEREOLEFT_B 0

#define CPM_STEREORIGHT_R 0
#define CPM_STEREORIGHT_G 255
#define CPM_STEREORIGHT_B 255
    // Optional input of color values for generation of anaglyph images.
    // If no input from here, code in cpmimagingtools.cpp comes in
    // action. If there is input from cpmconfig.ini that input overrides

```

```
// all previously fixed values. See cpmconfigwrc.ini

#define CPM_RUN_ID_LENGTH 4
// Number of digits of the run ID.
// Notice that the run ID is generated from the point in time of
// program start and is unrelated to the runName_ data member of class
// CpmApplication::IniFileBasedApplication, which can be read at
// run time from its ini-file. Only if runName_ is asked to be
// be 'auto', the run ID will be used to replace runName_.
// CpmTime::Greg::getCode(Z) is the function which determines 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 cpmcerr and cpmdata.
// This is recommended for important program runs since
// cpmcerr conserves all data which characterize a program
// run. However, in a mere experimental phase of program
// development it could be useful not to let each program
// run create log files with new names with the consequence
// of overloading the working directory. In such cases one
// not define this macro. Even for serious runs this is
// a viable option since one may then, after the run,
// rename cpmcerr.txt by appending then a more indicative
// run name (preferable the one which was appended to the
// other documentation files). Since 2011-03-30 there is
// the possibility to set CpmApplication::doc_=true which then
// makes a copy of cpmcerr with the run ID appended to the name.
// The new function CpmSystem::Message::augRunId(Word const&)
// allows to augment the numerical run ID by 'speaking' text.

// #define CPM_NOLOG
// If defined, no cpmcerr and cpmdata will be created.
// This may be an option if either these log-files would
// be very large, or the program has been very well tested
// and understood. Normally having cpmcerr.txt generated is
// very useful.

// #define CPM_NOGRAPHICS
// If defined, no graphical content will be shown on screen
// and no graphics libraries (OpenGL and GLUT) need to be linked.
// Nevertheless ppm-images (of arbitrary size) may be created.

#if !defined(CPM_NOGRAPHICS)
```

```

// otherwise we don't need graphics libraries
#ifdef WIN32
    #if defined(MINGW)
        #include <c:/opt/Dev-Cpp/include/GL/openglut.h>
        // on my system
    #else
        #include <glut.h>
    #endif
#else
    // 'Cygwin' or 'Linux'
    #include <GL/glut.h>
#endif
#endif

#endif

```

## 2 Defining main with the basic C++ application framework

The next example generates graphics to visualize physical processes and properties of approximation methods. It studies two integrating methods for a one-dimensional mechanical system, which is a non-harmonic oscillator for which an exact solution is known so that method errors are easy to characterize. The system is discussed in detail in Section 4 of

[www.ulrichmutze.de/articles/99-271.pdf](http://www.ulrichmutze.de/articles/99-271.pdf)

and the notion of *interaction picture* as a means to study errors of numerical evolution schemes is explained in Section 7 of

[www.ulrichmutze.de/articles/03-491.pdf](http://www.ulrichmutze.de/articles/03-491.pdf)

It is a C++ application in the narrower sense that it does not only use C++ classes in its implementation but also uses the C++ application framework. This means that among the source files of the projects there is the file `cpmapplication.cpp` which initializes graphics classes (including fonts) and allows a user program to create graphical representations rather conveniently. This will be evident from the code to follow.

### 2.1 Code example `tut2.cpp`

```

///tut2.cpp
///C++ by Ulrich Mutze. Status of work 2012-01-12.
///Copyright (c) 2012 Ulrich Mutze, www.ulrichmutze.de
///All rights reserved

/*****
    Studying the Kepler oscillator
    *****/
#include <cpmbas1.h>
    // Actually, this is a collection of header files.
#include <cpmvectormore.h>

```

```
using namespace CpmRoot;
using namespace CpmRootX;
using namespace CpmSystem;
using namespace CpmArrays;
using namespace CpmGraphics;

using CpmLinAlg::R2;
using CpmGeo::Iv;
using CpmAlgorithms::solKepEqu;
using CpmAlgorithms::CyclicAlarm;
using CpmImaging::YELLOW;
using CpmImaging::GREEN;
using CpmImaging::darkSky;
using CpmImaging::twiSky;
using CpmImaging::morSky;

namespace CpmKepler{

bool stepRed_=false; // step reduction
    // Has influence only for ALF
R facALF=1.0;
    // An experimental method parameter.
    // Normal value is 1.0

class KepOsc{ // 'Kepler oscillator'
// My name for the system defined as the radial part of the Kepler
// motion arround a fixed central mass. Physical units are used in which
// the moving mass, the gravitational potential at unit distance, and the
// angular momentum have value 1.
// see www.ulrichmutze.de/articles/99-271.pdf, Section 4

// Methodologic remarks:
// The primary aim here is to compare various integrators for the
// equation of motion for the physical system mentioned above.
// There are two strategies for implementing this aim:
// 1. Define for each integration method a separate class.
//     This is the normal way to be followed according to the pure doctrine
//     of object oriented programming. It is also very natural since the
//     various integration methods to be considered need different state
//     data for implementing an integration step as a mapping of the
//     state space into itself.
// 2. Define each integration method as a separate member function
//     within a single class. This requires the state space to contain
//     data which are needed for implementing e.g. the leap-frog integrator
//     and are not needed e.g. for the asynchronous leap-frog integrator.
//     If data belong to a type for which the default constructor is
//     computationally inexpensive such partially relevant data seem to
//     be acceptable and the present class implementation follows this
//     path.

// data
```

```

// 1. common data
R t_, x_, v_, dt_; // time, position, velocity, time step
Z met_; // method selector for function step_
// 2. additional data for implementing stepALF_(R dt)
R1 w_, a_;
// 3. additional data for implementing stepLF_(R dt)
R tp_, xp_, vp_; // p for previous

// static functions
static R f(R const& x){R xi=cpmInv(x); return (xi-1)*xi*xi;}
// force law
// member functions
void init_(Z met, R eps, R nPerRev, R E);
// assumes 0<=eps<1, E arbitrary
// initializes the common data
void init2_(){ w_=v_; a_=f(x_);}
// does the additional initializations for met_=2
void init3_()
// does the additional initializations for met_=3
// simplified version, we need not be more accurate than second
// order
{
R h= -dt_;
if (stepRed_) h*=0.5;
tp_=t_+h;
R a=f(x_);
R dv=a*h*0.5;
xp_=x_+(v_+dv)*h; // here xp_ is previous x_
vp_=v_+(a+f(xp_))*h*0.5;
}

void initAux_(){
if (met_==2) init2_();
if (met_==3) init3_();
}
// does the additional initializations for met_=2 and met_=3

void stepDMI_(R dt); // direct midpoint integrator (2nd order)
// needs only the common data
void stepRK_(R dt); // Runge-Kutta (2nd order)
// needs only the common data
void stepALF_(R dt); // asynchronous leap-frog (2nd order)
// needs data w_, a_ for implementation
// See 'Fortsetzung von UM 10.9.08' in my notebook UM 2008a
void stepLF_(R dt); // leap-frog (2nd order)
// needs tp_, xp_, vp_
public:
Word nameOf()const{ return "KepOsc, mehod="&cpm(met_);}
KepOsc(){init_(0,0,32,0);}
explicit KepOsc(Z met, R eps, R nPerRev=64, R E=0)
{init_(met,eps,nPerRev,E);}
R t()const{ return t_;}

```

```

R dt()const{ return dt_;}
R x()const{ return x_;}
R v()const{ return v_;}
R eKin()const{ return v_*v_*0.5;}
    // kinetic energy
R ePot()const{ R xi=cpminv(x_); return xi*(xi*0.5-1);}
    // potential energy
R eTot()const{ return eKin()+ePot();}
    //: e total
R force()const{ return f(x_);}
    //: force
    // The superficially correct name 'for' is not correct since it is
    // a C++ key word.
R_Vector orbDes()const;
    //: orbit descriptors
    // Returns a list of the usual orbit elements
    // a: major semi-axis, eps: numerical excentricity,
    // n: mean motion, E: excentric anomaly, M: mean anomaly,
    // tP: orbital period (revolution time)
void rev_(){ v_=-v_;}
    //: reversed
    // Notice name ending in '_' since it is a non-constant function
void extEvl_(R t, R acc=1e-8);
    //: exact evolution

void step_()
    //: step
    // The actual step algorithm depends on method parameters
{
    if (met_==0) stepRK_(dt_);
    else if (met_==1) stepDMI_(dt_);
    else if (met_==2){
        if (stepRed_){
            R tau=dt_*0.5;
            stepALF_(tau);
            stepALF_(tau);
        }
        else stepALF_(dt_);
    }
    else if (met_==3){
        if (stepRed_){
            R tau=dt_*0.5;
            stepLF_(tau);
            stepLF_(tau);
        }
        else stepLF_(dt_);
    }
    else cpmerror("bad value of KepOsc::met_");
}

void shf_(R dx, R dv)
    //: shift

```

```

{
    x_+=dx; v_+=dv;
    initAux_();
}

X3<R,Iv,Iv> orbSize()const;
    //: orbit size
    // intervals which contain the t_-range, x_-range , the y_range
    // for a full revolution

Color col()const
    //: color for visualization
{
    if (met_==0) return WHITE; // RK
    else if (met_==1) return YELLOW; // DMI
    else if (met_==2) return GREEN; // ALF
    else if (met_==3) return RED; // LF
    else cpmerror("bad value of KepOsc::met_ in KepOsc::col()");
}

void mark(Graph& g)const{ g.mark(R2(x_,v_),col());}
    //: mark

R2 phsPos()const{ return R2(x_,v_);}
    //: phase (space) position

R per()const{ return orbDes()[6];}
    //:period

Word met()const
{
    if (met_==0) return "RK2";
    if (met_==1) return "DMI";
    if (met_==2) return "ALF";
    if (met_==3) return "LF";
    else return "unvalid method";
}
};

void KepOsc::init_(Z met, R eps, R nPerRev, R E)
// initializing x_ anv v_ from numerical excentricity and excentric
// anomaly. Further, initialization of the dependent quantities.
{
    Word loc("KepOsc::init_(R,R)");
    cpmassert(eps>=0,loc);
    cpmassert(eps<1,loc);
    met_=met;
    t_=0;
    R ai=(1-eps)*(1+eps);
    R a=1./ai;
    x_=a*(1-eps*cpmcos(E));
    v_=eps*cpmsqrt(a)*cpmsin(E)/x_;
}

```

```

    R tP=cpm2pi*cpmpow(a,1.5);
    dt_=tP/nPerRev;
    initAux_();
}

R_Vector KepOsc::orbDes() const
{
    Word loc("KepOsc::orbDes()");
    Z mL=2;
    CPM_MA
    R e=eTot();
    if (e>=0){
        cpmcerr<<"We have e>=0, un-bound state. x_="<<x_<<" v_="<<v_<<endl;
        cpmdebug(ePot());
        cpmdebug(eKin());
    }
    cpmassert(e<0,loc); // we need a bound state
        // then also x>0.5
    R a=-0.5*cpminv(e);
    R eps=cpmsqrt(1.-1./a);
    R n=cpmpow(a,-1.5);
    R tP=cpm2pi/n;
    R z1=1-x_/a;
    R z2=x_*v_/cpmsqrt(a);
    C z(z1,z2);
    R E=z.arg();
    R M=E-eps*cpmsin(E);
    CPM_MZ
    return R_Vector("",a,eps,n,E,M,tP);
}

void KepOsc::extEvl_(R t, R acc)
// exact step with t arbitrarily large, based on the
// famous transcendental Kepler equation
{
    Word loc("KepOsc::extEvl_(R,R)");
    Z mL=2;
    CPM_MA
    R_Vector y=orbDes();
    R a=y[1],eps=y[2],n=y[3],M=y[5];
    M+=n*t;
    R E=solKepEqu(M,eps,acc);
    x_=a*(1-eps*cpmcos(E));
    cpmassert(x_>=0.5,loc); // for safty, not needed
    v_=eps*a*a*n*cpmsin(E)/x_;
    t_+=t;
    initAux_();
    CPM_MZ
}

void KepOsc::stepDMI_(R dt)
// nice symmetry, f evaluated only once

```

```
{
    R tau=dt*0.5;
    t_+=tau;
    x_+=(v_*tau);
    v_+=f(x_)*dt;
    x_+=(v_*tau);
    t_+=tau;
}

void KepOsc::stepRK_(R dt)
// not so nice, involves two evaluations of f
{
    R tau=dt*0.5;
    R a=f(x_);
    R am=f(x_+v_*tau);
    R dv=am*dt;
    R dx=(v_+a*tau)*dt;
    x_+=dx;
    v_+=dv;
    t_+=dt;
}

void KepOsc::stepALF_(R dt)
// asynchronous leap-frog method with a factor facALF that normally
// should be 1.
{
    R tau=dt*0.5;
    R fac=2*facALF;
    t_ += tau;
    x_ += tau*w_;
    v_ += tau*a_;

    w_ += fac*(v_-w_);
    a_ += fac*(f(x_)-a_);

    v_ += tau*a_;
    x_ += tau*w_;
    t_ += tau;
}

void KepOsc::stepLF_(R dt)
{
    R dt2=dt*2;
    R tn=tp_+dt2;
    R vn=vp_+f(x_)*dt2;
    R xn=xp_+v_*dt2;
    tp_=t_;
    vp_=v_;
    xp_=x_;
    t_=tn;
    v_=vn;
    x_=xn;
}
```

```

}

X3<R, Iv, Iv> KepOsc::orbSize() const
{
    Z mL=2;
    Word loc("KepOsc::orbSize()");
    CPM_MA
    R_Vector y=orbDes();
    R a=y[1], eps=y[2], n=y[3], tP=y[6];
    R xMin=a*(1-eps);
    R xMax=a*(1+eps);
    R vMax=eps*a*a*n;
    CPM_MZ
    return X3<R, Iv, Iv>(tP, Iv(xMin, xMax), Iv(-vMax, vMax));
}

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

void cpmplot(V<V<R2>> > const& dat, Word const& fileName)
{
    Word loc("cpmplot (V<V<R2>>, Word)");
    Z mL=2;
    CPM_MA
    if (fileName.isVoid()){
        CPM_MZ
        return;
    }
    Z i, j, md=dat.dim();
    V<Z> vd(md);
    for (i=1; i<=md; ++i){
        vd[i]=dat[i].dim();
    }
    S<Z> sd(vd);
    cpmassert(sd.car()==1, loc);
    Z nd=sd[1];
    OFileStream ofs(fileName);

    for (i=1; i<=nd; ++i){
        ofs()<<endl;
        for (j=1; j<=md; ++j){
            R2 y=dat[j][i];
            ofs()<<y[1]<<" "<<y[2]<<" ";
        }
    }
    CPM_MZ
}

void appl(void) // orbit in phase space (v = momentum since m=1)
// simultaneously for a selected list of integration methods
{
    Word loc("appl");
    Color cb=darkSky; // night sky blue. Each task appi

```

```

// has here its characteristic background color
RecordHandler rch("appl.ini");
    // This says that we expect data input to be available in file
    // appl.ini in a directory which is determined by entries in
    // cpmssystemdependencies.h and cpmconfig.ini.
Word sec="data"; // names 'rch' and 'sec' are to be employed
    // for the macro cpmrh to work
R eps=0.25, fac=1.2, nPerRev=16, tWaitFinal=1;
Z nTot=1000, displayPeriod=1;
Word imageName, plotDataName;
V<Z> methods;
// getting quantities from section 'data' of file appl.ini
cpmrh(eps); // rh: read handler
cpmrh(nPerRev);
cpmrh(nTot);
cpmrhf(stepRed_);
cpmrh(fac);
cpmrh(tWaitFinal);
cpmrh(imageName);
cpmrhf(plotDataName);
cpmrh(displayPeriod);
    // now the program has all its data
cpmrh(methods);
cpmrhf(facALF);
S<Z> met(methods); // eliminates duplicates
Z nm=met.dim();
V<KepOsc> vk(nm);
for (Z i=1; i<=nm; ++i) {
    vk[i]=KepOsc(met[i], eps, nPerRev);
}
X3<R, Iv, Iv> os=vk[1].orbSize(); // cartesian product templates
    // are defined from X2 to X8
Iv ivx=os.c2(); // Iv = interval is a valuable type
cpmdebug(ivx);
Iv ivy=os.c3();
cpmdebug(ivy);
ivx*=fac; // changing the size of an interval, while holding the center
    // fixed
ivy*=fac; // same factor for the other direction
V<V<R2> > vvr(nm, V<R2>(nTot)); // memory for all phase space positions
    // during run
Graph g(cb); // basic graphical 'window to the world'
g.setXY(ivx, ivy);
    // this associates physical sizes (in space) with the
    // 'pixeled' rectangle g
R nInv=1./nTot; // for use in cpmprogress
CyclicAlarm cy(displayPeriod); // device for triggering periodic
    // actions
for (Z i=1; i<=nTot; ++i) {
    for (Z j=vk.b(); j<=vk.e(); ++j) {
        vk[j].mark(g); // 'mark' acts on memory only
        R2 pj=vk[j].phsPos();
    }
}

```

```

        if (i==1) cpmdebug(pj);
        vvr[j][i]=pj;
        vk[j].step_();
    }
    if (cy()||i==nTot){ // cy() becomes 'true' periodically
        g.vis(); // transfers the whole memory frame
                // to screen. This expensive action should probably not be
                // orderd for each integration step. Object cy allows
                // to achieve this.
        cpmprogress("loop",i*nInv); // progress indicator on
                // pane 1 of he status bar
    }
}
g.wrt(imageName); // writes the final frame to a ppm image file
                // if imageName is not the void word.
cpmplot(vvr,plotDataName); // does nothing if the name is void
cpmwait(tWaitFinal,2); // allows us to inspect the last frame
                // for the time (in seconds) put in here. The second argument
                // says that on pane 2 of the status bar there will be a
                // 'countdown' indication of the time left for inspection.
}

void app2(void) // method error of the two integration methods
// represented by means of interaction picture dynamics. For
// applying this terminology, we interpret the dynamics as defined by
// the numerical methods as resulting from the exact Kepler dynamics
// by 'interaction'. This interaction picture motion is much slower for
// the more accurate direct midpoint method. The Runge Kutta motion
// finally explodes and reaches non-bound states for which the exact
// evolution back is no longer defined (by the method KepOsc::extEvl_).
// Therefore the option is implemented to deal only with the stable
// direct midpoint method.
{
    Word loc("app2");
    Color cb=twiSky; // background
    RecordHandler rch("app2.ini");
    Word sec="data";
    R eps=0.25, nPerRev=16, acc=1e-8, tWaitFinal=1;
    Z i, j, nTot=1000, nIter=4;
    Word imageName, plotDataName;
    V<Z> methods;
    bool backEvl=true;
    cpmrh(eps);
    cpmrh(nPerRev);
    cpmrh(nTot);
    cpmrh(acc);
    cpmrh(tWaitFinal);
    cpmrh(imageName);
    cpmrhf(plotDataName);
    cpmrh(methods);
    cpmrhf(backEvl);
    cpmrhf(stepRed_);
}

```

```

cpmrhf (facALF);

S<Z> met (methods); // eliminates duplicates
Z nm=met.dim();
cpmassert (nm>0, "app2 ()");
V<KepOsc> vk (nm);
for (Z i=1; i<=nm; ++i) {
    vk [i]=KepOsc (met [i], eps, nPerRev);
}
KepOsc k0=vk [1];
// This defines the initial state.
X3<R, Iv, Iv> os=k0.orbSize ();
R2 p0=k0.phsPos ();
R xInv=cpminv (os.c2 ().abs ());
R vInv=cpminv (os.c3 ().abs ());

V<V<R2> > vvr (nm, V<R2> (nTot));
R nInv=1./nTot;
for (i=1; i<=nTot; ++i) { // no graphical actions
    // within the loop. Terribly fast!!!
    KepOsc k1Act=vk [1];
    R tAct=k1Act.t (); // the states are aware of the time to which they
        // belong: Schroedinger picture
    if (!backEvl) {
        KepOsc kExt (k0);
        kExt.extEvl_ (tAct, acc); // exact evolution
        R2 posExt=kExt.phsPos (); // storing the exact phase space
        // position
        for (j=vk.b (); j<=vk.e (); ++j) {
            R2 diff=vk [j].phsPos ()-posExt;
            diff [1]*=xInv;
            diff [2]*=vInv;
            vvr [j] [i]=diff;
        }
    }
    else {
        for (j=vk.b (); j<=vk.e (); ++j) {
            KepOsc vkj (vk [j]);
            vkj.rev_ ();
            vkj.extEvl_ (tAct, acc);
            R2 diff=vkj.phsPos ()-p0;
            diff [1]*=xInv;
            diff [2]*=vInv;
            vvr [j] [i]=diff;
        }
    }
    for (j=vk.b (); j<=vk.e (); ++j) {
        vk [j].step_ ();
    }
    cpmprogress ("loop", i*nInv, 1, 2); // only 2 digits for
        // progress indication to avoid 'much adoo about nothing'
}

```

```

Graph g(cb);
g.clr_(cb);
g.setAutoScale(true);
g.setNoGridText(false);
g.setNoGrid(true);
g.setGridTextColor(LIGHTBLUE);
V<Color> vc(nm);
for (j=1;j<=nm;++j) vc[j]=vk[j].col();
g.mark(vvr,vc); // powerful method for graphical
    // representation of polygons; here one or two
    // of those
g.vis();
g.wrt(imageName); // does nothing if file name void
cpmplot(vvr,plotDataName);
cpmwait(tWaitFinal,2);
}

void app3(void) // interaction picture dynamics for not a single
// point as an initial condition but for an ensemble of
// phase space points which form a closed curve. The deformation
// of this closed curve by interaction picture dynamics is monitored.
// Here the field of view in phase space follows the moving states.
// Otherwise they would slowly drift out. For the asynchronous
// leapfrog method (method 2) it was observed that the loop
// deformation can lead to crossing over. This puzzled me for a while
// since this cannot happen with mappings of the phase space into
// the phase space. However for the asynchronous leapfrog method
// the statespace (which is properly mapped) has additional degrees
// of freedom, which are not shown in the phase space curves. This
// resolves what seemed to be a logical contradiction. The details of
// the process are not clear to me. It could be interesting to see
// whether the area enclosed by the loop remains constant even if
// crossing occurs.
{
Word loc("app3");
Color cb=morSky; // background
RecordHandler rch("app3.ini");
Word sec="data";
R eps=0.25, fac=1.2, nPerRev=16, acc=1e-8, E=0;
R facX=1, facV=1, tWaitFinal=1, tWaitFrame=0.1;
Z nTot=1000, nK=12, displayPeriod=1, method=1;
Word imageName;
bool fileOnShow=false;
V<Z> filingLimits;

cpmrh(eps);
cpmrh(E);
cpmrh(fac);
cpmrh(nPerRev);
cpmrh(nTot);
cpmrh(acc);
cpmrh(tWaitFinal);
}

```

```

cpmrh(tWaitFrame);
cpmrh(nK);
cpmrh(facX);
cpmrh(facV);
cpmrh(displayPeriod);
cpmrh(method);
cpmrhf(imageName);
cpmrhf(stepRed_);
cpmrhf(fileOnShow);
cpmrhf(filingLimits);

IvZ imgInterval; // void
if (filingLimits.dim()>=2){
    imgInterval=IvZ(filingLimits[1],filingLimits[2]);
}
// It turned out to be necessary for being able to document the
// phenomena that occur in runs to write screen images to file.
// Since filing an image takes time, it is desirable to have a way
// to file only a controlled subset of all screen frames.
// Letting the program run without any filing one can make notes
// of the frame numbers that contain the interesting events
// (such as crossing over of loops). In a second run one then may
// write these frames to file by suitable setting filingLimits.

KepOsc ko(method,eps,nPerRev,E);
R x0=ko.x();
R v0=ko.v();
X3<R,Iv,Iv> os=ko.orbSize();

KepOsc kt(ko);
kt.step_();
R xt=kt.x();
R vt=kt.v();
R dx=xt-x0;
R dv=vt-v0;
R r=R2(dx,dv).abs();
R rx=r*facX;
R rv=r*facV;
V<KepOsc> vk(nK,ko);

R phi=0;
R dphi=cpm2pi/(nK-1);
R_Vector xx(nK);
R_Vector vv(nK);

Z i,j;
for (i=vk.b();i<=vk.e();++i){
    vk[i].shf_(rx*cpmcos(phi),rv*cpmsin(phi));
    xx[i]=vk[i].x();
    vv[i]=vk[i].v();
    phi+=dphi;
}

```

```

R_Vector xx0=xx;
R_Vector vv0=vv;

R rf=r*fac;
Iv ivx(-rf, rf);
Iv ivy=ivx;
Graph g(cb);
g.setXY(ivx, ivy);
R nInv=1./nTot;
Z nPrev=nTot-1;
Z imgCounter=0;
CyclicAlarm cy(displayPeriod);
for (i=1; i<=nTot; ++i) {
    for (j=vk.b(); j<=vk.e(); ++j) {
        vk[j].step_();
        R tAct=vk[j].t();
        KepOsc koAct=vk[j];
        koAct.rev_();
        koAct.extEvl_(tAct, acc);
        R xxj=koAct.x();
        R vvj=koAct.v();
        xx[j]=xxj;
        vv[j]=vvj;
    }
    g.draw(xx.subMean(), vv.subMean(), ko.col());
    if (i<nPrev) {
        if (cy()) {
            g.setText("frame "&cpm(imgCounter++), 0.05, 0, ko.col());
            bool fileAct=false;
            if (fileOnShow) fileAct=imgInterval.hasElm(imgCounter);
            g.vis(fileAct);
            cpmprogress("loop", i*nInv);
            cpmwait(tWaitFrame);
            g.clr_(cb);
        }
    }
    else if (i==nPrev) g.clr_(cb); // that only a single
        // drawing is on the final display
    else {
        R redFac=0.5; // color of initial configuration should
            // be a bit less bright
        Color ci=ko.col()*redFac;
        g.draw(xx0.subMean(), vv0.subMean(), ci); // the final
            // image should also show the initial configuration
        g.setText("frame "&cpm(imgCounter++), 0.05, 0, ko.col());
        g.vis(fileOnShow); // display
        cpmprogress("loop", i*nInv);
    }
}
g.wrt(imageName); // does nothing if file name is void
cpmwait(tWaitFinal, 2);
}

```

```

R cpmlog10s(R x)
{
    R y=cpmabs(x);
    if (y<1e-20) return -20;
    else return cpmlog10(y);
}

void app4(void)
// Analysis for order. Normal order turns out to be 2.
// However, if order is inferred from a run of exactly one
// period, i.e. for nPerRev==nTot one gets order 4.
// If one lets nPerRev vary near nTot (notice that nPerRev
// needs not to be integer) one sees that error reduction
// starts to increase for the initial phase of doubling.
// Notice that the order 4 behavior soon gives very small
// errors which made necessary to use a save version of the logarithm.
{
    Word loc("app4");
    RecordHandler rch("app4.ini");
    Word sec="data"; // names 'rch' and 'sec' are to be employed
        // for the macro cpmrh to work
    R eps=0.25, tWaitFinal=1, nPerRev=16, acc=1e-8;
    Z i,j, nTot=16, nDoubling=2, method=2;
// getting quantities from section 'data' of file app4.ini
    cpmrh(eps); // rh: read handler
    cpmrh(nPerRev);
    cpmrh(nTot);
    cpmrh(nDoubling);
    cpmrhf(stepRed_);
    cpmrhf(acc);
    cpmrh(tWaitFinal);
    cpmrh(method);
    cpmrhf(facALF);
    cpmrhf(cpmverbose);
    Z nData=nDoubling+1;
    R_Vector x(nData);
    R_Vector y(nData);
    KepOsc koExt(method,eps,nPerRev);
    R tTot=koExt.dt()*nTot;
    koExt.extEvl_(tTot,acc);
    R2 pfExt=koExt.phsPos(); // exact phase space position
        // at the end of the trajectory under consideration
    for (i=1;i<=nData;++i){
        KepOsc ko(method,eps,nPerRev);
        for (j=1;j<=nTot;++j) ko.step_();
        R2 pf=ko.phsPos();
        nTot*=2;
        nPerRev*=2.0;
        x[i]=cpmlog10(ko.dt());
        y[i]=cpmlog10s(cpmabs(pf.x1-pfExt.x1));
    }
}

```

```

R_Vector res=polyFit(x,y,1,0,1,false);
R order=cpmabs(res[4]);
Graph g;
g.setWithOrigin(0);
g.addText("order="&cpm(order)&", method="&koExt.met());
g.show(x,y);
cpmwait(tWaitFinal,2);
}

void app5(void)
{
    Word loc("app5");
    RecordHandler rch("app5.ini");
    // This says that we expect data input to be available in file
    // app5.ini in a directory which is determined by entries in
    // cpmssystemdependencies.h and cpmconfig.ini.
    Word sec="data"; // names 'rch' and 'sec' are to be employed
    // for the macro cpmrh to work
    R eps=0.25, tWaitFinal=1, nPerRev=16, order=2;
    Z i,j, nTot=16, nDoubling=2, method=2;
    // getting quantities from section 'data' of file app5.ini
    cpmrh(eps); // rh: read handler
    cpmrh(nPerRev);
    cpmrh(nTot);
    cpmrh(nDoubling);
    cpmrhf(stepRed_);
    cpmrh(tWaitFinal);
    cpmrhf(order);
    cpmrh(method);
    Z nData=nDoubling+1;
    V< V<R2> > res(nData);
    KepOsc koExt(method,eps,nPerRev);
    R fac=1.;
    for (i=1;i<=nData;++i){
        KepOsc ko(method,eps,nPerRev);
        V<R2> resi(nTot);
        for (j=1;j<=nTot;++j){
            ko.step_();
            R tj=ko.t();
            KepOsc kj=koExt;
            kj.extEvl_(tj);
            R2 dpos=ko.phsPos()-kj.phsPos();
            resi[j]=R2(tj,dpos.x1*fac);
        }
        res[i]=resi;
        nTot*=2;
        nPerRev*=2.0;
        fac*=cpmpow(2.0,order);
    }
    Graph g;
    g.setWithOrigin(0);
    g.addText("method="&koExt.met());
}

```

```

    g.mark(res);
    g.vis();
    cpmwait(tWaitFinal,2);
}

} // namespace CpmKepler

void info()
{
    cout<<"Studying the Kepler oscillator"<<endl
    <<" This function takes 0 to 1 integer arguments"<<endl
    <<"values 1 2 3 4 5"<<endl
    <<" trigger one of the five basic modes, each other number"<<endl
    <<" triggers execution of these modes in succession."<<endl
    <<" No argument is equivalent to an argument of value 1.";
}

Z CpmApplication::main_(void)
// Selects the task from the command line, if there is a numerical
// argument following the function name. Further, the program tries to
// find and read tut2.ini. Here the filename is the name of the executable
// with all capital letters removed (so tut2MS.exe would als search for
// tut2.ini).
// Places to search for this are determined by cpmsystemdependencies.h
// and cpmconfig.ini, where the latter enjoys priority.
// If no tut2.ini can be found, the coded default selection takes
// place.
// The functionality of the tasks appl(), app2(), app3(), app4(), app5()
// is controlled through files apl.ini, app2.ini, app3.ini, app4().ini,
// app5().ini for which search places are determined by the same logic as
// those of tut2.ini.
{
    using namespace CpmKepler;
    title_="tut2";
    // No longer determining the name of the ini-file
    Z mL=1;
    Z sel=1; // default selection
    cpmverbose=1;
    if (args_.valInd(2)){ // arrays allow testing the validity
        // of indexes without running into exceptions
        Word w2=args_[2];
        if (w2=="?"){
            info();
            return 0;
        }
        else sel=w2.toZ(); // conversion to Z (integer)
    }
    Word loc("CpmApplication::main_()", sel="&cpm(sel));
    CPM_MA

    if (sel==1) appl();
    else if (sel==2) app2();
}

```

```
else if (sel==3) app3();
else if (sel==4) app4();
else if (sel==5) app5();
else { // do everything sequentially
    app1();
    app2();
    app3();
    app4();
    app5();
}
CPM_MZ
return sel;
}

// end of tut2.cpp
```

## 2.2 Discussion

The commented code which defines functions `app1()`, `app2()`, `app3()` in file `tut2.cpp` should give an impression what is going on here. Selecting `app3()` gives probably the most appealing pictures. The way to represent concepts by defining classes which are made of attributes and methods (data members, and member functions) turns out as very natural also if it comes to documentation. Whenever some statement concerning the semantics of the concept needs to be made, the appropriate placement (as a comment to the class as a whole, to a specific attribute, or to a group of attributes, or to a method (or group of methods) suggest itself in most cases. An important first step towards presenting the intended meaning of these formal constructs is the selection of names. Notice the `///descriptive full name` style of communicating the intended *descriptive full name* of public methods, from which the actual identifier is formed following the simple deterministic rule described in Section 1.

## 2.3 Run control

In all programs which have `cpmapplication.cpp` among its source files (e.g. by using the collective source file `cpmbas1.cpp`) one may control the execution by an optional configuration file `cpmconfig.ini`. This allows to set, for instance, the size of the main graphical window, the size of the pixel count of the image files that can be created instead of screen graphics, the numerical precision to be used in case that the multiple precision macro `CPM_MP` was defined for compilation. Again, there is a version of such a configuration file in which all options are explained in comments. This file is reproduced here:

```
///cpmconfig.ini
///C++ by Ulrich Mutze. Status of work 2009-05-23.
///Copyright (c) 2009 Ulrich Mutze, www.ulrichmutze.de
///All rights reserved
//
```



```

// These sizes can be made much larger than the screen for
// producing high quality graphics.
// Notice that pictures with huge pixel numbers can be transformed into
// conveniently sized jpeg-images by first creating a slightly blurred
// ppm-versions and then converting to jpeg.

////////////////////////////////////
pane sizes
////////////////////////////////////

Z pane1=150
Z pane2=100
Z pane3=50
Z pane4=100
// This overrides the default values set in cpmsystemdependencies.h.
// The status bar of the program's main window is divided into four
// segments ('panes') which are given here in arbitrary units. The
// four panes will always add to the full length available for the
// status bar. Modifying these values is a convenient way to assure
// oneself that one is actually editing the file to which the
// program reacts. Also the data in 'size of viewport' can be used for
// this purpose.

////////////////////////////////////
stereo colors
////////////////////////////////////
Zs left= 255 0 0
Zs right=0 255 255
// Optional input of color values for generation of anaglyph images.
// This overrides the default values set in code in cpmimagingtools.cpp
// or in cpmsystemdependencies.h.
// The values entered here are ideal
// if the left eye looks through a red filter which blocks green
// light and blue light (as emitted from the RGB color screen of the
// system) perfectly, and if the right eye looks through a filter which
// blocks red light (also as emitted by the color screen) perfectly.
// For systems with less perfect color separation one may find other
// different values working better such as for my sytem
// (LCD color screen, Apromax red-green filters):
// Zs left= 190 0 0
// Zs right=0 0 255

////////////////////////////////////
numerical precision
////////////////////////////////////
Z val=40
// Input for CpmRoot::numPrc.
// This value has an effect only if we have set #define CPM_MP in file
// cpmdefinitions.h
// If no value can be read from numerical precision:val (i.e. from
// the present place) then CPM_MP should either not be defined at all
// or it should provide a value such as in

```

```

//      #define CPM_MP 32
// A value of what origin ever is read as a number of decimal digits, so
// that a value 15 or 16 corresponds roughly to the numerical precision
// provided by floating point numbers of type double.

////////////////////////////////////
write precision
////////////////////////////////////
Z val=20
// Input for CpmRoot::wrtPrc. A number of decimal digits.
// This overrides the default value set in cpmsystemdependencies.h.

////////////////////////////////////
write title
////////////////////////////////////
B val=true
// If this is true, writing class instances to file will contain the
// class name as 'title'.
// This overrides the default value set in cpmsystemdependencies.h.

////////////////////////////////////
append run id
////////////////////////////////////
B val=true
    // Sets Message::appRunId
    // If this is true, the run ID will be appended to the names of
    // auto-created image files.

////////////////////////////////////
suppress messages
////////////////////////////////////
B val=false
// if we have #define CPM_NOLOG this value does not matter, it remains
// unread. If CPM_NOLOG is not defined (probably the normal case) then
// messages can be suppressed, which may be desirable for measurements
// of execution speed of algorithms where the time consumption of messaging
// is not of interest.

```

Notice that this is no header file and it will nowhere be included. Rather it will be read by the program under the control of the powerful class `RecordHandler` which defines the syntax of CPM configuration files. As can be seen in the previous example, these organize input quantities by *sections* (e.g. 'size of viewport') and associate a *type* and a *name* (e.g. 'width') with each input quantity. The valid types are B (Boolean), Z (integer), R (real), W ('word', character string), and Bs, Zs, Rs, Ws for arbitrarily long lists of those. The input scheme is recursive since the full content of an arbitrarily complex configuration file may be included by a line

```
F <name of file to be included>
```

in any configuration file. Further, comment lines (starting with '/') may be placed everywhere so that a configuration file may conveniently be used as a documentation of the intents behind the program run which it defines. Input quantities of type W

can be read as names of directories or files (with path). We thus have much flexibility in defining a particular program run by the content of a configuration file. Since this file inputs data and no code, changes in `cpmconfig.ini` do not entail need for recompilation.

Further, the program tries to find an program-specific configuration file which allows to enter information according to the same logic as information can be read from a command line. If command-line information is available, it overrides the information from the file. The name of this program-specific configuration file is defined by code in the definition of function `main_()`, see comment to statement `title_ = "tut2"` there (in `tut2.cpp` above). Here is a copy of the content:

```
// tut2.ini

////////////////////////////////////
command line
////////////////////////////////////
    // obligatory header

// Value of the subprogram selector:

// 1: phase portraits
// 2: method errors as interaction: motion in interaction picture
// 3: phase cell evolution
// 4: Determination of the order of the integrator
// 5: Position error for various step sizes

Ws args=2
    // First entry: subprogram selector
    // valid entries 1,2,3,4,5
    // All other entries trigger a consecutive running of
    // 1,2,3,4,5.
    // ? gives a short info to the console
    // second, third, ... entry not defined
```

Here is the configuration file which is specific for selection 3:

```
// app3.ini
// Phase volume during Kepler oscillator evolution
// simultaneously computed with the Direct Midpoint Method (yellow dots)
// and the Runge Kutta method of second order (white dots)
// New controls added 2009-08-03.

////////////////////////////////////
data
////////////////////////////////////

R E=0.0
// Initial value of the 'excentric anomaly'
R eps=0.22
//R eps=0.22
    // numerical excentricity, e.g. 0.17
```

```
R nPerRev=40
  // computed points per revolution (needs not to be integer)
  // e.g. 32
Z nTot=600
//Z nTot=100
  // total number of computed steps (per system)
  // e.g. 3600
Z nK=1000
  // closed curve is made of so many points

B stepRed_=false
  // If this is true, in the asynchronous leapfrog method one step
  // is composed of two steps of half the time span

//R fac=1.8
R fac=3
  // factor by which the axes of the display area are larger than the
  // exact orbit, e.g. 2.2.
  // Needs to be considerably larger than 1, if the Runge Kutta dots
  // are to stay within the frame after a few revolutions

R acc=1e-10
  // accuracy used in solving Keplers equation by iteration.

R tWaitFinal=20
  // time in s for inspecting the final state

R tWaitFrame=0.0
  // time for viewing the individual frame

Z method=3
  // 0 Runge-Kutta, 1 DMI, 2 ALF, 3 LF

Z displayPeriod=1
  // after as many steps are computed, they get transfered to
  // the display screen. Since displaying takes time, having values
  // >1 here lets the program run much faster
  // e.g. 50

Zs filingLimits=0 0
//Zs filingLimits=810 900
// lowest and highest frame number to be written to file

B fileOnShow=false
  // only if this is true, filing of screen frames will happen.
  // The last frame will be then filed anyway, other frames only
  // if their frame number is in the range given by the
  // filingLimits.

R facX=0.4
R facV=0.4
  // controls the size of the phase space subset the boundary of which
```

```

// will be visualized.

//W imageName=kneading

```

This contains only references to a further configuration files between which one can switch by commentarization. This is a mechanism to let the file name and the added comments express an intent of the run defined by the file.

```

// app3_demo2.ini
// Phase volume during Kepler oscillator evolution
// Runge Kutta method of second order (white dots)

////////////////////////////////////
data
////////////////////////////////////

R E=0.0
  // initial excentric anomaly e.g. 0
R eps=0.2
  // numerical excentricity, e.g. 0.2
R nPerRev=32
  // computed points per revolution (needs not to be integer)
  // e.g. 32
Z nTot=520
  // total number of computed steps (per system)
  // 520
Z nK=300
  // 300

R fac=7
  // factor by which the axes of the display area are larger than the
  // exact orbit, e.g. 7
  // Needs to be considerably larger than 1, if the Runge Kutta dots
  // are to stay within the frame after a few revolutions

R acc=1e-8
  // 1e-8

R tWaitFinal=6
  // time in s for inspecting the final state
  // 6

R tWaitFrame=0.5
  // time for viewing the individual frame
  // 0.5

Z method=0
  // 0 Runge-Kutta
  // else DMI
  // 0

Z displayPeriod=16
  // after as many steps are computed, they get transfered to

```

```

// the display screen. Since displaying takes time, having values
// >1 here lets the program run much faster
// e.g. 16

R facX=1
  // 1
R facV=0.25
  // 0.25

W imageName=kneadingRK
// kneadingRK

```

### 3 Defining main with the advanced C++ application framework

The following example uses the most advanced C++ application framework. For using it one has to have not only `cpmapplication.cpp` among the source files of the project but also `cpminifilebasapp.cpp`. The main service of this extended framework is some automatizing in generating introductory screens for the subprograms and in generating documentation files. The topic under study is Romberg integration of singular functions which is chosen as a vehicle to show ‘functions as values’ in action.

#### 3.1 Code example `chebyshev.cpp`

```

/// chebyshev.cpp
/// C++ by Ulrich Mutze. Status of work 2012-01-12.
/// Copyright (c) 2012 Ulrich Mutze, www.ulrichmutze.de
/// All rights reserved

/*****

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 introductory part of this file which gives
more information on the topics the program deals with:

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

```



```

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 <cpmbasl.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 cpmPow(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.abs()/(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", (R_)eps,2);
    cpmvalue("exp", (R_)exponent,3);
    cpmvalue("res", (R_)res,4);
    // writing to the status bar
    return res;
}

R sinl(R const& x){ return sin(x);}
R cosl(R const& x){ return cos(x);}
R acosl(R const& x){ return acos(x);}
// If multiple precision arithmetic is active
// the native trigonometric functions have two
// arguments. For the second argument there is
// a default value provided so that a normal function call
// needs only a single argument (as above). For the way we
// use these functions in the following, the second argument
// must not be ignored. So, we define functions with the
// appropriate argument structure here.
}

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(acos1);
Fa<R,R> Cos(cos1);
    // bringing C -functions cos and acos into the F<R,R>
    // framework. To allow for arbitrary precision arithmetic
    // we do not simply write acos and cos here.
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/cpmsqrt(cpmpi/2);
R facN0=1/cpmsqrt(cpmpi);

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 background 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 curves
    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);

```



```

// 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.vis(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(Word(),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", (R_)eps,1);
cpmvalue("exp", (R_)exponent,2);
cpmvalue("yMin", (R_)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.vis(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.vis(writeOnShow);

  R_Func weight=F2<R,R,R,R>(eps,exponent)(rws);
  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();
  V< V<C> > vs(n, V<C>(n));
}

```

```

    for (i=1;i<=n;++i) for (Z j=1;j<=n;++j) vs[i][j]=C(ort[i][j]);
    Color::setGamma(gamma);
    gr2.clr();
    gr2.mark(vs,Iv(),interpolate);
    gr2.setText("Maximum of absolute values is "&cpm(absMax));
    gr2.vis(writeOnShow);

    lines.last()="Computation done";
    gr1.clr();
    gr1.setText(lines);
    gr1.vis(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;
}

```

## 3.2 Discussion

For questions which these code examples may rise, and which are not anticipated in the comments added, the reader is invited to look up the explaining remarks to classes and their methods in

[www.ulrichmutze.de/softwaredescriptions/cpmlisting.pdf](http://www.ulrichmutze.de/softwaredescriptions/cpmlisting.pdf)

Last modification: 2011-11-02.