Documentation of the Numerical classes for Plasma simulation

Kurt Garloff
<kurt@garloff.de>

14th November 2026

Abstract

C++ classes have been written by Kurt Garloff, Attila M. Bilgic, Andreas Ahland and others to serve their needs when numerically simulating microwave generated plasmas (MIPs). The classes however are general and powerful enough to serve more than this. In order to make it possible for other people to use them, this piece of documentation has been written.

The classes described here provide can be divided into two categories: On the one hand, there are classes to provide basic classes for linear algebra, i.e. complex numbers, vectors, matrices, tensors, ..., some optimized versions for special cases of those and solvers. These are very designed in a way to work very efficiently. On the other hand there are classes constructed on top of these which implement things you need for solving partial differential equations on an arbitrary grid, such as differential operators, grids, metric tensors, stretching functions, geometry description, ...

This document describes some design considerations, compiler needs, features of the classes and interface details.

Conventions: C or C++ keywords or code examples are displayed in sans serif font, and functions are additionally marked by (). Filenames are typeset in a `typewriter font`. Important notions are emphasized with *italic type*.

Contents

1 Design, Compiler and Makefile considerations **3**
1.1 Why C++ 3

1.2 Design . . . 6
1.3 Performance considerations . . . 7
1.3.1 C++ can be slow . . . 7
1.3.2 Ways to improve C++ classes . . . 8
1.4 The Temporary Base Class Idiom (TBCI) . . . 9
1.5 More optimizations . . . 11
1.5.1 constness and (const-) references . . . 11
1.5.2 Loop unrolling . . . 11
1.5.3 inline functions . . . 12
1.5.4 Double indirection instead of multiplication . . . 12
1.5.5 register variables . . . 12
1.5.6 Own memory management . . . 12
1.5.7 Compiler and architecture specialities . . . 13
1.5.8 Initialization . . . 13
1.5.9 Caching . . . 14
1.6 Defered operations . . . 14
1.7 Parallelization . . . 15
1.8 Algorithms . . . 16
1.9 Makefiles . . . 17
1.10 Compiler . . . 18
1.11 CVS . . . 18

2 Directory structure **19**
2.1 The lina tree . . . 20
2.2 Overview of solvers in lina/include/solver . . . 21
2.3 The mpt tree . . . 22
2.4 The bench directory . . . 22
2.5 The grid directory . . . 23
2.6 The doc directory . . . 23

3 The files and classes in detail **24**
3.1 Files in lina/include . . . 24
3.1.1 constants.h, basics.h and except.h . . . 24
3.1.2 smp.h . . . 25
3.1.3 cplx.h . . . 25
3.1.4 bvector.h and vector.h . . . 26
3.1.5 matrix.h . . . 27
3.1.6 band_matrix.h . . . 28
3.1.7 tensor.h . . . 28
3.1.8 my_nr.h and mathplus.h . . . 29
3.2 The directory lina/include/solver/ . . . 29

3.2.1 gauss_jordan.h and lu_solver.h 29
3.2.2 bd_lu_solver.h 30
3.2.3 Iterative solvers 30
3.2.4 Preconditioners 30
3.3 The files in lina/source 30
3.4 Files in mpt/include 30
3.5 Files in mpt/source 30
3.6 Files in mpt/plasma_coax 30
3.7 Files in grid 30

4 Acknowledgements **30**

1 Design, Compiler and Makefile considerations

1.1 Why C++ ?

As will later be shown (chap. 1.3.1), C++ and other object oriented languages[1] can be slow, when doing numerics. There are a couple of ways out of this problem, but these require some effort. So the natural question to be asked is: Why using C++ and not plain C or Fortran?

The answer is: Because it's object-oriented. And because it's close to C which is close enough to the machine code to be efficiently translated into machine instructions by the compiler. It seems cryptic expressions like `*c++ = a<5? a: 10-a;` allowed in C are something really fascinating for a lot of programmers and the majority of programs nowadays is written in C or C++. Thus, a large number of libraries and tools is available which facilitate the development of programs in C/C++.

Object-Oriented Programming (OOP) There are several ways to describe what object-oriented programming is like. There's no intention to give a full description here, there are heavy books written on this, but the reader should get an idea and see, why it's attractive and advantageous to use C++, and understand some of the concepts of the language.

In non object oriented languages, there are data structures and algorithms acting on these. In an object-oriented language, *the data and the methods* to act on the data reflecting the properties of the data are put together into a `class`. This `class` contains everything that is needed to access and to modify

[1] Why do so many people still use Fortran77 instead of 90 or 95?

the data in a way the properties of the represented *object* suggest.[2] The internals are hidden from the user of this class. So the class consists of data and functions accessible (public) from outside (the *interface*) and of internal (protected or private) data and functions not directly accessible from outside (= the *implementation*).

The class itsself is something abstract defining the way things are and not really something the compiler produces code for in the first place. It's like a law regulating what treatises are and what they imply, but nobody cares unless you make a treaty. It's just a collection of data structures and functions. Now, if the user wants to use class X, she creates (*instantiates*) an object obj of class X by writing X obj; just like one creates (declares) an integer number by using int num;. Now there's something real: The object (= variable) obj. The compiler now produces code for its member functions.

When the program containing the obj is called, a so called constructor is called to do some initialization at the place where the declaration gets effective. constructors can also have arguments to specify how it should be initialized. One of the most important ones is the constructor which has an argument of its own type to make a copy from it, the so called copy constructor. It's called when you create an object b like this: complex a (3.2,-1.9); complex b(a); complex c = a; Here, a is created by a constructor accepting two floats or doubles. For b, we tell the compiler to use a copy constructor, while for c the default constructor is used and the assignment operator = is called. However, as copy constructors are slightly more efficient, you can tell the compiler to use it instead.[3] When a object is destroyed, because its frame of reference is left, a destructor is called to do the cleaning up.

A good example are complex numbers. The class contains data to store the real and imaginary part of the number and methods to modify them, such as complex addition, multiplication, trigonometrics, ... One nice thing in C++ is, that the *operators* like +, * can also be redefined, so the user does not have to write something like cplxmul (&res, op1, op2); but just uses res = op1 * op2; Of course, the operator * has to be defined in the complex class, provided op1 and op2 are objects of type complex. This is more readable and less error prone than the C-style syntax, as it allows to write expressions just like the mathematicians do on paper. Matrices can be multiplied this way, etc. The compiler knows what type the objects are, so it can select the right operation. Redefining operators is called *overloading*.

[2] In (Borland) Pascal an object is what is called a class in C++. With C++ the word object is rather used for an instantiation of a class, which better matches the normal imagination of an object being something real.

[3] This is what the `-felide-constructors` compiler switch for the GNU-CC does.

As the program has access only to the interface methods, but not to the internal structures of an object it cannot rely on the latter. Now, as the implementation of this is hidden to the user of this class, the programmer can change it, e.g. make optimizations without the user even noticing this. No program has to be rewritten as long as the interface doesn't change. With complex numbers you can imagine to store the absolute value and the phase of the number instead of real and imaginary part, which might be useful if you take a lot of multiplications. Or maybe both, but only updating the easier and/or the needed representation.

Another nice feature of C++ are *templates*. Can you imagine to write an algorithm in C working on both plain floating point numbers and complex numbers using the same piece of source code? You probably can't. (OK, you can: With heavy usage of macros and preprocessor statements, but his will result in very ugly and unreadable code.) With C++ it's easy: You create a `template <typename T> algorithm (T input1, const T& input2);` and you are done. When you use it, you write `algorithm<double> (a, b)` (a and b must be `double`s) and the algorithm will work on double numbers. You can do the same with complex numbers, as long as the operations used within the algorithm are defined for complex numbers. Of course, the compiler has to know what type an object has, when it produces code, that's why you have to specify `<double>`. Now in this case, it could have concluded from the type of the arguments, and it actually would.

If you have a couple of functions doing almost the same, but some are special cases of others, you might be happy to learn that C++ provides *default arguments*. So a constructor for complex numbers can be declared like this: `complex (const double r, const double i=0);` and this same constructor can be called with one or with two arguments. If you only give one, the second will be taken to be zero.

As the implementation of a certain piece of code can be changed anytime, it gets less important. The real important thing with C++ is the definition of the interfaces. If a lot of developers work on a large software project, they have to start asking themselves what data from whom they need and *design the interfaces* to provide this information. Once they agreed on it, they can work on the implementation without needing much interaction.

The features described make it easier to write code which can be reused for another purpose. Another feature helps doing this: *Inheritance*.[4] Classes

[4]When I first met OOP, the text I was reading considered inheritance as the main feature of OOP. It used animals as example. This resulted in delaying my understanding of why it should be any useful. IMHO, complex numbers, vectors or other mathematical objects are a much better example. And I dare to doubt that the inventors of OOP had animals in mind.

can be based on other classes. They inherit the data and methods from the *base class* and can add or replace some of the functions. The feature of child classes being able to redefine functions is called *polymorphism*. However no data or functions can be removed. This way, an object of type child is still also an object of type parent, which doesn't prevent it from behaving differently in some situations. And, of course, its abilities can surpass the one of its parent.

So if there are a couple of classes with some common features, it is a good idea to implement the common features in a common base class and to make the classes children of this base class. Also, if they only share a common interface, it's still useful to use a (then called virtual) base class. Large class hierarchies can be created this way and are in real life. If you use a Graphical User Interface (GUI), most probably the underlying code is structured in class hierarchy as the buttons, sliders, windows and checkboxes have some common features, as e.g. they have to react somehow on being clicked on with the mouse.

On the other hand, classes can also contain objects of other classes just as they can contain data of the elementary data types such as char* or unsigned long. This is called *composition*.

There are a few advanced features such as *exception handling* and (related) *Run Time Type Identification*, which won't be discussed here, cause they are not needed to understand the library design. There are also some exceptions to the rule, that the compiler has to know the type of objects at compile time. If the base class is known, derived classes can be handled via virtual functions, and there are function tables from which the appropriate function is chosen at run time. Furthermore, for the management of very large projects, the C++ language provides *namespaces* to prevent name conflicts.

The latest C++ standard does not only define the language features described above, but also some standard libraries for I/O (iostream), for Containers (STL), complex numbers (complex) and some other useful stuff. For details see Stroustrup (3rd edition) [Str97] or Josuttis [Jos96].

You may be able to image that large software projects really profit from the structure that C++ offers. As you defined interfaces, people can easily work together and make changes without causing harm to a different group working on another part of the same software project.

1.2 Design

Note that the header files not only contain the declaration but also the definition (implementation) of the operations. This is especially convenient for

the programmer when templated C++- classes are used. Of course all header files are protected against multiple inclusion by an #ifdef _FILE_H #define _FILE_H ... #endif mechanism.

All classes are templated to be able to operate on various data types. This is a very convenient feature of C++, because it's not necessary to define operations on every data type separately but just once. The matrix class e.g. is declared template <typename T> class Matrix;

This allows you to use e.g. matrices with real or with complex data elements, which are declared as follows: Matrix <float> a; Matrix <cplx <double> > b;.[5] In the following examples the data type however is omitted for better readabiliy. Imagine e.g. a <double> behind every Matrix.

Any object of a class has its own data, i.e. if a Matrix is assigned to another, the data is copied and belongs exclusively to the variable refering to it. This is necessary to have it behave like ordinary numbers in C, so you can change them without destroying the contents of another variable. Some issues to prevent too much copying are discussed in 1.4. However, the user of the class does not have to consider such issues, only the library designer.

1.3 Performance considerations

Parts of the code are designed to use the Temporary Base Class Idiom as proposed by Spanderen and Xylander [SX96] in order to avoid superfluous duplication of data when performing arithmetic operations with large objects.

1.3.1 C++ can be slow

It's easy to see that the redundant use of a temporary involves a considerable overhead: An object to hold a result must be created (i.e. a *constructor* is called which allocates a (large) chunk of memory and makes some initializations), the object is copied (*copy constructor*) when it's returned and afterwards removed (i.e. the *destructor* is called where the allocated memory is freed, normally). On large objects, this is the main reason for the poor performance of C++(and other object-oriented languages) compared to traditional programming languages such as Fortran77 or ANSI C.

Let's consider the following situation: Matrix a(100), b(100), c(100), d(100); creates four 100x100 matrices. Now a calculation of the form a = b + c + d; is performed. The compiler translates this into a.operator = (b.operator +

[5] The complex class provided by this library are called cplx and is also templated to allow the user choosing single or double precision floating point numbers (or even integers, which rarely makes sense).

(c)).operator + (d); This is evaluated the following way: a.operator = (Matrix temp1 (Matrix temp2 (b.operator + (c))).operator + (d)); There are two redundant temporaries temp1 and temp2 involved in this expression.

With most compilers, even create more temporaries are created: Within the definition of Matrix operator + (const Matrix&) there is a Matrix res which is created to store the return value of the + operation. When returning from the operation, this Matrix is copied (via copy constructor) into the tempX value. In the worst case four temporaries are created. However good compilers[6] are able remove two of them.[7]

Two temporaries are still too much. Temporaries not only take time for allocation, copying and deallocation but also consume a large amount of memory given that the objects are large. If one uses a straight forward implementation of the numerical classes, the whole contents (i.e. all elements) of the matrix is being duplicated for each temporary.

The code could be easily transformed to read a = b; a += c; a += d; and yield the same result without any temporary and only one copy constructor. In many cases it is possible to find an optimal solution manually and only introducing as many temporaries as are needed. However this is less convenient. The user expects the compiler to do such work for him. Moreover, the code would become harder to read and therefore more prone to errors. One advantage of C++ is to allow human readable expressions and we'd better design the library to circumvent performance hits.

1.3.2 Ways to improve C++ classes

There are several solutions for the problems described above. One very promising way is to use Expression Templates / Template Metaprogramming[8] as described by Todd Veldhuizen to have the compiler generate parse trees after the rules given by the user and transform the above expression into the optimal form.

This technique is being heavily used in the Blitz++ library[9]. However this library is not fully functional yet and requires a very recent compiler.[10] Also compile time and memory needs (during compilation) are very high. As compilation time makes a considerable factor during the development of our

[6] The KAI C++ is known to be very strong at this point

[7] See discussion of named return values in the gcc-manual, 7.1 / egcs-manual, 5.1

[8] http://monet.uwaterloo.ca/~tveldhui/papers.html

[9] ftp://monet.uwaterloo.ca/blitz

[10] At the time of writing, the KAI++ is the only one known to me to compile all of the Blitz++ code. The egcs-Compiler already is able to compile large parts of it and is getting better every week.

numerical applications (mostly testing), we decided against Blitz++.

Another solution consists in reducing the overhead needed for allocating, copying and deallocating temporaries. If creating a temporary only needs to allocate (and later deallocate) space for some pointers on the stack and only needs to copy these, not much time is lost. We can, for example, define the copy constructor in a way not to copy the data elements but just some pointers pointing to it. So we have data shared by more than one object, which is called aliasing. We have to keep track of the objects pointing to the data and to copy them when they are changed and if more than one objects refers to them. This technique is known as Reference Counting.[11] The problem is that this involves some overhead at run-time: We have to protect the data against write access and make a private copy, if data is changed. For Operating Systems this can be done very efficiently by using the CPU's features provided for that, but for userspace applications a more complicated approach has to be taken.

A similar approach is the Temporary Base Class Idiom which is described in more detail in the next section, cause it is what is used inside our classes.

1.4 The Temporary Base Class Idiom (TBCI)

The idea is the following: Each class which is designed to consist of large objects is represented by two classes. One Class (C) for the representation of the objects and a Base Class which is meant for temporaries(TBC).

The user only interfaces the C objects. The TBC objects are only used for intermediate representation. The main difference is the copying behaviour. When TBC or C objects are created as a copy of C object, a real copy is performed, i.e. all data elements are copied. When a TBC or C object is created as a copy of a TBC object, only a few pointers are copied, but not the data itself. This requires the destructor of the TBC class – unlike the one of the C class – *not* to destroy the data.

All operations (+, -, *, /, ...) create and return objects of the TBC. Once created, further computation with the TBC objects can be done without having to make redundant copies of its elements. The library designer knows, that he's dealing with a temporary representation of this object, which is not refered to by any user-visible variable, so he can manipulate it in further operations instead of creating new objects, i.e. he can e.g. apply operator += instead of operator +.

As soon as the user assigns the result of all these operations to a real object, i.e. a object of type C, this object takes possession of the data of the

[11] Sorry, no good reference found yet.

```
template <typename T>
TVector<T> Vector<T>::operator + (const Vector<T>& v) const
{
  // Note: Check, whether size() = v.size() belongs here ...
  TVector<T> res (size());            // TBC object in created
  for (unsigned i = 0; i < size(); i++)
     res.set ((*this)(i) + v(i), i);  // res.operator ()
                                      //  would destroy res
  return res;       // TBC copy constructor is called
                    //  and res ist destroyed
};
```

Figure 1: Example implementation for the addition of two Vectors using the TBCI technique. The addition of course has two arguments, but one is taken to be *this.

TBC object just by copying a few pointers.

This way we can evaluate the expression a = b + c + d; with just one real copy operation. We are able to prevent almost any speed penalty because of temporaries with this technique.

As the return values of the operators are of the TBC type, the problem with additional copy constructors (when the values are returned from the function body of the operator) are prevented. Of course you cannot prevent the compiler from calling it, but copy constructors of the TBC class are very cheap, because they don't copy the data.

The result of the additional effort is a substantial speed increase. Spanderen and Xylander report a factor 3 which also matches the experience of the author of this docu. By the time of writing the TBCI mechanism were implemented in the Matrix and the Vector class. TBC objects are of type TMatrix resp. TVector there.

However you have to take care for two things when using a library designed with the TBCI. You should *never* instantiate an object of the TBC[12] and never ignore the value of an TBC computation. In the first case you would risk to have references to deallocated memory (which gives a nonsense result or a Segmentation Fault). Too bad, the TBC can't be declared private, because some solvers need access. In the second case memory leaks would

[12]If you want to make optimizations and give back a TBC object from your function, you should really know what you are doing. Note that a lot of operations on a TBC object simply result in destroying it.

be created because nobody deallocates the memory for these objects. Every TBC object must in the end be assigned to / copied to a C object. (Inside the library, care is taken to prevent memory holes: If one of the arguments of an operator or a function is a TBC object and isn't needed any longer, the care is taken of its destruction.)

If the library user does not use the TBC objects and does not ignore the results of operations, the use of the TBCI is fully transparent. When the implementation of the Vector and Matrix class was changed to use the TBCI, no single line of our programs using these classes had to be changed. (Another reason to like C++.)

1.5 More optimizations

1.5.1 constness and (const-) references

Whereever appropriate the keyword const is used to tell the compiler that a value won't be changed by a certain operation. This allows the compiler to check if we really don't change it and allows it to cache the value in a register e.g.

Operators operating on large objects take const references (e.g. Matrix operator + (const Matrix& m)) as arguments to avoid copying of large amount of data when being called.

Assignment operations such as =, +=, *= ... are implemented to give back a reference to itself, so that it is possible to write a = b += c; with matrices just like with doubles in C.[13]

1.5.2 Loop unrolling

Working with objects such as vectors and matrices typically involves a lot of loops to manipulate / copy these objects. Whenever the number of iterations of a loop is known, the compiler can unroll such loops[14] and save the overhead for the counter. This is especially true for small objects, e.g. threedimensional vectors. From this point of view the creation of a class template <typename T, unsigned dim> class vector; would have been the right choice, because the compiler then knows the dimension. This also enables the compiler to perform a better type checking.

However this is paid for with a loss of flexibility, because then the size of vectors and matrices has to be known at compile time already. We decided against such a design. Size checking is partially performed at run time, if

[13] The given expression is defined to be evaluated from right to left by ANSI C.

[14] GNU-CC: option `-funroll-loops`

the ERRCHECK directive was set during compile time. This is done by the BCHK(...) macro.

1.5.3 inline functions

For many small functions, the overhead of the calling procedure (pushing arguments on the stack, calling the function, returning and cleaning the stack) can be avoided by using inline functions. As in general this increases the size of the produced code (which has the disadvantage to get a worse cache hit rate) this is only recommended for small, fast functions.

1.5.4 Double indirection instead of multiplication

The Matrix class uses an array of pointers to the rows of the matrix instead of calculating the memory address by performing a multiplication. Especially on the Pent*um, where the multiplication of two integers takes 11 cycles, this saves time. However for small matrices and other CPUs it might be possible, that the overhead due to the management of the pointer array takes more time than it saves.

1.5.5 register variables

Using register variables can be a good hint for the compiler to optimize the program not to use memory but processor registers for some variables, which is faster.

1.5.6 Own memory management

An approach to further increase speed is to optimize the memory management for the objects used and use own allocators and deallocators. However this only gives a considerable advantage if the objects managed this way are all of the same size, to be known at compile time. This is not the case in our Vector and Matrix implementation.

Moreover, the memory management of modern Operating Systems such as Linux is already very elaborated. Allocating virtual memory does not really result in reserving a large amount of physical memory. Only when this memory is written to, the CPU will raise a page fault exception and the kernel will physically allocate the memory (and swap out other data if necessary).

1.5.7 Compiler and architecture specialities

Some compilers have special flags which can speed up operations. There are some general optimization flags such as `-O3` for the GNU-CC and some special flags which can change (break) the program behaviour for some code. For the GNU-CC we found the flags `-felide-constructors` and `-fnonnull-objects` to give a slightly better performance without any price to be paid for. Also `-funroll-loops` speeded up some operations, but caused the executable to grow in size.

On the i386 (and especially for Pent*um class computers) the alignment of doubles can be critical. Normally the compiler aligns doubles on 4 byte boundaries[15] which is suboptimal. One can achieve better alignment by using the __attribute__ ((aligned(x))) extension of the GNU compiler or by using the flag `-malign-double`. Note that the latter will be incompatible to libraries which contain structs with doubles.

The flag `-ffast-math` was found to be useful, too. This flag allows certain deviations from the IEEE754 standard to treat floating point numbers, which only hurts if you have operations producing denormals and other funny things. On the ix86 architecture this does not happen very often, whereas on the DEC-Alpha processor, it's easy to get a floating point exception (SIGFPE) as a result of an operation with denormalized numbers.[16]

For C++-code also `-fno-exceptions` and `-fno-rtti` are useful, if these C++-features are not used.

Modern CPUs are able to perform more than one operation in parallel, if these are not dependent on each other and don't need the same execution units on the CPU. The late egcs-compilers have knowledge about this CPU feature built in, and can schedule instructions to profit from this. This is turned on by `-fschedule-insns2`.

`-freduce-all-givs` and `-frerun-loop-opt` have also shown a positive influence on some applications.

1.5.8 Initialization

Whenever possible, one should refrain from initializing variables in the default constructor. Otherwise it would take considerable amount of time to create large arrays of objects of this type.

For allocating memory, the C++ syntax is to use the operator new. If

[15] This is specified in the Application Binary Interface (ABI) specification

[16] If you are performing calculations on the Alpha, resulting in bad numbers, you should prevent it. If you can't, you can force full IEEE compliance with the `-mieee` compiler flag. Note that this will decrease your performance by 10 to 20%.

you call X* ptr = new X[20];, twenty objects of type X are created using the default constructor to initalize them and ptr points to them. Now, if the default constructors do more than needed, time is lost. Note that you can enable the compile time switch C_MEMALLOC which uses the C functions for allocating and deallocation memory, i.e. malloc () or memalign () and free (). But take care: You will crash your program if containers for objects which need intitalization are created. You have been warned![17]

1.5.9 Caching

Some quantities are expensive to store and are thus calculated on the fly, when needed. This is true for Christoffel symbols, e.g. Storing their value at every grid point can easily consume some Megabytes. They are only needed during matrix setup, so they are only accessed a few times and then not needed any longer.

However, if for every grid point a Christoffel is needed only twice, it could be a good idea to maintain a small cache to hold the last few Christoffels calculated and to only calculate one, if it was not calculated before. Such caching strategy can improve performance and did result in 30% faster matrix setup in our code.

1.6 Defered operations

For some special operations we could execute more than one operation within one loop. A prominent example is the SAXPY-Operation (Scale aX plus Y) which is a multiplication by a scalar and the addition of a vector or a matrix. Doing it within one loop is definitely better than doing it in two. The number of arithmetic operations is the same, whether doing within one or two loops, but: For small objects, the overhead of the (second) loop header is too large and for large objects, which don't fit into the processor's cache, the cache gets filled with the values needed twice instead of only once, which takes some time. For such cases we could provide special functions, which can then be called by the user where approriate.

It's even possible to make the compiler call such a function automatically[18] whenever it finds an expression like Z = a*X + Y. We have to create extra classes that represent the result of a certain operation on two objects without actually performing it: The operation is defered and the operands are stored in the class (The defered operation class, DOC). When the next

[17] Of course, this feature would not be there, if it was only dangerous. Allocating memory with C-style functions is slightly more efficient.

[18] See Stroustrup [Str97], chapter 22.4.7 for details

operation is performed with this DOC we might be able to perform two operations in one loop. If the DOC is just assigned to a real object, the operation is performed then and no time is lost or won. However the programmer has to take care not to get a performance hit by copying the stored objects but just storing pointers. Note that it's not allowed to write an expression like `a = (a + b) + ((b+=c) + d);`. However as C++ does not guarantee the order in which this expression is evaluated, the result is unpredictable anyway.

The implementation of such a mechanism is very hard, because for every operation, we have to create an own class to represent the result. Of course, every allowed combination of operations between these classes have to be defined as well.

Currently, this technique is only applied for one special case: The multiplication of a scalar with Matrix or Vector creates a TSMatrix resp. TSVector whichs stores references to the operands for later execution of the operation. The restrictions are the same as for the TBC objects: Never ever create objects of such a type in your programs if you don't know every detail of their memory management! Don't ignore results of calculations!

The above speeds operations with multiplications / divisions which involve scalars and matrices, which occur quite often in certain applications, e.g. recursive matrix solvers (Krylov solvers). [B+94]

1.7 Parallelization

There is code in the library to provide a framework to parallelize your code. To be more exact: It helps to create multithreaded programs running on multiprocessor machines. This approach can easily be extended to be used on shared memory clusters, but for some applications the Message Passing approach, which is fundamentally different, is considered more powerful.

The approach taken within the library uses POSIX threads. These are well standardized and are supported on a lot of Un*x-like machines. However, as usual, Win32 does not comply to the standard and is not supported by this. A thread is something like a lightweight child process, which shares the data (and some other things) with its parent. Such threads can be implemented by the POSIX library in two different ways. Either they use the Operating System (OS) to support it and process-like objects visible to the OS are created. These are called kernel-level threads. As the OS handles these, it can assign different CPUs to different threads of a process, which is what we want. However the overhead by going through the OS consumes some time (context switch, etc.), and therefore there are also implementations which do all in userspace and don't even make the OS notice that the program is multithreaded (= user-level threads). However, with the latter implementation,

no advantage is taken from having multiple CPUs, and it's really pointless to use multithreading for speeding up numerics with user-level threads. Fortunately, most implementations do (also) support kernel-level threads, at least the ones on Linux and Digital-Unix do.

On startup, all the subthreads are created and initialized by the init_threads (); function (declared in `lina/include/smp.h`), but then block on (= sleep without consuming CPU cycles) waiting for a job to be submitted to them. Now, if there's something to be done, they are woken up, do their job and signal the main thread when they are done. On program exit, the threads are destroyed by free_threads (); or by the clean up functions of your Operating system. Note, that for operations taking a very long time, subthreads could be created on the fly, but normally this takes too long, that's why it's done on initialization of the program.

Now, if there is a piece of code which can be parallelized, this piece should not be too short (in time). There is some overhead involved in passing the data to the thread which performs the operations. This time is in the order 0.1 ms or a little bit better, which means that operations taking only 1μs to complete are not good candidates for parallelization. The longer the task which is submitted to a subthread the better. As communication only occurs at the start and the end of a job, its overhead will account less compared with the job.

In the numerical library, currently only the Matrix-Vector multiplication is parallelized. For large matrices almost the double performance is reached on a two processor machine. Unfortunately, machines with more CPUs were not available to the author.

Note that if you want to use massively parallel algorithms, custom machines might be better than general purpose computers. Current general purpose architectures are restricted to $1 \ldots 16$ processors per machine. Now clustering them together via some network needs very fast connection between them. Note that the TCP/IP stack can have a high throughput on a 100MBit network, but latency is much too high. Special communication means are necessary.

If some percent of the code can *not* be done in parallel, say $\alpha = 0.10$ (10%), the maximum performance which can be reached with an unlimited number of processors is a factor of $\frac{1}{\alpha} = 10$. If communication overhead is considered, the maximum performance drops further.

1.8 Algorithms

It should be noted that choosing a better algorithm can do more improvement for some cases than any programming skills may be able to achieve. With the

framework of C++-Classes, it is easy to reuse such algorithms, if the classes share a common interface and provide the information, the algorithm needs. Details have been discussed in [Fur98].

1.9 Makefiles

Whereever objects, libraries or binaries are created, there are directories `bin-`*`arch`*, where *`arch`* describes the architecture and is replaced by `ix86` or `alpha` for example. This is done to make it possible to share the tree between computers of different architecture via distributed filesystems such as NFS. The architecture name is determined by `uname -m` translated into lower case and some replacements (eg. `i?86->ix86`) occur.

All Makefiles should include the standard definitions in `Make.Std` which itself includes `Make.Sys`. For this to work, the Makefile variable MAKEINC should be set to the root path of the numerics directory where these files reside. This e.g. results in the following lines in a Makefile:

```
MAKEINC = ..
include $(MAKEINC)/Make.Std
```

This helps to keep the Makefiles short and sets some standard variables such as e.g. SYSDEP which gets set to the `bin-`*`arch`* name.

Most Makefiles support at least the following targets: static, shared and clean. The default is shared, which means that a shared library or a binary linked to the shared lib is created. When interchanging programs it can be convenient to use the static target. As the linker by default prefers to link against shared libraries (.so suffix on Un*x systems, .DLL on Wind*ws), remove them before linking. clean just deletes all created object, library and/or binary files.

Note that you have to set the shell variable LD_LIBRARY_PATH to the directories containing the libraries in order to make the dynamic linker find them. On our machine, this is done by `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/plasma/numerix/lina/bin-ix86:/home/plasma/numerix/mpt/bin-ix86`. This statement can be added to the user's `~/.bashrc`. You can also make it system-wide default by adding the directories to `/etc/ld.so.conf`, if you have the right to do so (i.e. the root password).

The Makefiles are designed in a way that the user can override options by specifying variable names, e.g. CC, CXX, CFLAGS, CXXFLAGS ... and specify additional options via some other variables, such as EXTRA_CFLAGS, EXTRA_LIBS. Options can be set via defining DEBUG, DEBUGINFO, ERRCHECK, EXCEPT, ABORT_ON_ERR, OPT and SMP, meaning to emit debug info into the binary for a debugger, enabling the INFO() macro meant for

debugging by printing messages at certain checkpoints, doing some run-time sanity checks (e.g. bound checking for Vectors), generating C++-exceptions (instead of printing messages to stderr), raise SIGABRT on such a condition (which by default terminates the program), enabling some extra compiler flags for optimization and enabling multithreading support, respectively. An alias for ERRCHECK is BOUNDCHECK.

1.10 Compiler

The code requires some of the recent features of the C++ language. The code was first developed using the pentium-gcc[19] snapshots and later using snapshots and release versions of the egcs[20] Compiler. Both are based on snapshots of the FSF's GNU-CC (gcc). The code also works with recent versions (2.95) of the FSF-gcc[21]. We found the egcs-Compiler best complying to the draft C++ standard[22]. Most of the code also works with the CXX-Compiler (5.6x) from Digital, and the most recent version (6.0) is expected to fully work, which could not be tested, though.

Recent changes to workaround some deviations of M*crosoft's Visual C++ compiler from the standard allow most of the code to be also compiled by this compiler under Win32. The main problem with MSVC++ is its brain-deadness with respect to the declaration of variables in loop headers.

1.11 CVS

When several developers work on the same piece of code, often changes done by some of them are on the same file. This leads to conflicts, as none of the two versions of the file contains all fixes. So these changes have to be merged manually or with the help of the Un*x tools `diff` and `patch`. But often conflicts are not recognized and work is lost that way, which can make programmers very angry.

This can be avoided by assigning the job "maintainer" to one of the involved people and every change to the "maintained" source tree has to go via him/her. This is a nasty job to do, and can only be done correctly, if only `diffs` (i.e. files containing a machine readable description of the changes) are accepted. But tell those Win32 developers to use a command line tool like `diff`! In the end the "maintainer" has a lot of extra work, but the others

[19] http://www.goof.com/pcg
[20] http://www.cygnus.com/egcs
[21] ftp://vger.rutgers.edu/pub/gcc
[22] http://www.maths.warwick.ac.uk/cpp/pub/

won't like her or him for dictating rules like using `diff`. The effect of all: Nobody wants to do the job.

As the integration of changes from other developers cannot be done without spending some time, the merging of changes does not happen very often. The result is, that the official source tree is never up to date.

To avoid this, our whole source code is put under `CVS` (=Concurrent Versions System), a tool to manage different versions of (source)code. It keeps a history of changes to allow the user to review these. It also has means to merge changes done by several developers on the same file simultaneously. It can resolve changes done on the same file, unless they were done to the same lines. However, it has no intelligence to ensure that the changes don't imply logical inconsistencies.[23] So, it does not replace communication between developers ...

For achieving this, it has an archive, called *repository*, which is stored on one machine. Copies of the files in the *repository* can be retrieved locally or via network. Developers work on these copies and as soon as they think they finished a task, they send their changes to the *repository*. This operation is called *commit*. If somebody else commited changes to the same file before, an *update* operation has to be done before, but local changes are *not* lost when updating the files.

To summarize: `CVS` avoids a lot of conflicts between developers and ensures that changes made by somebody are accessible to the others. It allows a flat hierarchy between the developers, as nobody has to play the master of all. The person formerly known to be the maintainer only has to convince people to use `CVS`, which can still be a hard job to do.[24] The maintenance of the central source tree is done by a program, so the developers only have to communicate about the structure, the design and the implementation of the problem.

2 Directory structure

There are several trees in the `numerix` directory:

- The main directory itsself contains a master `Makefile` and some definitions to be included into all Makefiles: `Make.Std` and `Make.Sys`.

[23] This is impossible, until Artificial Intelligence takes over the programmer's jobs. This will – fortunately – probably never happen.

[24] `CVS` is available on almost any platform, but it's a command-line tool and unfortunately, some programmers nowadays are used to those fancy GUIs. There are IDEs (Integrated Developer Environments) which have support for `CVS`, such as `WipeOut` or `XEmacs`, but these don't exist (yet?) on Win32.

- The `lina`[25] tree containing mathematical objects such as complex numbers cplx, Vector, Matrix or Tensor and optimized versions for special cases (BdMatrix, CRMatrix , ...,) and a lot of basic operations with them. There is also a `solver/` subdirectory containing a collection of direct and iterative solvers and an `interface` directory containing definitions for interfacing with *LAPACK*. Basic definitions (preprocessor directives, macros ...`basics.h`), framework for multiprocessing (`smp.h`) and some other useful stuff also go here.

- The `mpt` tree containing (old) code for the simulation of electrical fields in the MPT.

- The `bench` tree containing programs to test library efficiency.

- The `grid` tree containing code to implement stretch functions, metric tensors, differential operators and other things needed to solve Partial Differential Equations (PDEs) on a general grid. It also contains the current code for the simulation of microwaves in coaxial waveguides and in the MPT.

- Last but not least the `doc` tree containing documentation.

2.1 The lina tree

The `lina` directory contains

- a `Makefile` to create the libraries. The version number is taken from the file `Version`.

- the `bin-`*`arch`* directories

- the `include` directory where most of the headers for the classes are.

- the `include/solver` directory contains the templated solvers to solve matrix-vector equations.

- the `source` directory, where code for the explicit instantiation for the C++ classes reside.

- the `interface` directory contains an interface to *LAPACK* and one to a so-called Super-LU solver.

[25] Used to be called `lib`

- the `test` directory with some little programs to test some functions and operations of the classes in `include`.

Remember that all classes are templated and that it won't be explicitly written down template <typename T> class cplx; here, but just class cplx; in the next sections.

The class names such as Vector, Matrix ... were formerly written in lowercase letters but were changed not to conflict with the STL[26] which is part of the Standard C++ library.

Classes for complex numbers cplx, vectors Vector, matrices Matrix, Matrices with band structure BdMatrix, Matrices in Fortran compatible storage format F_Matrix, with band strucutre F_BdMatrix, compressed column matrices CSCMatrix, compressed row matrices CRMatrix and tensors Tensor are provided.[27] For the Vector and Matrix classes, the TBCI is implemented to speed up operations.

Also there is a collection of helper functions, which accomodate the parallelization of applications, time measurements for performance testing, interface to the standard C++complex class, interface to LAPACK, a double linked list implementation and a collection of constants and mathematical functions, such as erfc (Gauss error distribution). The Levenberg-Marquard algorithm and some other fitting procedures are also implemented.

2.2 Overview of solvers in lina/include/solver

There are two direct solvers, gauss_jordan and lu_solver. Both methods are described in [PTVF92]. The gauss_jordan is pivoted therefore very stable; if a Matrix can be inverted, the gauss_jordan will do. Also, it's very slow. The lu_solver works by making a LU decomposition of the Matrix. It does not do any pivoting, so it doesn't like diagonal zeroes at all. According to [PTVF92], the matrix has to be diagonal dominant. The solver will detetct, if there are any problems. There is also a variant of the LU solver that operates on BdMatrix. It's optimized, cause it knows the decomposed BdMatrix will only be filled to the outermost diagonal. It's quite efficient.[28]

Also a number of iterative solvers is included in the dircetory. Those were contributed by Jens Lenge, Andreas Ahland and Jörn Kastner. Thanks!

[26] http://www.sgi.com/Technology/STL/

[27] While the framework and the base classes were done by ourselves, the classes F_Matrix, F_BdMatrix, CSCMatrix, CRMatrix were contributed by Jens Lenge, Andreas Ahland and Jörn Kastner

[28] There are more elaborated ways to preview what elements will differ from zero. A solver based on this knowledge is called Super-LU and is very useful, if 3-dimensional problems have to be solved.

These solvers work by having a start vector and iteratively finding better values to solve the matrix vector equation. Their advantage is that no inverse at all has to be build, so if the Matrix had some sort of sparse occupation, we don't need any more memory than before. If a good start vector is known, the iterative solvers are also fast. If not, the LU for a BdMatrix is faster! The methods include: CG, CGS, BiCG, BiCGstab, QMR, GMRes, ... [B+94] The method to be applied depends on the exact condition of the matrix. Hermitean matrices can be handled in another way than general ones. The BiCGstab and CGS solvers are applicable to all type of matrices, while the others have some limitations.

To improve the convergence of the iterative solvers, Preconditioners can be used[B+94]. Currently, there's only one: A diagonal Preconditioner. Also there's a dummy: NoPreconditioner.

2.3 The mpt tree

The `mpt` directory contains programs for simulating the electromagnetic fields in the torch as well as one class for EM simulations with coaxial symmetry: hphi_wave. The earlier can be found in the directories `plasma_coax, open_coax, short_coax` whereas the latter is spreaded into the directories `include, source` and `bin-`*`arch`*. The `Makefile` builds the library `libFD.a` resp. `libFD.so.0.6.0`.

The hphi_wave class implements a finite difference (FD) scheme used to solve PDEs. At every point of the computation domain (grid), the equation holding there can be given by an index. Equations for free space, metal bounds, absorbing boundary conditions (abc = analytic boundary conditions) etc. are implemented in it.

2.4 The bench directory

The `libbench` program is built by calling `make`. It tests the efficiency of vector and matrix operations and compares them to optimized versions. This is useful when implementing optimizations, such as the TBCI (1.4) in classes and see how much benefit can be drawn from that.

Type `libbench -h` to get a list of options.

Currently, the TBCI design eliminated the temporary overhead. We can see that d = a + b + c; is just as fast as d = a; a += b; a += c;. However, the addition could be performed in one loop instead of two. This is what is meant by hand optimized. This actually results in better performance (almost double performance for large objects on some machines, where we have a bad cache hit rate and the time to access memory prevails the computation

time).[29] If the library will need this extra performance somewhere in the future, advanced optimization techniques such as the DOC classes (see 1.6) will have to be used for this kind of operations, too.

2.5 The grid directory

The classes in this directory provide everything that is needed to solve partial differential equations using a finite difference method on an arbitrary grid. There are classes to represent the grid on which the calculations are carried out. As the metrics and Christoffels are properties of the grid, the routines to provide these are also contained in this grid.

On the grid, you can define fields and differential operators. Those also have their classes to be represented by. There's some intelligence to automatically determine what stencil to choose if approaching a border and the FD coefficients are calculated automagically.

The geometry of the setup has to be represented in the computer, when doing calculations. Therefore a geometry class is provided to help you setting up the boundary of objects and automatically assigning nubers to diff. equations and moving grid lines. To modify the grid, there are some stretching functions classes which can be set up to make a grid more dense at the wanted places.

On top of these classes, applications for microwave simulation are built. Currently there are simulation programs for the MPT, a coax line, 1D transmission lines, ...

2.6 The doc directory

Just type `make` to make the documentation in `dvi` format. There are two more options. `make ps` and `make ps2` to make a Postscr*pt or a half size (two pages on a sheet) Postscr*pt file.

[29] This is actually true for quite some modern systems. The AMD K6 CPU, e.g., is memory bound for a lot of operations. A speedup of more then 30% was achieved by just using the PreFetch instruction (3DNow instruction set) in the matrix-vector multiplication!

3 The files and classes in detail

3.1 Files in lina/include

3.1.1 constants.h, basics.h and except.h

`basics.h` is included by every file in the `lina/include` directory. It includes the files `constants.h` and – if the preprocessing directive EXCEPT is defined – `except.h`.

The file `constants.h` contains some physical constants – of course in SI units:
Speed of light (vacuum): c_0, Pi: pi, elementary charge: e_0, (rest) mass of electron: m_e, dielectricity of vacuum: epsilon_0, permeability of vacuum mu_0, waveimpedance of vacuum: Z_0, Boltzmann constant k_B, (natural) logarithm of 2: LN_2, Atomic mass unit: amu, Mass of argon: Ar_mass

The file `except.h` contains the exception class NumErr which is the base class for every exception class generatable in the other files in this directory. There is a default constructor NumErr (), one taking a text as an argument NumErr (char*) and one taking an text as well as an index: NumErr (char*, int).

The exception generation scheme is not very well designed at the moment. (It's not used anyway.). Currently we have an exception class for every object type, so we can see from the exception type (VecErr, MatErr, BdMatErr) where the error occured. The reason is given in the text in the exception constructor. A much cleaner design would have the exception classes to reflect the reason for the exception (such as out of memory, index out of range, operation with non-matching sizes, division by zero, ...). The location where the exception is generated could be given by the text argument.

The file `basiscs.h` itself contains some macros for exception handling and optimization.

The most often used one is the macro BCHK (cond, exclass, txt, ind, rtval) which is used for runtime error checking. If neither ERRCHECK nor BOUNDCHECK is set when `basics.h` is included, the error checking is turned off resulting in superior performance. If it is defined, BCHK performs error checking: If EXCEPT is set, an exception of type exclass is thrown with (txt, ind) as argument list. If not, an error message is printed to stderr with the exclass name, the txt and the ind as information text and the retval is returned.

The MIN and the MAX macros provide just a convenient way for finding the smaller resp. larger value of two. When compiling with GNU-CC, the side-effect safe operators <? resp. >? are used for this. Otherwise, the classical C solution (a<b?a:b) is used.

The macro INFO(txt) is expanded to cout ■ txt or to nothing dependent on whether DEBUGINFO is set or not.

A macro ALIGN(x) and a constant MIN_ALIGN are defined in order to achieve alignment. They can be used the following way: double d ALIGN(8) = 2.1; T t(c) ALIGN(MIN_ALIGN);

There are a few templated helper functions: void swap (T&, T&) and T sqr (const T).

Finally it contains a class named vararg which is used to pass an unknown number of parameters to a function using the C-style ellipsis (...) approach.

3.1.2 smp.h

The file `smp.h` consists of a struct to control the threads, the declaration of a function type thr_job_t suited for use with the threads and of a handful of functions to use these. Please note that all these are No-Ops if SMP is not defined.

On programs which use multithreading, init_threads (); should be called on startup and free_threads (); on exit. The init_threads () function accepts an optional argument, telling it how many threads should be created. On Linux systems, this is not necessary, because the number of CPUs can be autodetected. If this fails, it defaults to 2.

Now, if a piece of code should be parallelized, functions of type thr_job_t should be created to do part of the work. The master function, when called, now should check, if the threads are available by calling threads_avail (). This is needed, because the thread might be busy because parallelization has been done on a higher level. If the threads are available, the job should be divided into sonme slices and parameters should be set up accordingly. Then the subthreads are started by thread_start (). It's a good idea to have the master process the last slice itself. After doing this, it should wait for the subthreads to have finished by calling thread_wait (); on every started thread. The results don't have to passed, cause we have a shared memory architecture.

That's it. For more details, the reader is refered to the comments in the file `lina/source/smp.cc` and to the code for matrix-vector multiplication in `lina/include/matrix.h`, which provides an example on how to write code using multiple threads.

3.1.3 cplx.h

When the projects started, the complex class provided by the compiler did not work satisfyingly, so we were forced to design our own class. The results of this work can be found in `cplx.h`, the complex class carries the name cplx.

In the meantime some people worked on the compiler provided class, so that it is usable now. The file `builtin_complex.h` contains some definitions and wrappers to adapt the syntax of the complex class to the one used in cplx. builtin_complex.h was used to compare the performance to cplx.h and is not currently used. The compiler builtin complex class is slightly faster (10%) than our own implementation, but it's specific to the GNU compiler.

A third file is providing access to the standard C++ complex class: `std_cplx.h`. This class is slightly slower (5%) than ours.

The class contains all necessary operations, constructor and destructors, so that you can operate with complex numbers just like with doubles. The default constructor is cplx<T>::cplx () and the one normally used looks like cplx<T>::cplx (const T& re, const T& im = (T)0); To conjugate a clpx use the operator ~ () or the function conj. There are functions fabs, polar, expi, exp, sqrt provided, member functions real, imag for access and the operator >> and << are overloaded for in- and output.

The operators <, <=, >, >+ are also defined. They return the relationship of the absolute values of the complex numbers. (And not the one of the complex numbers itself. These are of course undefined.)

3.1.4 bvector.h and vector.h

There is a base class called BVector defined in `bvector.h`. It can be used as a container to store numbers or pointers in it. However no arithmetic operations are defined. This is necessary to make it possible to store objects that don't define arithmetic operations, such as pointers.

An object of type BVector is created by one of the following constructors: BVector<T> (); BVector<T> (const unsigned dim); BVector<T> (const T& val, const unsigned dim); BVector<T> (const BVector<T>& bv). The meaning is obvious. dim is the number of elements to be stored (default constructor: 0), the val is an initalization value which the BVector is filled with.

The member access is provided by the T& operator () (const unsigned); all indices start with 0. For valarray-compatibility there is also an operator [] (const unsigned).

The member functions fill, clear, contains, size may also provide some help to the user. The resize member function can be used to change the shape of the BVector. BVectors can be compared with the == and != operators. Two of them are concatenated using the BVector<T> concat (const BVector<T>, const BVector<T>&) function.

I/O is provided by overloading the iostream operators << and >>.

The class Vector is derived from BVector and is defined in `vector.h`. It

addidtionally contains arithmetic operations: +=, −=, *=, /=, +, −, *, /. Not all combinations of operators and arguments are meaningful though. The T operator (const Vector<T>&) const computes the scalar product of two Vectors. There is an alternative operation (T dot (const Vector, const Vector)) which does almost the same but conjugates the first argument (which is interesting only for complex arguments). We can perform elementwise multiplications with emul and cemul which result in a Vector again. For complex type, we can use the conj, real and imag functions to perform these operations on a whole Vector.

Moreover we got the min (), max () and the sum () member functions.

The design with a base class just for storage (BVector) and a derived class with arithmetic functions (Vector) is a little bit hard to optimize with the TBCI design. The BVector class is fully functional except for the arithmetics. We have to define an non-empty destructor for it to clean up used memory. If we put a TBC (eg. TVector) in between BVector and Vector, TVector inherits the BVector destructor which destroys the data, which is not what we want. One possible solution is the use of a virtual destructor. The main disadvantage of this is that the call to the destructor then gets routed over the virtual table which costs some time. Also the design would have to be changed: Libraries with explicitely instantiated classes and functions linked together with code without any template instantiations make problems when some of the code is virtual. So we have to find another way out: We use a flag (bool keep) within BVector to indicate whether the destructor is supposed to delete data or not.

The TBC of Vector is called TVector. Never ever use it except when you are implementing functions that return a vector and you know what you are doing.

The BVector and Vector do check the range of the indices and check if the two vectors are of the same dimension (at the interesting operations) etc. if the variable BOUNDCHECK or ERRCHECK is set during compile time. This is useful for the development of an application whereas for maximum speed, it's better not set. This also applies to Matrix, SpMat, ...

The exception class of BVector and Vector is called VecErr.

3.1.5 matrix.h

Implemented using the TBCI design described above (1.4). The class TMatrix – which the user should *never* use – stores temporaries and return values, whereas the derived class Matrix represents the user accessible objects.

This way we don't waste any time copying large amounts of data. The file gets more complex because of it, but the user interface doesn't change.

(That's C++ ;-)

The functions and operations on a Matrix are similar to those on a Vector. The assignment of or initialization with a value does *not* set the Matrix to this value times the unity matrix as one might expect but fills all elements. The operation to set the diagonal (on quadratic matrices only) is called setunit (const T& = 1).

Constructors: Matrix (unsigned rowcol); Matrix (unsigned rows, unsigned cols); Matrix (T val, unsigned rows, unsigned cols); Matrix (Vector v, enum rowcolvec = colvec); The meaning is obvious ...

Operations: like Vector. Additionally: Matrix Vector multiplication.

Access: operator () (unsigned row, unsigned col). Vectors contained in the Matrix: get_row (unsigned row); set_row (Vector, unsigned row); get_col (unsigned col); set_col (Vector, unsigned col); Note that the Vectors created that way are copies of the data in the Matrix, not references to it.

Info: unsigned columns (); unsigned rows ();

If we use the exception handling, the Matrix class will throw exceptions of class MatErr.

3.1.6 band_matrix.h

The class BdMatrix is implemented in this file. It is a matrix class, which optimizes the memory needs for sparse matrices. We store only the tridiagonal plus a few user-supplied off-diagonals. The interface is designed to be very similar to the one of the matrix class. If we try to access any not stored element a (reference to a) value of zero is returned.

The configuration of the BdMatrix is given by a BVector containing numbers to specify which offdiagonals have to be stored. The diagonal is stored in any case, and if you don't supply a config BVector, both neighbouring off-diagonals are also stored.

The numbers in the config BVector are the distance from the diagonal. The BdMatrix class supposes that the *shape* of the matrix is symmetric. If we want to have a pentadiagonal 200x200 BdMatrix with three inner and two outer diagonals with distance 40 from the main diagonal, we have to create a BVector<unsigned> bv (1); bv(0) = 1; bv(1) = 40; and pass it to the BdMatrix<double> sm (200, bv); constructor.

The operations are very much the same than for Matrix. Generated exceptions are of class BdMatrixErr.

3.1.7 tensor.h

TBW

3.1.8 my_nr.h and mathplus.h

To be written ...

3.2 The directory lina/include/solver/

3.2.1 gauss_jordan.h and lu_solver.h

For numerically solving problems, we almost always need matrix solvers to solve an equation like $Ax = b$, where A is a matrix, x and b are vectors and x is unkown. We got two of them: One implementing the Gauss-Jordan algorithm and one solving the problem by making an LU-decomposition of the matrix. Both are direct solvers as opposed to iterative methods (Krylov solvers). The main disadvantage of them is the large need for memory, because the inverse resp. the LU decomposed matrix needs to be stored. On the other hand, they are relatively easy to implement and have a well known behaviour. (Iterative solvers do not always converge.)

The gauss_jordan solver does use full pivoting and should produce correct reults even for nearly singular matrices. (If your matrix gets nearly singular, better use the Singular Value Decomposition – it will provide more information then.) The lu_solver is currently not using pivoting and you have to be sure that the matrix is diagonal-dominant for a precise solution.

The file `gauss_jordan.h` provides one function: void gaussj (Matrix<T>& M, Matrix<T>& v). The Matrix M gets inverted to M^{-1}, whereas the Matrix v contains one or more vectors to changed into $M^{-1}v$.

The file `lu_solver.h` contains a bunch of functions. The heart of all is int LU_decomp(Matrix<T>& M) which performs the LU decomposition. The LU decomposed matrix is stored in M again.[30]. To obtain a solution for an equation $LUv = b$ we need to substitute the vector into the LU decomposed form, i.e. $Uv = L^{-1}b$ (LU_fwd_subst) and $v = U^{-1}L^{-1}b$ (LU_bkw_subst).

The user however interfaces only the functions TVector lu_solve (Matrix&, const Vector&) and LU_solve. The difference is, that lu_solve does perform an LU decomposition, whereas LU_solve assumes, that the supplied Matrix is already LU decomposed. We also provide version which take a Matrix as second argument, containing multiple vectors to be solved in order to provide the same interface as gaussj does.

We provide two more functions: lu_det, LU_det to calculate the determinant and lu_invert, LU_invert to return the inverse $M^{-1} = U^{-1}L^{-1}$.

[30] Press et al.: Numerical recipes, eq. 2.3.14

3.2.2 bd_lu_solver.h

A version of the lu_solver for the BdMatrix-class. It is optimized to take profit of the structure of the sparse matrix: Only the elements up to the outermost diagonals have to be computed. If the BdMatrix is configured not to hold the elements in between, it will be reconfigured.

The functions contained in `sp_lu_solver.h` are the same as described in `lu_solver.h`. The only difference is, that the matrix M to be LU decomposed is a BdMatrix.

3.2.3 Iterative solvers

CG, CGS, QMR, BiCG, BiCGstab, GMRes, Chebyshev, Exponential Expansion, Iterative Refinement (Richardson), SuperLU (interface)

3.2.4 Preconditioners

None, Diagonal (Jacobi), Incomplete LU

3.3 The files in lina/source

The files consist of some lines of code to explicitly instantiate the classes and operators on them. All code produced that way is linked to a library whose name depends on the type T of data. In the end we get libraries such as `libdouble.a` or `libcplxdouble.so.2.0.0`.

The code which needs the classes instantiations can then be compiled with the -fno-implicit-templates compiler flag[31] and be linked against the libraries created before.

3.4 Files in mpt/include

3.5 Files in mpt/source

3.6 Files in mpt/plasma_coax

3.7 Files in grid

4 Acknowledgements

The work on this library has been carried at the Department for Electrical Engineering, chair High-Frequency Engineering of the University of Dort-

[31] See gcc-manual, ch. 7.5 resp. egcs-manual, ch. 5.5

mund, FR Germany. It has been started by us (Kurt Garloff, Attila Bilgic), but many people contributed to this, especially Andreas Ahland, Joern Kastner and Jens Lenge. Fortunately we also had the support of Prof. Voges; he finally also agreed on releasing the library to the public.

The library is used in a couple of other numerical projects. The most important known to me is PLASIMO. Actually, I work on this here at the ETP group in the Applied Physics Department of Eindhoven Univerity of Technology (NL) now. A lot of useful input came also from my colleagues here. Thanks Jan, Harm, Bart, Colin, Ger! Thanks to the group leader (Joost v.d. Mullen) to support my work as well.

References

[B+94] R. Barret and others: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, Philadelphia, 1994. Available via `ftp://netlib2.cs.utk.edu/linalg/templates2.tar`.

[Fur98] Geoffrey Furnish: *Container-free numerical algorithms in C++*. Computers in Physics **12**(3):258 – 265, May 1998.

[Jos96] Nicolai Josuttis: *Die C++-Standardbibliothek*. Addison-Wesley, Erste Auflage, 1996.

[PTVF92] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery: *Numerical Recipes in C*. Cambridge University Press, second edition, 1992.

[Str97] Bjarne Stroustrup: *The C++ Programming language*. Addison-Wesley, Reading, Massachusetts / AT&T, 3rd edition, 1997.

[SX96] K. Spanderen und Y. Xylander: *Gut kalkuliert, Effiziente Numerik mit C++*. iX Seiten 166–173, November 1996.