On the C+− project

Ulrich Mutze
www.ulrichmutze.de

March 23, 2016

The aim of this project is to represent an interesting part [1] of computational physics as a system of C++ classes and also to provide a corresponding C++ representation of the mathematical structures and methods that are needed to implement specific physical systems and their behavior. Also 2D and 3D visualization and image processing are treated.

Presently this physical and mathematical class system comprises 3.0 MB of zipped code in 275 files. As far as physics is concerned, it deals mainly with electrostatic and magnetostatic fields, mechanics of rigid bodies, and non-relativistic quantum mechanics of many-particle systems. The mathematical framework which comprises linear algebra, interpolation, calculus, fast Fourier transform, minimization, geometry, propositional logic, hereditarily finite sets, and graphical primitives (including alphabets) is even larger.

By means of the program *pdflatex* all code can be listed in a single file *cpmlisting.pdf*,[17], of about 3200 pages and 4.9 MB. For each of the files there is a chapter, registered in a table of content. Due to the fast searching abilities of modern PDF-readers one has all code details at ones finger tips.

Although the material was accumulated over an extended period of time (starting in 1989), the style of presentation and coding is fairly homogeneous since all major reorganizations which the system had to survive during its growth were extended to the whole of the existing material.

It is the main purpose of this article to explain and document the principles of this style in order to enable extensions of the class system without unnecessary obscuration of this style. A further intent is to invite readers to download the code and to experience how their own C++ programming can take advantage of the many man years of programming labor condensed in this code. In the last section of this article I'll come back to this point.

[1] actually, the one on which I happened to gain some experience

1 On the C+− programming style

As Bjarne Stroustrup says: ‘Within C++ , there is a much smaller and cleaner language struggling to get out’ ([4], p. 207). And, ‘I think *a smaller language* is number one on any wish list for C++ . . . ’ ([4] p. 198). With all the C++ code in use, and the expectations of the C++ community concerning continuity, C++ should not be expected to get ever cut down to such a smaller and cleaner language. What can be done, however, is to figure out the subset of C++ which supports a preferred programming style. To cite Stroustrup again: ‘From one point of view, C++ is really three languages in one:

- A C-like language (supporting low-level programming)
- An Ada-like language (supporting abstract data type techniques)
- A Simula-like language (supporting object-oriented programming)

. . . There always is a design choice but in most languages the language designer has made the choice for you. For C++ I did not; the choice is yours’ ([4] p. 199).

It is in the spirit of such statements, that I like to consider the system of C++ classes and functions to be described here as a programming language, named C+− . This name [2] is to express that the language results from C++ by taking away (actually encapsulating) language constructs.

1.1 No pointers

For instance, in C+− one never needs the address operator `&` or the dereferencing operator `*` or the operators `delete` and `delete[]`. In particular one is unable to create *bad deletes* — common sources of disaster in C++ programming. Unlike the situation with a new standalone programming language one needs no separate compiler for C+− . And, all features of the good old mother language C++ remain available for situations for which C+− turns out not to provide the appropriate tools [3] . As is clear to every C++ programmer, C+− cannot simply be a programming style guide saying that one should avoid pointers and references. This would come close to saying that one should avoid all non-trivial programming. So, C+− has to provide the tools for handling the problems that normally are solved by pointers, such as looping over heterotypic collections of objects. To be specific, the C+− tool for this task is the polymorphic vector template `Vp<>` ; the normal (homotypic) vector template is `V<>`. A typical loop over the index range of a vector `v` reads

```
for (Z i=v.b();i<=v.e();++i) v[i]*=2;
```

[2] the programming language joke http://www.danielsen.com/jokes/cplusminus.txt became known to me much later

[3] Whenever I encountered such a situation, I added the tools, to the effect that it no longer happened for quite a while in the core material. Actually, in October 2010 a major internal concept shift took place in providing the basic array template V with an arbitrary index range. Prior to this change one had a template Vl with indexes starting at 0 and the template V with indexes starting at 1.

where `Z` is the C+− alias for `int`; `R` is the C+− alias for `double` [4] . Of course, the implementation of `V` and `Vp` relies on pointers, only the interface is free of them. The simplest constructor such as for vectors `x` and `y` in

```
Z n=...;
R t=...;
V<R> x(n,t);
V<Z> y(n);
```

have valid indexes starting with 1. The statements

```
x.b_(0);
y.e_(n-1);
```

turn `x` and `y` into arrays with valid indexes

$$i \in \{0, \ldots, n-1\} \, .$$

Code such as

```
R sum=0;
Z i,j;
for (i=x.b(),j=y.b();i<=x.e();++i,++j) sum+=x[i]*y[j];
```

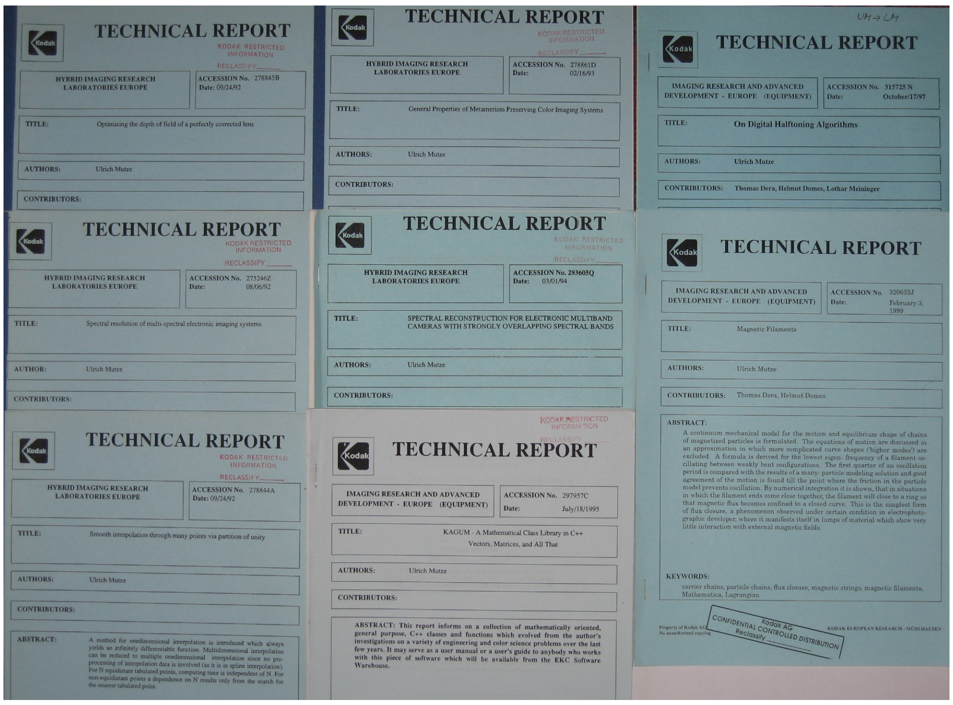
works irrespective of index shifts for `x` or `y`.

1.2 Values, values, values ...

The problems that led me to develop the C+− programming style arose in industrial engineering projects, see Figure 1, and in personal work on computational physics and constructive mathematics. It was not so much the problem field that shaped this style, but the attempt to write programs that represent natural reasoning as directly as possible. Although the evolving style appeared to me for a long time as a bundle of loosely related tastes and preferences (e.g. for short names, so that `int` and `double` get renamed to `Z` and `R` as encountered already), it turned out to allow a succinct characterization in terms of C++ class terminology: Consider a class `X` that behaves as much as a fundamental type (e.g. int, alias `Z`) as possible: Obviously it should have

1. a default constructor which allows to say `X x;` to the effect that now object `x` exists and is of type `X`,
2. a copy constructor which allows to say `X y=x;` to the effect that we have in addition to `x` an `X`-typed object `y` which has the same state as `x`,
3. an assignment operator which allows to say `y=z;` for any `X`-typed object `z` to the effect that now `y` has the same state as this `z`.

[4]There is no relation to the well-known statistics oriented software system named R. Of course, the association of 'R' with the set of real numbers is much older than the R software.


TECHNICAL REPORT
KODAK RESTRICTED INFORMATION
RECLASSIFY_______

HYBRID IMAGING RESEARCH LABORATORIES EUROPE
ACCESSION No. 278845B
Date: 09/24/92
TITLE: Optimizing the depth of field of a perfectly corrected lens
AUTHORS: Ulrich Mutze
CONTRIBUTORS:

TECHNICAL REPORT
KODAK RESTRICTED INFORMATION
RECLASSIFY_______

HYBRID IMAGING RESEARCH LABORATORIES EUROPE
ACCESSION No. 278861D
Date: 02/16/93
TITLE: General Properties of Metamerism Preserving Color Imaging Systems
AUTHORS: Ulrich Mutze
CONTRIBUTORS:

UM→LM

TECHNICAL REPORT

IMAGING RESEARCH AND ADVANCED DEVELOPMENT - EUROPE (EQUIPMENT)
ACCESSION No. 315725 N
Date: October/17/97
TITLE: On Digital Halftoning Algorithms
AUTHORS: Ulrich Mutze
CONTRIBUTORS: Thomas Dera, Helmut Domes, Lothar Meininger

TECHNICAL REPORT
KODAK RESTRICTED INFORMATION
RECLASSIFY_______

HYBRID IMAGING RESEARCH LABORATORIES EUROPE
ACCESSION No. 275246Z
Date: 08/06/92
TITLE: Spectral resolution of multi-spectral electronic imaging systems
AUTHOR: Ulrich Mutze
CONTRIBUTORS:

TECHNICAL REPORT
KODAK RESTRICTED INFORMATION
RECLASSIFY_______

HYBRID IMAGING RESEARCH LABORATORIES EUROPE
ACCESSION No. 293605Q
Date: 03/01/94
TITLE: SPECTRAL RECONSTRUCTION FOR ELECTRONIC MULTIBAND CAMERAS WITH STRONGLY OVERLAPPING SPECTRAL BANDS
AUTHORS: Ulrich Mutze
CONTRIBUTORS:

TECHNICAL REPORT

IMAGING RESEARCH AND ADVANCED DEVELOPMENT - EUROPE (EQUIPMENT)
ACCESSION No. 320633J
Date: February 3, 1999
TITLE: Magnetic Filaments
AUTHORS: Ulrich Mutze
CONTRIBUTORS: Thomas Dera, Helmut Domes
ABSTRACT:
A continuum mechanical model for the motion and equilibrium shape of chains of magnetized particles is formulated. The equations of motion are discussed in an approximation in which more complicated curve shapes ('higher modes') are excluded. A formula is derived for the lowest eigen- frequency of a filament oscillating between weakly bent configurations. The first quarter of an oscillation period is compared with the results of a many- particle modeling solution and good agreement of the motion is found till the point where the friction in the particle model prevents oscillation. By numerical integration it is shown, that in situations in which the filament ends come close together, the filament will close to a ring so that magnetic flux becomes confined to a closed curve. This is the simplest form of flux closure, a phenomenon observed under certain condition in electrophotographic developer, where it manifests itself in lumps of material which show very little interaction with external magnetic fields.
KEYWORDS:
carrier chains, particle chains, flux closure, magnetic strings, magnetic filaments, Mathematica, Lagrangian
Property of Kodak AG
No unauthorized copying
Kodak AG
CONFIDENTIAL CONTROLLED DISTRIBUTION
Reclassify_______
KODAK EUROPEAN RESEARCH - MÜHLHAUSEN

TECHNICAL REPORT
KODAK RESTRICTED INFORMATION
RECLASSIFY_______

HYBRID IMAGING RESEARCH LABORATORIES EUROPE
ACCESSION No. 278844A
Date: 09/24/92
TITLE: Smooth interpolation through many points via partition of unity
AUTHORS: Ulrich Mutze
CONTRIBUTORS:
ABSTRACT: A method for onedimensional interpolation is introduced which always yields an infinitely differentiable function. Multidimensional interpolation can be reduced to multiple onedimensional interpolation since no preprocessing of interpolation data is involved (as it is in spline interpolation). For N equidistant tabulated points, computing time is independent of N. For non-equidistant points a dependence on N results only from the search for the nearest tabulated point.

KODAK RESTRICTED INFORMATION
RECLASSIFY_______
TECHNICAL REPORT

IMAGING RESEARCH AND ADVANCED DEVELOPMENT - EUROPE (EQUIPMENT)
ACCESSION No. 297957C
Date: July/18/1995
TITLE: KAGUM - A Mathematical Class Library in C++ Vectors, Matrices, and All That
AUTHORS: Ulrich Mutze
CONTRIBUTORS:
ABSTRACT: This report informs on a collection of mathematically oriented, general purpose, C++ classes and functions which evolved from the author's investigations on a variety of engineering and color science problems over the last few years. It may serve as a user manual or a user's guide to anybody who works with this piece of software which will be available from the EKC Software Warehouse.

Figure 1: Technical Reports which define a part of the problem field on which C+− grew.

Further, an existing instance of `X` should not change its state unless by action of code in which it 'appears personally' — with its own name mentioned — as `y` in the foregoing `y=z` example. By the same token, the code `x=y; y.changeThis();` should not change `x`, although it does so in Java, C# , Ruby, Python, and in Julia. Let us call a class `X` that satisfies these requirements a 'value class', and each instance of a value class a 'value'. With this notion the C+− main strategy can be expressed as follows:

- Whenever a class is to be defined, try hard to make it a value class.
- Whenever a class template is to be defined, ensure that it defines a value class whenever the template arguments are value classes.

To put these rules into perspective, we note that a class `X` the data members of which all are values, is itself a value class with the trivial definition `X(){}` of the default constructor and the implicit (tacitly provided by the compiler) definition of a copy constructor and assignment operator. So, values are already privileged by the C++ language definition. Restricting the language to them (and providing replacements for pointers, which are non-values par excellence) has a chance to make this 'smaller and cleaner' language Stroustrup speaks about. C+− has value classes which improve the C++ concepts of *function pointers*, and of *file streams*. Further, there are value classes for *graphical windows* and for graphical objects of various kinds.

In analyzing a general C++ program, understanding the values under consideration is the easy part and understanding the pointers and the references can be difficult. In a C+− program there are only values, and the difficult part is missing by design.

This characterization of values does not imply that values are passed to functions as 'call by value'. Actually, all C+− functions call class-typed arguments as references and in virtually all cases as constant references. Thus copying the probably large objects to the function stack never takes place.

1.3 Reference counting and 'copy on write'

However, in working with value objects it is easy to ask for copies of objects that could be replaced by mere references if the value nature of the quantities would not give clear directions to the contrary. For large objects this may cause performance problems. C+− considers this not a problem for those objects which get their data content by code that introduces constant-size data with an individual identifier for each, as in

`class X{ R a,b,c,d,e; ...};`,

and it *provides a radical solution* for cases where memory gets allocated under the control of variables such as in

`Z n=...; V<R> v(n);`.

This is achieved by having a single class template `V<>` for dynamic memory allocation in C+− and by letting this class implement 'reference counting' and 'copy-on-write' — the well known remedy for the superfluous copying problem. Also the other C+− container-like class templates, namely those for building sets and maps are based on (ordered) arrays (not red-black trees, see [5] p. 241) and thus benefit from the same copy protection. This is in remarkable contrast to the C++ standard library container classes (e.g. [8],[7],[5]) which don't implement reference counting. Therefore, program parts which repeatedly do copy construction and assignment on heavy types such as `vec<vec<R> >` may run by an arbitrarily large factor faster for vec=`V` than for vec=`vector` or vec=`valarray` [5] . In a given C+− based program one assesses the benefit from using reference counting by experimentally commentarizing the line `#define CPM_USECOUNT` in file cpmdefinitions.h. I found some of my programs running much slower with disabled reference counting. But in most of them, there is no significant difference. The slow programs could be accelerated by the techniques indicated in [8] Section 22.4.7. However, it is the ambition of C+− to free the user from such low-level considerations.

The C+− container classes take advantage of the freedom to provide full support only for value classes. The C++ standard library, by the basic intention of C++ to be 'a better C', is not allowed to exclude valid types such as `char*` from their services. This has decisive influence on the structure of this standard library as shall be pointed out with the example of sorting an array. Is there something more natural than saying `x.sort()` to an array object `x` to bring it into ordered form? (Actually, C+− says `x.sort_()` to express that `x` gets changed, Ruby says `x.sort!` instead.) In the C+− framework this works since C+− assumes that the the components of `x` are instances of classes (or

[5]Surprisingly this is still the case even though class std::vector now (i.e with C++11) is endowed with the new move functionality

of a fundamental type, which makes no difference due to a template based unification method to be explained in the next sub-section). If order matters at all for these classes, its definition has to be given as a part of the class definition. For the C++ standard library such an assumption is not acceptable since also `char[]` is a decent C++ type and the standard library has to provide means to sort objects of this poorly equipped type. Thus the C++ standard library is forced to define the sorting algorithm as a classless function with an iterator as a connecting agent and a function object as template argument which says how to order chars in the situation under consideration. Who has worked with this scheme for a while might start to consider it natural and to admire its universality. Nevertheless, it is forced upon us by adverse conditions, which we are free to escape — not as C++ , but as C+− . Sorting as a service of an array class is a sound concept, as may be concluded from how well it works in the programming language Ruby [11] — and in C+− .

1.4 Unifying the treatment of C+− classes and built-in types

Having both fundamental types and classes in C+− is not ideal. Especially, template code could be simplified if the fundamental types would be replaced by classes [4], p. 380. The pure doctrine is formulated boldly in [6], p. 777 guideline 43: 'Avoid C's built-in types. They are supported in C++ for backward compatibility, but they are much less robust than C++ classes, so your bug-hunting time will increase.' All my experiments with class-typed numbers were not very encouraging due to considerable speed loss of programs. So I decided to continue to use the built-in types, particularly type double as a work horse for fast floating point arithmetic [6] . On the other hand, to be restricted to the numerical precision supported by double numbers is unsatisfactory for many problems and for two such problems I became interested in autumn 2008:

1. There are well-behaved step-wise solution algorithms (integrators) for dynamical systems which always produce reversible discrete trajectories even if the true continuous trajectories are not reversible. To avoid a hard contradiction the discrete algorithm escapes into extremely large numbers which transcend the limitations of double and long double [12].

2. There are dynamical systems which exhibit a symmetry, such as mirror reflection symmetry, which gets lost due to numerical errors surprisingly fast. Only increasing the numerical precision allows one to see symmetric motion over a an extended period of time. An animated demonstration of this behavior is to be seen on my home page [15].

I first tackled these problems by means of the programming language *Ruby* since here arithmetic with arbitrarily precise numbers is readily available with the Ruby class `BigDecimal` and the Ruby module `BigMath`, both created by Shigeo Kobayashi. Nevertheless, coding dynamical system evolution in a manner that switching between inherent

[6] Experiments in May 2012 under Linux and g++ 4.6.3 gave only a few percent performance loss when R was replaced by a wrapper class.

floating point precision and arbitrarily selected precision becomes possible without much ado asked for some additions to these Ruby tools. These additions are now available for all Ruby users from [13]. A contribution by Roger Pack to my Ruby forum discussion: *Proposing an arbitrary precision class building on BigDecimal and being derived from Numeric* hinted to similar activities in C++ and it was class `mpreal` by Pavel Holoborodko [14] which looked particularly useful to me. It provides a simple user interface along the line of thinking that also underlies my Ruby class `AppMath::R`. Actually it turned out that replacing C++ type `double` by class `mpreal` in all C+− code was quite feasible and enforced only minor code modifications to satisfy my two 'reference compilers/linkers' which at that time were Microsoft Visual C++ 2008 Express Edition and g++ 3.4.4 (cygming special). The principle which makes `mpreal` replaceable with `double` in virtually all expressions is that it overloads arithmetic operators for all relevant C++ built-in types as in the following small section of mpreal.h where addition of any number to a `mpreal`-number is declared:

```
const mpreal operator+(const mpz_t b, const mpreal& a);
const mpreal operator+(const mpq_t b, const mpreal& a);
const mpreal operator+(const long double b, const mpreal& a);
const mpreal operator+(const double  b, const mpreal& a);
const mpreal operator+(const unsigned long int b, const mpreal& a);
const mpreal operator+(const unsigned int b, const mpreal& a);
const mpreal operator+(const long int b, const mpreal& a);
const mpreal operator+(const int b, const mpreal& a);
```

I never considered this method an option and thought that one should avoid the need for so many function definitions by relying on automatic type conversions. This however created a lot of unexpected ambiguities in expressions occurring in my huge C+− code base, for which fixing was in no case difficult but, as a consequence of the large number of defective expressions, very tedious. So I came to admire the brave and clean approach [7] of P. Holoborodko and decided to make replacing type `double` by class `mpreal` an option in C+− . Since in C+− type `double` appears always under the alias name `R`, this is done easily by switching the definition of `R` which is given in cpmnumbers.h and which is partially shown here:

```
#if defined(CPM_MP) // MP stands here for 'multiple precision'.
   // a typical line to appear in cpmdefinitions.h is
   // #define CPM_MP 32
   // where the number gives the number of decimal digits.
   // In projects which use cpmapplication.cpp this number
   // can be overridden in cpmconfig.ini without causing a need for
   // recompilation. The expected entry there is of the form:
   //    numerical precision     // section
   //    Z val=64                // value
      // mpfr::mpreal is a wrapper class by Pavel Holoborodko to the
      // MPFR function system. See
      // http://www.holoborodko.com/pavel/
      // This class has an interface to other
      // C++ types which is very similar to that of type double.
      // So an algorithm coded in conventional style will still work
      // when type double is replaced by class mpfr::mpreal. Since the
      // information content (number of digits) can be set 'arbitrarily'
      // large, there is no longer a fundamental difference between
```

[7]The most recent version of mpreal.h takes advantage of type conversions to some extend.

```
      // class mpfr::mpreal and the mathematical concept of the real
      // number field. See the section 'motivation and rational'
      // in the documentation to class AppMath::R in
      // http://www.ulrichmutze.de/rubystuff/doc_rnum/index.html
      // for more motivation concerning this point of view.
   #define CPM_ mpfr
      // CPM_::sin(r) (i.e. mpfr::sin(r)) is the full name of the sine
      // function for argument r \in mpfr::mpreal. This is then a clear
      // distinction from e.g. CpmRoot::sin(C const&).
   typedef mpfr::mpreal R;
      // Defining type R, the real type with which present C+- code uses
      // in all places where floating point values are needed. No longer
      // I use floating point types of multiple and fixed precision (such
      // as R together with Rh or R_) in parallel.
#else
   // Then we don't need the multiple precision libraries and get
   // more speed. This was the only mode of C+- till March 2009. We have
   // the choice between double (64 bit) and long double (80 bit on most
   // systems) as a definition of type R.
   #define CPM_ std
      // Again, CPM_::sin(r) (i.e. std::sin(r)) is the full name of the
      // sine function for argument r of type double or long double.
   #if defined(CPM_LONG) // then: typedef long int Z
      typedef long double R;
   #else
      typedef double R;
   #endif
#endif
```

After all, a common infrastructure for all C+− types cannot be achieved by providing a common base class for all of them, since certainly Z and probably R is not a class at all. Instead, C+− provides an efficient template-based mechanism to handle value classes and fundamental types on equal footing. It might be instructive to describe this mechanism here for the example of input from streams and output to streams — a topic which often is referred to as *marshaling*. A more technical description is in the commentaries to class template IO<> in file cpmnumbers.h. There is a particular issue in marshaling that is addressed in C+− : Data content of files becomes much more useful if it is allowed to contain comments that do not interfere with the data content. C+− knows the concept of a *comment line* in a data file; the first non-white-space character of such a line is '/' or one of the other characters singled out by the following function

```
bool CpmRoot::startsComment(char c)
{ return c=='/' || c=='*' || c==';' || c=='#';}
```

Thus a single slash would be sufficient to mark a line as a comment, but all comment lines generated automatically by C+− use //, so that these look just like comment lines in C++ source files. Of course, a user of C+− source code is free to change the definition of this function according to his needs. This will change the concept of a comment line whenever it appears somewhere in C+− code — in accordance with one of the C+− guiding principles: *Each concept should be defined in a single place.* With the advent of namespaces, this place needs no longer be a class; it can also be a class-less function in a namespace, or even data in a namespace. Comments will be automatically inserted when data of an aggregated data type get written to files: Suppose that we have in a program a quantity `x` of type `V<V<R> >` and that some unexpected behavior of this program hints at the possibility that this quantity did not get the value it was expected to get. We simply add a statement `cpmdebug(x);` to our code and expect to get a readable account of this quantity on the basic logfile cpmcerr.txt. Assume that `x` was created as `V<V<R> > x(3,V<R>(2));` and — by some misconception in a control structure — has still its initial value. This then can be seen on cpmcerr.txt as follows:

```
C+- debug: x=//V<V<double>> begin
//IvZ
3
1
//V<double> begin
//IvZ
2
1
0
0
//V<double> end
//V<double> begin
//IvZ
2
1
0
0
//V<double> end
//V<double> begin
//IvZ
2
1
0
0
//V<double> end
//V<V<double>> end
```

Here we see that writing a `V` on file has the form typename begin ... typename end which is very convenient for large vectors. The first data item to be written is the index range which is of type `IvZ` (interval of integers). This type corresonds to class `Range` of Ruby but defines more useful operations, such as the lattice operations join and meet, than Ruby's `Range`. The data characterizing an instance of `IvZ` are the number of elements (its cardinality) and the first (lowest) of its elements. After these data the list of components follows. In our case, these components are instances of `V<double>` and can be recognized as such. Their components are not preceded by a typename since they are simple numbers. The names of classes appearing here do not rely on the typeid system of C++ which typically gives very long names since all qualifications are included. This output to the log file comes finally from a call to the stream shift operator `<<` as is clear from the definition

```
#define cpmdebug(X) cpmcerr<<endl<<"C+- debug: "<<#X "="<< X <<endl
   // useful for placing temporary debugging statements into ones code.
   // From Bruce Eckel: Thinking in C++, Prentice Hall 1995, p. 325
```

in file cpmtypes.h. For most C+− classes, the the stream shift operator `<<` is defined by the declaration/implementation macro `CPM_IO`, see section 4 and file cpminterfaces.h, in terms of the class template `IO<>` as

```
friend std::ostream& operator<<(std::ostream& out, Type const& x)
{ CpmRoot::IO<Type>().o(x,out); return out;}
```

This becomes active only in classes which employ a macro of the `CPM_IO`-family. It is not active for aliased built-in types like `Z` and `L` and for classes which know nothing from C+− such as `std::string` or `mpreal`. For these latter types the macro `cpmdebug` thus will call their own operator `<<` and this outputs only a value and no comment which indicates the type name of the value. Notice that writing a title together with data is acceptable only if the corresponding reading process `>>` is defined such that it reads over these titles. Otherwise one would be unable to retrieve written data. Since writing human-readable titles in addition to the data enlarges file size, and also makes no sense in writing well-understood data that need not be read by any human, it can be disabled by letting the macro `CPM_WRITE_TITLE` undefined in file cpmsystemdependencies.h. The definition of class template `IO<>` is

```
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);}
};
```

This definition is valid for all types `T` for which there is no re-definition (specialization or partial specialization in template parlance). If `T` is not a class, the previous definition cannot apply since then there are no member functions and the terms `t.prnOn(str)` and `t.scanFrom(str)` are not defined. Also if `T` is a class (e.g. `mpreal`) but does not define member functions `prnOn` and `scanFrom` we are able to make `IO<T>` defined by providing a so-called specialization, or — if `T` itself is a class template — a partial specialization. For the important case that `T` is `Z` the specialization is

```
template<>
class IO<Z>{ // specialization of IO for T = Z
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);}
};
```

Here the function `CpmRoot::read` is defined in a way that it reads over comments. See the definition in cpmnumbers.(h,cpp) for details. Analog specializations are given for `IO<double>`, `IO<float>` and — if `CPM_MP` is defined — also for `IO<mpreal>`. For these cases the term `CpmRoot::read(t,str)` is defined by overloading function `read`. If `T` is a class, but has no member-functions `prnOn` and `scanFrom`, one also has to define a specialization, as it is done here for `std::vector` to let it cooperate with the C+− array classes:

```
template<class T>
   class IO< std::vector<T> >{ // 'partial specialization' , since the
      // template parameter T implies that some 'generality' remains
```

```
    public:
      IO(){}
      bool o(std::vector<T> const& v, ostream& str)const
      { return CpmArrays::V<T>(v).prnOn(str);}
      bool i(std::vector<T>& v, istream& str)const
      {
         CpmArrays::V<T> vl;
         bool res=vl.scanFrom(str);
         v=vl.std();
         return res;
      }
  };
```

Here the definition of `prnOn` and `scanFrom` is naturally carried over from `V<T>` to `std::vector<T>`. C+− does not aim at integrating the standard library containers in so far as inserting any standard container class as template argument into a C+− class template provides for the resulting class the same functionality which it would provide for a C+− class as argument. The most intimate connection of C+− classes with standard container classes is with class `std::vector<>`. For small types as template arguments, such as `Z`, `R`, and `C` the class `std::vector` is rather efficient and it is considerably faster than `V<>` if appending of components occurs on a regular basis. This is the special demand for which `std::vector`, unlike all C+− arrays, has special provisions built in. `V<>` can efficiently be converted to `std::vector<>` back and forth. Also the class `std::map<>` seems to be indispensable in some situations as it was found for small types as template arguments considerably faster (I observed a factor of 13 in a quantum mechanical many particle application) than `CpmArrays::M<>` for the same arguments.

1.5 Functions as values

There is an important facility in C+− which is not [8] present in C++ . For value classes `X` and `Y` there is the value class `CpmFunctions::F<X,Y>` of function objects `f: X->Y` (i.e. for any instance `x` of `X`, `f(x)` is an instance of `Y`). This then implies that we have two ways of incorporating functions in classes: The traditional way of using member functions or the new way of using `F<>`-typed data members [9]. Unlike member functions, these can be set in constructors and modified by non-constant member functions. The basic dichotomy with classes, to have data members and member functions seems to fade away with this mechanism. C+− continues to organize most of the class functionality in member functions. Sometimes, however, classes become more functional by having function-typed data members. This is especially the case for classes which describe shapes of material bodies or fields in space. In both cases the preferred representation in not by discrete nodes or discrete points carrying field values but by functions that can directly (i.e. without involving interpolation) be evaluated at any point in space.

For bodies, the representation is a special case of the well-known implicit representation of a surface by an equation of the form `f(x)=0`: here `f` is chosen such that for

[8]With C++11 we have $function\langle Y(X)\rangle$, which, however, doesn't fully behave as a value class.

[9] functions as values are discussed in [9] p. 241

points `x` which are near the surface of the body, the value of `f(x)` equals approximately the distance of `x` from the surface. In any case, it is positive for points outside the body and negative for interior points. I call such a function a *metrical indicator function* of the body. For two bodies, each described by a metrical indicator function, the set union is described by the maximum of the two indicator functions and the set section by the minimum. Thus it is easy to obtain metrical indicator functions for a large variety of shapes, whereas functions that give the exact distance to the surface could be given only for very simple shapes like spheres, cylinders, and parallelepipeds.

For fields, the definition of the field strength at arbitrary points is derived from the field sources, which are assumed to be given. For sources such as point charges, dipoles, homogeneously charged rectangular surface patches, or homogeneously magnetized rectangular parallelepipeds computationally efficient formulas are implemented.

To come back to the general aspects of functions as data: The spirit of 'object orientation' guides us to discover the true distinction between normal data elements and function-type data elements. We have to consider their behavior, i.e. the methods which they allow to define: reading from streams and writing to streams (or files) is natural for normal data elements. The same is true for the notion of equality. Both concepts are far from trivial for purely functional data, since pieces of code cannot be handled in C++ as data in any direct manner. For a definition of equality of functions, even access to the defining code would not solve the problem, since equal functions does not mean equal code.

In many practical situations functions depend on parameters: For instance, when considering the distance traversed by a falling body as a function of time, the gravitational acceleration g enters as a parameter. Now it may happen that we have to consider the function which has g as the function argument (e.g. when dealing with free fall on various planets) and some characteristic duration (say, 1.37 seconds) as a parameter. This could lead us to define two functions

```
R ft(R const& t, R const& g){
   return g*t*t*0.5;
} // t is the argument, g the parameter

R fg(R const& g, R const& t){
   return ft(t,g);
} // g is the argument, t is the parameter

F<R,R> st = F1<R,R,R>(9.81)(ft);
   // st is the function t |--> 9.81*t*t*0.5

F<R,R> sg = F1<R,R,R>(1.37)(fg);
   // sg is the function g |--> g*1.37*1.37*0.5
```

although we actually deal with two aspects of a single natural law. C+− provides the tools to create these two functions out of a single definition of the basic law:

```
R f(R const& t, R const& g){
   return g*t*t*0.5;
} // law of free fall coded once
```

```
   F<R,R> st = F1<R,R,R>(9.81)(f);
      // st is the function t |--> 9.81*t*t*0.5

   F<R,R> sg = F1_1<R,R,R>(1.37)(f);
      // sg is the function g |--> g*1.37*1.37*0.5
```

The normal way is to make the function depending on parameters a static member of a class and let the constructors `Fn_m` generate the paramter-binded forms which are useful as non-static member functions.

2 C+− applications

It is the philosophy of C++ not directly to support the definition of user interfaces of programs and interfaces of programs to graphical hardware. C+− , however, should allow the creation of working programs (applications) and thus cannot hold such a restrictive position. The application framework provided with C+− only supports applications which read instructions from one or more files, and write data on files, and show screen graphics and status indicators during run. User interaction is restricted to canceling the run, and the program enjoys high priority [10]. An application made by C+− tools alone is primarily an console application but together with the terminal window it opens a main graphical window entitled 'Non-interactive OPENGLUT window for <name of executable>'. There are two versions of the C+− application framework, the second being a bit more functional and sophisticated than the first one. Using the first one is done by making cpmapplication.cpp one of the source files of the project and give a project specific implementation of a function `Z main_(void)` in the main source file of the project. (the traditional `int main(...)` is defined in cpmapplication.cpp in a way that it calls `Z main_(void)` after preparatory actions concerning graphics and reading of input files).

For the second version of the framework one needs cpminifilebasapp.cpp as an additional source file. The main source file of the project has to define a project specific class which has to be derived from `CpmApplication::IniFileBasedApplication`. Let us describe things by the example of project chebyshev, [18]. Here the project specific content is in the file cpmchebyshev.cpp and is of the following form:

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

ChebyApp::ChebyApp(void):IniFileBasedApplication("chebyshev"){}

void ChebyApp::doTheWork()
```

[10] Of course, all the C+− workhorse classes like arrays, functions, minimizers, integrators, ... can well be used also to implement interactive programs.

```
// does what the name says
// Here comes the project specific part of the code
{
....
}

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;
}
```

The files cpmapplication.h and cpminifilebasapp.h give a full description of the C+− application framework.

This type of application supports what I imagine that a C+− based program will primarily be made for: to produce data that one really wants to have, e.g. for a diagram in a publication or for deciding a relevant question. The final form of these data will be created by a 'production run' and the determinants of this run have to be documented in a set of files which, by a run identifier appended to their name, are reliably marked as belonging together. This allows us to retrieve the values of all parameters with which our results were obtained. These run determinants often result from a development process that includes both design and experimentation. At least in the early phase of experimentation it would often be inconvenient to create all this documentation material. Therefore a handy way is implemented to switch between light documentation of experimental runs and full documentation for production runs.

In principle there are three ways to enable experimentation:

- Modifying the program state during run by user interaction.
- Modifying values of initialization data.
- Modifying the program in a way that new control parameters are introduced or that the influence of existing control parameters is modified.

To organize user interaction with a program in a way that it allows fast and flexible experimentation is difficult — at least with the GUI systems I worked with so far. In order to let a production run have all its influencing data well documented, an interactive system has to document its internal state in a way that also is not easy to achieve in a sufficiently flexible manner. The second way is the regular one: the values of parameters can be changed in the initialization file(s) and the run documentation will list all values in readable and commented form. In programs that aim at solving 'real world problems' one will hardly escape the need to go the third way — despite all intended flexibility and universality of our program. A substantial part of the functionality of the C+− classes was given its present form in an continued attempt to make this way as convenient and safe as possible. Especially this is true for the classes `Record` and `RecordHandler`, and the reading macros `cpmrh` and `cpmrhf` which proved their power in an engineering project in which several teams submitted their needs to extend the set of several thousand control parameters on a daily basis. Due to these achievements, this third way is highly efficient and convenient for C+− . Organizing the code in moderately sized translation units, reduces the inconvenience of re-compilation. However, re-compilation has also its convenience since, due to the simple structure of typical C+− statements, oversights are very unlikely to result in code that compiles. If an error happens at run time, it is not important, even not desirable, that the program continues somehow. It is very important, however, that it leaves a message on a log file that allows to identify the problem. This is often much more efficient than working with a debugger.

3 The C+− naming style

The advice of Bjarne Stroustrup ([8], p. 85, 4.10.6) is: Maintain a consistent naming style. These are the naming rules concerning the C+− project:

1. The header files of the project are named `cpm*.h` and the implementation files are named `cpm*.cpp`. They don't contain blanks or underscores.
2. All declarations contained in one of these files is placed in a namespace. All these namespaces have names beginning with `Cpm` and continuing with at least one upper case letter. A list of all namespaces may be instructive:

```
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
```

Here the number at the end of some names refers to the dimension of space. Although space is 3-dimensional, a 2-dimensional surrogate 'flatland' is often useful in physical models for faster execution and easier visualization. The namespace `CpmCamera3` deals with generating anaglyph stereo images of spatial physical structures.

3. All preprocessor defines (macros) begin with `CPM_` with all letters upper case. The macros used as double inclusion guard are the file name with `.h` replaced by `_H_` or `_H`.
4. Class names (which, as stated above, are all in namespaces) never begin with `Cpm`. They always begin with an upper case letter. Some structs, which also are all in namespaces, have names which are lowercase as `rgb`, `xy`, and `xyxy` in namespace `CpmGraphics`. No rules for selecting type names are given. Preference is on short names, more in the style used in mathematics than in programming. Best example: complex numbers have to be named `C` and not `Complex` or even `ComplexNumbers`.
5. Data members have not to be public, except of the numerical tuple - classes such as `R2`, `R3`. Their names should end in an underscore [11]. This rule is followed only in the more recent classes.
6. The naming of public member functions is governed by strict rules to be described in the next subsection.

3.1 Naming rules for member functions

In cases where a well introduced name for the concept exists in C++ or the pertinent discipline, we use this name. (E.g. `atan` for *arctan* or `gcd` for *greatest common divisor*.) In all other cases we use an identifier which is derived from a reasonable *primary full name* [12] by a deterministic rule. The primary full name has to be clearly stated in the comment to its declaration, preferably in a single line directly following this declaration, preferably not simply preceded by `//`, but by `//:` . In addition to `//:`, we rarely use `//.` and `//::` to mark exceptionally short and long names, respectively. Also, the comment indicators `//?` and `//doc` occur to assist script processing. The primary full name may consist of several words, e.g.

[11] It is very convenient to have a typographic indicator for data members (as in Ruby) since implementation code then can use short identifiers without any danger of interfering with data members.
[12] in some places also referred to as *descriptive full name*

`right upper corner` .

The corresponding C+− name is formed by a simple set of rules that allow the fluent usage of the name if the primary full name is memorized correctly. These rules are:

1. Process (contract) the individual words first. Hyphens and punctuation marks are considered as blanks for this purpose. So 'right-upper corner' and 'right upper corner' both are treated as the list 'right, upper, corner' of words.
2. If a word is not longer than 4 letters: don't change it. Else, take the first two letters as they are and append as a third letter the first following consonant if there is one (notice that we are considering a word with 5 or more letters) or the next letter else (which then would be a vowel, this seems never to happen with words of plain English).
3. Concatenate the processed individual words and turn all letters into lower case except the beginning of the second, third, . . . processed word.

This transforms our `right upper corner` into `rigUppCor`. This is not a favorable example; in most cases the name comes out nicer such as `rot` for `rotation`, `absVal` for `absolute value`, and `sqr` for `square`.

Addition: Sometimes the standard symbol for a physical quantity, such as `t` for time, should occur in a function name: the intended full name of such a function could be `get t` or `set t`. Applying the previous rules would lead to `getT` and `setT` which would be misleading since one could think that the function is dealing with getting and setting the absolute temperature `T`. So we write `get_t` and `set_t`; thus the rule is to precede each lower case one letter symbol by an underscore. A function that sets x-coordinate `x` and y-coordinate `y` could be named in full text `set x and y` and would lead to `set_xAnd_y`, which looks ugly. So it is preferable to consider `set x,y` as the appropriate full name. This then goes to `set_x_y`, which looks good.

4 Member function declarations

Many declarations that should have exactly the same form for many classes are formulated using declaration macros. The most frequently used such macros are `CPM_ORDER` and `CPM_IO` which supply the declarations and also some in-lined implementation of order-related functions and functions concerning interaction with input- and output-streams. Such declaration macros are very similar to the *mixin* construct of the language Ruby. Here is an example

```
// 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;}
```

Reasons for writing `Type const&` instead of the more common `const Type&` are given in [10], pp. 3-4.

For C+− -classes it is normal to have a part of the interface defined by macros. We thus may see a class declaration like this:

```
#include <cpminterfaces.h>
class X{
   ...
   typedef X Type;
public:
   CPM_IO
   CPM_ORDER
   CPM_SUM_C
   ...
};
```

Notice that an interface in which `Type` appears as a return value of a function cannot be directly coded by deriving from a suitable abstract class. Only a combination of abstract classes and templates allows a macro-less definition. The combination of interfaces has then to be done by multiple inheritance and virtual functions had to be defined even for lean workhorse classes. This convinced me that in this case a solution based on preprocessing is superior to the solutions provided by C++ language means. It should be mentioned that the usage promoted here refers to macros without parameters and thus avoids some of the pitfalls associated with macros.

It is highly desirable that the basic logical structure of code should be understandable without having to look up the declarations of the functions involved. Since the types of function arguments are obvious from the code, the question is mainly if calling a member function changes the calling instance or any additional function argument. In C+− , the name of a non-constant member function ends in an underscore and virtually always it does not change arguments (which in these cases are declared as constant references as mentioned earlier). In the case of exceptions, an underscore followed by a number appended to the function name indicates the number of an argument slot which is typed as a non-constant reference. Thus the following declaration (which also demonstrates the preferred style for comments)

```
// Some demo classes
   class Y; // class X uses Y (and not only Y&)
   class X { // demonstrating function naming
      ...
      Y doTooMuch_1_2_(A&, B&, X const&);
         //: do too much
         // Example that demonstrates the influence of the
         // argument structure on the function name. Notice the format
         // for communicating the primary full name of the function.
      ...
   };

   class Y { // demonstrating ...
   };
```

agrees with the rules since function arguments 1 and 2 are non-constant references and the final underscore says that the function itself is non-constant. Seeing a call

```
   y=x.doTooMuch_1_2_(a,b,x1);
```

we know without looking to the declaration of function `X::doTooMuch_1_2_` that `y`, `x`, `a`, and `b` are intended to change their values by this call, whereas `x1` is invariant.

It is to be admitted, that in the older parts of the C+− class system the transition to the most recent version of the naming rules has not been completed so far.

It was stated that C+− functions very rarely take arguments as non-constant references. One could see a problem here, since it is a very common programming situation that a function is expected to output various related objects at once: think of the three ‘components’ making up the ‘singular value decomposition’ of a matrix. The C+− way to do this is to make use of tuple class templates `T2<>, T3<>` and the Cartesian product class templates `X2<,>, X3<,,>` to create the multiple output as a return value. E.g.

```
R_Matrix a=...;
T3<R_Matrix> res = sinValDec(a);
R_Matrix u=res.c1();
R_Matrix diag=res.c2();
R_Matrix v=res.c3();
```

or, if we like to represent the diagonal matrix by the vector of its diagonal elements (i.e. its singular values):

```
R_Matrix a=...;
X3<R_Matrix,R_Vector,R_Matrix> res = sinValDecMod(a);
R_Matrix u=res.c1();
R_Vector sv=res.c2();
R_Matrix v=res.c3();
```

Whenever we seem to be forced to define functions with a complicated structure of arguments and return value we probably would better define a class. This provides a clear separation into the main computations to be done upon calling the constructor and cheap access functions to return the partial results of the main computation. With the actual C+− implementation of singular value decomposition (see `cpmrmatrix.h`) the previous calls read

```
using namespace CpmArrays;
R_Matrix a=...;
SVD svd(a); // does the work by calling the constructor of class SVD
R_Matrix u=svd.getU();
R_Matrix diag=svd.getW();
R_Vector sv=svd.get_w();
// Both diag and sv are cheap. It depends on later usage which of
// these two data formats fits best.
R_Matrix v=svd.getV();
```

5 Freely available

The reader who considers using C+− classes in his own work should also glance over [16]. The complete system of classes can be downloaded as source code (in zipped form) from [19]. Listings, project code with makefiles,an also be downloaded from the various links on the C+− related part of my homepage [15]. Since I do all my C++ programming on Linux/Ubuntu/g++ I do no longer provide Windows binaries. I'm open to provide help via email to those who try using C+− as a tool.

References

[1] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico: Data Abstraction and Object-Oriented Programming in C++. Wiley 1990

[2] Margaret A. Ellis, Bjarne Stroustrup: The Annotated C++ Reference Manual. AT &T 1990, Addison Wesley 1994

[3] Guido Buzzi-Ferraris: Scientific C++, Building Numerical Libraries the Object-Oriented Way. Addison Wesley 1993

[4] Bjarne Stroustrup: The Design and Evolution of C++ . Addison-Wesley 1994

[5] Mark Nelson: C++ Programmer's Guide to the STANDARD TEMPLATE LIBRARY. IDG books 1995

[6] Bruce Eckel: Thinking in C++. Prentice Hall 1995

[7] David R. Musser and Atul Saini: STL Tutorial and Reference Guide. Addison-Wesley 1996

[8] Bjarne Stroustrup: The C++ Programming Language, Third Edition. Addison-Wesley 1997

[9] Andrew Koenig and Barbara Moo: Ruminations on C++ . Addison-Wesley 1997

[10] David Vandervoorde and Nicolai M. Josuttis: C++ Templates. Addison-Wesley 2003

[11] Dave Thomas et al.: Programming Ruby. The Pragmatic Bookshelf 2005

[12] Ulrich Mutze: An asynchronous leap-frog method.
Mathematical Physics Preprint Archive 2008–197
`http://www.ma.utexas.edu/mp_arc/c/08/08-197.pdf`

[13] Ulrich Mutze: Arbitrary precision arithmetic as an optional and easily reversible replacement of Float arithmetic.
Linear algebra, function visualization, and dynamical systems as examples of 'precision independent' coding.
`https://rubyforge.org/projects/appmath/`

[14] Pavel Holoborodko: MPFR C++
`http://www.holoborodko.com/pavel/`

[15] Homepage of Ulrich Mutze
`http://www.ulrichmutze.de`

[16] The C+− Tutorial
`http://www.ulrichmutze.de/softwaredescriptions/tut.pdf`

[17] The C+− Listing
http://www.ulrichmutze.de/softwaredescriptions/cpmlisting.pdf

[18] code of C+− project chebyshev
http://www.ulrichmutze.de/sourcecode/chebyshev.zip

[19] All C+− code
http://www.ulrichmutze.de/sourcecode/cpmsource.zip