The MATH Library

(Version alpha 3-22-2002)

César A. R. Crusius
1. Introduction. The MATH library is my attempt to finally bring to the world the wonderful sensation of the power of object-oriented programming to numerical mathematics. The library is completely templated, so that adding a new type of matrix (such as a tridiagonal matrix) should not only be straightforward but old functions could still be used with it. The library can in principle be used with any system with STL installed and a C++ compiler that supports templates.

When designing the library I had the following philosophy in mind: when you deal with matrices, that’s what you do. You deal with matrices. A matrix is a matrix, and only after that it is a sparse matrix, a symmetric matrix and so on. So the first decision I made was to provide only one matrix type, the matrix class. This is, in my opinion, the correct, intuitive and natural thing to do, and many libraries fail in this point because if you want to add a new matrix type you will actually have to define a whole new matrix class (when in fact you want to redefine the structure or whatever).

Next, I was concerned about an interesting issue: in most matrix libraries, there is no distinction between storage and structure. I’ll explain better: you can have sparse matrices, or you can have symmetric matrices. But can you have sparse symmetric matrices? The concepts are really independent, but most systems do not address this issue correctly: we have to separate these concepts, and that’s what I did. A matrix has as attributes a structure and a storage method. I’ll now explain both in more detail:

- The storage method of a matrix determines how the entries are stored and how they are assigned values. For example, in a sparse matrix assigning an element to zero will actually remove it from memory. Note that this has nothing to do with the matrix structure!

- The structure of a matrix determines the relationship between the entries, and decides on how the storage will be used. I’ll give examples: in a diagonal matrix, for example, the data could be stored in an one-column vector. In a symmetric matrix, the \((i, j)\) element is equal to the \((j, i)\) element, thus the structure can decide that assignment to \((i, j)\) where \(j > i\) will actually not take place. Note that the structure does not need to know how the entries are stored!

In fact, a symmetric structure actually has to deny assignment to half of the matrix. Consider, for example, the algorithm

\[
A(i, j) = 2 \times A(i, j).
\]

The first assignment works fine, but when you try \(A(j, i) = 2 \times A(j, i)\) you will actually be multiplying the new \(A(i, j)\) by two, and you’ll have a wrong result. Due to this fact, we must make the following assumption: all algorithms compute all the elements of the matrix, unless you really know what you are doing. In the symmetric matrix case, for example, you can specialize an algorithm but you need to be sure which elements the symmetric structure really assign a value.

Now some words about performance. Suppose you are a performance freak, who likes to make nice graphics comparing how long does it take to make an SVD decomposition using various libraries. Probably the MATH library will loose. My main concern is to define a philosophically correct library in the programming sense. There is room, however, for performance improvements: you can always specialize the matrix type you are concerned with. The most obvious case is the dense and unstructured (or symmetric) matrix, which is used everywhere.

Brief remark: actually there are two correct philosophies: the template approach I just described and an “storage and structure hierarchy” approach, which would enable lists of different types of matrices but no specialization. Hence, I was guided by performance in some sense.
2. **Basic definitions.** The natural place to start is with the basic definitions, which are the the `index` and the `error` types: together, they provide us with the necessary tools to begin the longer `matrix` definition. These basic definitions, along with the matrix definition, are declared in the `math.h` file. As customary, the definitions are stated in the header file, and some bigger function bodies are defined in `math.cc`. As will happen with all MATH classes and functions, these definitions are declared inside the `math` namespace. The next three sections follow a pattern on this document: first we define the beginning of the source and header files, and then proceed with the code.

```c
#include "math.h"
(Big definitions 112)
```

3. ```c
   #ifndef __MATH__
   #define __MATH__ 1.0
   (Include files math 6)
   namespace math {
   (Preprocessor definitions)
   namespace math {
   (Basic definitions 4)
   (Element definition 65)
   (Structure definition 11)
   (Storage definition 10)
   (Matrix definition 12)
   (Submatrix definition 81)
   (Basic algebraic operations 105)
   (Specializations 106)
   (export-waiting big definitions 33)
   }
   #endif
```

4. The first thing we do is to define the C type that will be used for indexing elements of matrices. We define it as an `unsigned int`, which means, for one thing, that you can not define behaviors based on an `index` being negative. Since the entire library is based on the (1, 1) origin default, you can always use a zero value as an indicator. Note that an `index` definition already exists – it is defined in the standard library’s `string` class. Therefore, a safe use of the type will be `math::index`, even when you declare that you are using namespace `math`.

```c
typedef unsigned int index;
(Basic definitions 4) ≡
```

See also sections 5, 40, and 82.

This code is used in section 3.
5. In order to be able to differentiate MATH errors from others, we will define some classes under the namespace `math::error` which we will use to signal anomalies. Our objective here is to provide a consistent and flexible means of passing errors from the library. The first generic error class will contain a string that can be used to describe the cause of the errors. Common errors can be derived from this class.

```cpp
namespace error {
    class generic {
        string theMessage;
        public:
            virtual ~generic() {} /* A base class it is. */
    };
}
```

This code is used in section 5.

6. Include files `math`

```cpp
#include <string>
```

See also sections 31, 113, and 114.

This code is used in section 3.

7. The `generic` error class is intended to be used in `try-catch` mechanisms, so when something goes wrong you simply `throw` an error. In this way, it doesn’t make sense to declare an `error` variable and update it during the program. Hence, the only provided way to modify an `error` is at the time of construction, and we will then need the appropriate constructors: the two defined below are used when the error we want to signal is not predefined. In that case, we create an unknown error with or without an explaining message.

```cpp
const string &message(void) const { return theMessage; }
```

See also section 8.

This code is used in section 5.

8. Of course, you may want to check out the message that was passed. We provide one method to retrieve the friendly error message. Depending on the error type we could even not consider it an error (see, for example, the `det` function).

```cpp
const string &message(void) const { return theMessage; }
```
Next we deal with predefined errors. What we do is to derive some classes that implement common errors.

\[
\text{(Predefined error types) } \equiv
\]

```c++
class singular : public generic {
  public: singular()
    : generic("Matrix is singular to working precision.") {} };
class fileerr : public generic {
  public: fileerr()
    : generic("Generic file error.") {} };
class infeasible : public generic {
  public: infeasible()
    : generic("Problem is infeasible.") {} };
class nonsquare : public generic {
  public: nonsquare()
    : generic("Matrix should be square") {} };
class nonpositivedef : public generic {
  public: nonpositivedef()
    : generic("Matrix should be positive definite.") {} };
class dimension : public generic {
  public: dimension()
    : generic("Wrong matrix dimensions.") {} };
class notimplemented : public generic {
  public: notimplemented()
    : generic("Feature not yet implemented.") {} };
class maxiterations : public generic {
  public: maxiterations()
    : generic("Maximum number of iterations reached.") {} };
class unboundedbelow : public generic {
  public: unboundedbelow()
    : generic("Problem is unbounded below.") {} };
class domain : public generic {
  public: domain()
    : generic("Domain violation.") {} };
class rankdeficient : public generic {
  public: rankdeficient()
    : generic("Matrix is rank deficient.") {} };
```

This code is used in section 5.
10. Matrix basics. Let us now begin the fun part. A matrix is a template on the type of its elements, its structure and its storage. Being that way, a matrix can be symmetric and sparse at the same time. The matrix is, then, a container for a storage of elements that are to be used according to some structure. The function of a matrix consists in coordinating the use of its storage by asking the structure what to do. In what follows we will define this protocol as necessity arrives by using the standard dense storage and unstructured structure.

⟨Storage definition 10⟩ ≡

```cpp
template<class T>
class dense {
    typedef T element_type;
    public:
    ⟨Dense storage internal variables 28⟩
    public:
    ⟨Dense storage methods 29⟩
}
```

This code is used in section 3.

11. ⟨Structure definition 11⟩ ≡

```cpp
template<class T>
class unstructured {
    typedef T element_type;
    public:
    ⟨Unstructured structure methods 27⟩
}
```

This code is used in section 3.

12. Now to the user’s matrix type. The first thing we need to realize is that we will inevitably need to return matrices from some functions (it suffices to think of a function to return the identity matrix). In order to minimize the overhead in passing matrices around, we separate the matrix representation from the matrix handle. The matrix type is, then, a handle to a representation that can be shared among many matrices. An assignment of the type $A = B$ will be equivalent to “matrix $A$ will share matrix $B$ representation.” Of course, we need to be careful to not modify matrices we don’t want to. For example, $A(1,1) = 1$ can not modify $B$. As we proceed we will take care of all these cases.

```cpp
#define matrix_simple_template
    class T ← double ,
    template<class> class structure ← unstructured ,
    template<class> class storage ← dense
```

⟨Matrix definition 12⟩ ≡

```cpp
⟨Matrix representation definition 13⟩
    template<matrix_simple_template>
class matrix {
    public:
    ⟨Matrix internal types 14⟩
    private:
    ⟨Matrix internal variables 15⟩
    public:
    ⟨Matrix methods 19⟩
}
```

This code is used in section 3.
13. As aligned before, a matrix is simply a handle to a matrix representation. It is the representation who holds the storage and has the structure information. Hence, a representation is a template of all of them.

\[
\langle \text{Matrix representation definition 13} \rangle \equiv \\
\text{template} \langle \text{matrix_simple_template} \rangle \\
\text{class representation} \{} \\
\text{typedef} \ T \text{ element_type; } \\
\text{\langle Matrix representation internal variables 17 \rangle} \\
\text{public:} \\
\text{\langle Matrix representation methods 18 \rangle} \\
\text{\};}
\]

This code is used in section 12.

14. A matrix knows its representation through a pointer to it. In this way a single representation can be shared among various matrices. This also means that it is the matrix itself who must coordinate creation of new representations and destruction of them, as we shall see later.

\[
\langle \text{Matrix internal types 14} \rangle \equiv \\
\text{typedef representation} \langle T, \text{structure, storage} \rangle \text{ rep_type; }
\]

See also sections 16 and 88.

This code is used in section 12.

15. \[
\langle \text{Matrix internal variables 15} \rangle \equiv \\
\text{rep_type } *\text{theRepresentation;}
\]

This code is used in section 12.

16. Before going on, we note that we must have a means to obtain the type of elements, structure and storage of a matrix. Since a matrix is a template, we provide a means of accessing the type of the matrix as internal types in the same way as the Standard Template Library does.

\[
\langle \text{Matrix internal types 14} \rangle +\equiv \\
\text{typedef} \ T \text{ element_type; } \\
\text{typedef storage} \langle T \rangle \text{ storage_type; } \\
\text{typedef structure} \langle T \rangle \text{ structure_type; } \\
\text{typedef matrix} \langle T, \text{structure, storage} \rangle \text{ matrix_type; }
\]

17. Now back to business. Since the representation is the owner of the storage, it is the representation who holds the size of the matrix, not the matrix type.

\[
\langle \text{Matrix representation internal variables 17} \rangle \equiv \\
\text{index num_rows, num_cols;}
\]

See also sections 20, 21, and 50.

This code is used in section 13.
18. As will happen often, the representation provides methods to access the information about the matrix, and the matrix type provides the interface to them. Often the representation methods don’t perform any error checking, since the user will in the end use only the matrix type. The definition of two types of methods (one const and one not) is necessary for specializations (for example, the reshape function will modify the number of rows directly).

\[
\text{Matrix representation methods} \equiv
\]

\[
\begin{align*}
\text{index} & \ \& \ \text{rows} (\text{void}) \ \{ \ \text{return} \ \text{num\_rows}; \ \}
\text{index} & \ \& \ \text{cols} (\text{void}) \ \{ \ \text{return} \ \text{num\_cols}; \ \}
\text{index} \ \text{rows} (\text{void}) \ \text{const} \ \{ \ \text{return} \ \text{num\_rows}; \ \}
\text{index} \ \text{cols} (\text{void}) \ \text{const} \ \{ \ \text{return} \ \text{num\_cols}; \ \}
\end{align*}
\]

See also sections 22, 25, 26, 47, 51, 54, 59, 63, 72, and 77.

This code is used in section 13.

19. \text{(Matrix methods) } \equiv

\[
\begin{align*}
\text{index} \ \text{rows} (\text{void}) \ \text{const} \ \{ \ \text{return} \ (\text{theRepresentation} ? \text{theRepresentation\_rows}() : 0); \ \}
\text{index} \ \text{cols} (\text{void}) \ \text{const} \ \{ \ \text{return} \ (\text{theRepresentation} ? \text{theRepresentation\_cols}() : 0); \ \}
\end{align*}
\]

See also sections 23, 24, 34, 36, 37, 38, 42, 44, 45, 46, 53, 57, 58, 62, 71, 73, 74, 76, 89, 92, 94, 96, 99, 102, 108, and 115.

This code is used in section 12.

20. Also, as aligned before, it is the representation who has the storage and structure information. It will be the job of the representation to use them appropriately.

\[
\text{(Matrix representation internal variables) } \equiv
\]

\[
\begin{align*}
\text{storage} (\text{T}) * \text{theStorage}; \\
\text{structure} (\text{T}) * \text{theStructure};
\end{align*}
\]
21. Creating and copying a matrix. We can now begin to handle the ways in which a representation is handled by the matrix, and we will start by the ways a matrix is created. First, we need to know how many matrices are sharing a single representation. Obviously, this information can only be kept by the representation itself, and the matrix must be able to retrieve this information.

\[
\begin{align*}
\text{Matrix representation internal variables} & \vdash \\
\text{int } & \text{num\_instances;}
\end{align*}
\]

22. When a matrix is emptied or some operation modifies it, it is often the case that a matrix will need to create a new representation. On the other hand, copying matrices simply means incrementing the number of shared instances of some representation. Since it is the matrix who deals with this protocol, we return a reference to the number of instances so that a matrix can modify it itself.

\[
\begin{align*}
\text{Matrix representation methods} & \vdash \\
\text{inline } & \text{int } \&\text{instances(}) \text{void} \text{)} \{ \text{return num\_instances; } \}
\end{align*}
\]

23. Let us then enable a matrix to be created. The default constructor should create an empty matrix, and the dimension constructor creates a fresh representation.

\[
\begin{align*}
\text{Matrix methods} & \vdash \\
\text{matrix(} \text{void}): & \text{theRepresentation(}0\text{)} \text{ } \{ \}
\end{align*}
\]

24. A representation is never empty. It is always created with dimensions, in the way you just saw. In order to really forbid the empty representation creation we define a default constructor which throws an error.

\[
\begin{align*}
\text{Matrix representation methods} & \vdash \\
\text{representation(} \text{void)} & \{ \\
\text{throw error::generic("Cannot instantiate empty matrix representation!");}
\end{align*}
\]

25. It is the job of the representation constructor to create the structure and the storage. By defining the representation constructor we begin, then, to definite the communication protocol between storages and structures. We will explain the protocol by defining the dense and unstructured classes.

\[
\begin{align*}
\text{Matrix representation methods} & \vdash \\
\text{representation(index rows, index cols):} & \text{num\_rows(rows), num\_cols(cols), num\_instances(1)} \{
\text{theStructure } & \text{← new structure(element\_type)(&rows, &cols); }
\text{theStorage } & \text{← new storage(element\_type))(rows, cols); }
\end{align*}
\]
27. The first part of the protocol consists in creating a matrix with enough storage for its elements. A symmetric matrix, for example, does not need to store \( \text{rows} \times \text{cols} \) elements. Hence we first create the structure, and rely on the constructor to modify its arguments so that the a posteriori storage creation works accordingly. For the unstructured type, however, we don’t need to modify the arguments.

\[
\text{Unstructured structure methods } 27 \equiv \\
\text{unstructured(index *, index *)} \{ \}
\]

See also sections 49, 60, and 78.

This code is used in section 11.

28. Having now the correct dimensions of the matrix we can create the storage. Let’s do it for the dense case. A dense matrix is stored as a vector in \( \text{elements} \) by columns, that is, the first elements are from the first column and so on. The index to the first element of a column is indexed in \( \text{data} \). Later on this can appear to be unintuitive, but this scheme saves memory for vectors, when there is only one element in \( \text{data} \), and also it will facilitate the build of a LAPACK interface. We will also allow the storage to have storage for more rows and columns than what is needed. This will be very useful when resizing.

\[
\text{Dense storage internal variables } 28 \equiv \\
\text{element_type *\text{elements};}
\text{element_type **\text{data};}
\text{index num_rows, num_cols;}
\text{index max_rows, max_cols;}
\]

This code is used in section 10.

29. We define, as usual, a default constructor to zero everything.

\[
\text{Dense storage methods } 29 \equiv \\
\text{dense( );:\text{elements}(0), \text{data}(0), num_rows(0), num_cols(0), max_rows(0), max_cols(0) \{} \}
\]

See also sections 30, 32, 48, 52, 55, 56, 61, 64, 75, and 79.

This code is used in section 10.

30. Now to the useful dense constructor. We set the initial array values to zero, and we use \text{memset} \ for doing this. One could argue that we should do a loop and assign \text{element_type}(0) \ to each element, but let’s get real... Since the task of initializing the data is also useful outside the constructor, we define an \text{init} \ method to be used for initialization and call it from the constructor.

\[
\text{Dense storage methods } 29 \equiv \\
\text{dense(const index &rows, const index &cols) : \text{elements}(0), \text{data}(0) \{} \\
\text{init(rows, cols);}
\text{memset(elements, 0, num_rows * num_cols * sizeof(element_type));}
\}
\]

31. (Include files \text{math 6}) \equiv \\
\text{#include <string.h>} \quad /* For \text{memset} \ (and \text{memcpy}). */

32. The \text{init} \ function allocates space and initializes the data pointers, but leaves the data contents unchanged. The method will be constructed so that it can be used to resize the data if the number of elements remains the same (if they are not, you’ll have segfault somewhere later). The trick in this method is that the vectors are initialized in a way such that the \((1, 1)\) origin standard works, that is, \(\text{data}[1][1]\) is the first element of the matrix. The only catch is that, due to the way the data is stored, the element \(A(i, j)\) is stored in \(\text{data}[j][i]\).

\[
\text{Dense storage methods } 29 \equiv \\
\text{void init(const index &rows, const index &cols);}
\]
33. (export-waiting big definitions 33) ≡

```cpp
template<class T> void dense(T)::init(const index &rows, const index &cols)
{
    num_rows ← max_rows ← rows;
    num_cols ← max_cols ← cols;
    if (!elements) elements ← new element_type[rows * cols];
    if (data) delete[] ++data;
    data ← new element_type*[cols];
    for (index i ← 0; i < cols; ++i) data[i] ← elements + i * rows - 1;
    data--;}
```

See also sections 35, 39, 43, and 80.
This code is used in section 3.

34. Voilà! We are now able to create an empty matrix and a matrix filled with zeros. Sometimes, however, it is useful to create a matrix filled with some specified value. We provide a method to fill a matrix with a specified value and a constructor to do the same thing.

```cpp
void fillwith(const element type &value);
```

35. (export-waiting big definitions 33) ≡

```cpp
template<matrix simple template> void matrix(T,structure,storage)::fillwith(const element type &value)
{
    for (index i ← 1; i ≤ rows(); i++)
        for (index j ← 1; j ≤ cols(); j++) theRepresentation.set(i,j,value);
}
```

36. (Matrix methods 19) ≡

```cpp
matrix(const index rows, const index cols, const element type &value)
{
    #ifdef __MATH_INSANE_DEBUG__
        cout "[math]:matrix(" "< rows « ', '< cols « ', '< value « ")\oldrep=" « theRepresentation;
    #endif
    theRepresentation ← new rep_type(rows, cols);
    fillwith(value);
    #ifdef __MATH_INSANE_DEBUG__
        cout "\newrep=" « theRepresentation « \n;
    #endif
}
```

37. Next we define the copy constructor. As noted before, the only thing we need to do is to share the representation and update the number of shared instances.

```cpp
matrix(const matrix type &source)
{
    if (source.theRepresentation) source.theRepresentation−instances()++;
    theRepresentation ← source.theRepresentation;
}
```
38. Now we have all the basic constructors we need. The next step is to define the copy operations, which are very similar. In order to be able to do that we define a method that reinitializes a matrix to a given size. This method is like a constructor, but it takes into account the fact that the matrix can already have a representation. If it has one and its not shared and it has the same dimensions we don’t need to do anything. In the other case we need to create a new representation anyway, taking care of the old one (the deletion of a representation is the topic of the next section, but the details are not necessary here).

Matrix methods
\[\text{void init(const index num\_rows, const index num\_cols);}\]

39. \[\text{template(matrix\_simple\_template) void matrix\langle T, structure, storage\rangle::init(const index num\_rows, const index num\_cols);}\]

\[
\begin{aligned}
&\text{if (theRepresentation \& cols( ) \equiv num\_cols \& rows( ) \equiv num\_rows \& theRepresentation\_instances( ) \equiv 1) return;} \\
&\text{if (theRepresentation \& --theRepresentation\_instances( ) \equiv 0) delete theRepresentation;} \\
&\text{theRepresentation \leftarrow new rep\_type(num\_rows, num\_cols);} \\
\end{aligned}
\]

40. We are now able to define the assignment operator. In our context the task is a simple matter of using the same representation. Sometimes, however, this is not what we want: we want the contents copied and a new representation created. This normally occurs in numerical code where all matrices are modified almost instantly anyway. We provide a way to control the behavior of the assignment operator through the global fast_assignment. If fast_assignment is true, then the assignment operator uses the faster but memorywise expensive copyfrom method, to be defined later.

Basic definitions
\[\text{extern bool fast\_assignment;}\]

41. bool math::fast\_assignment \leftarrow false;

42. Now to the method itself. In case fast_assignment is false, we share the representation, the only catch being that we need to avoid confusions when making ridiculous things like \( A = A \).

Matrix methods
\[\text{matrix\_type \&operator\leftarrow(const matrix\_type \&source);}\]

43. \[\text{template(matrix\_simple\_template) matrix\langle T, structure, storage\rangle::matrix\_type}\]
\[\&\text{matrix\langle T, structure, storage\rangle::operator\leftarrow(const matrix\_type \&source);}\]

\[
\begin{aligned}
&\text{if (fast\_assignment) return copyfrom(source);} \\
&\text{if (source.theRepresentation \& source.theRepresentation\_instances( )++) source.theRepresentation\_instances( )++;} \\
&\text{if (theRepresentation \& --theRepresentation\_instances( ) \equiv 0) delete theRepresentation;} \\
&\text{theRepresentation \leftarrow source.theRepresentation;} \\
&\text{return \&this;} \\
\end{aligned}
\]
44. We may also want to perform assignments with different types of source matrices. In that case the only way is to copy all elements, one by one. This method requires the operator \((\quad)\), which retrieves an element from a matrix, and the methods \textit{set}, which assigns a value to a particular element. The details are irrelevant for now.

\[
\#define\ \text{matrix\_template}(M)
\]

\[
\begin{align*}
\text{class } & T\ #\ M, \\
\text{template}\langle\text{class}\rangle & \text{class } \text{STR}\ #\ M, \\
\text{template}\langle\text{class}\rangle & \text{class } \text{STO}\ #\ M
\end{align*}
\]

\langle\text{Matrix\ methods } 19\rangle +\equiv

\[
\text{template}\langle\text{matrix\_template}(A)\rangle
\]

\[
\text{matrix\_type} & \&\text{operator} \leftarrow (\text{const matrix}(\text{TA}, \text{STRA}, \text{STOA}) & \text{source})
\]

\[
\begin{align*}
\text{init}(\text{source.rows()}, \text{source.cols}()); \\
\text{index } i, j; \\
\text{for } (i \leftarrow 1; \ i \leq \text{rows}(); \ ++i) \\
\text{for } (j \leftarrow 1; \ j \leq \text{cols}(); \ ++j) \ \text{set}(i, j, \text{source}(i, j)); \\
\text{return } *\text{this};
\end{align*}
\]

45. \langle\text{Matrix\ methods } 19\rangle +\equiv

\[
\text{template}\langle\text{matrix\_template}(A)\rangle
\]

\[
\text{matrix}(\text{const matrix}(\text{TA}, \text{STRA}, \text{STOA}) & \text{source}); \text{theRepresentation}(0)
\]

\[
\begin{align*}
\text{if } (\text{theRepresentation} \equiv \text{source.theRepresentation}) & \ \text{return } *\text{this}; \\
\text{if } (\text{theRepresentation} \wedge \text{theRepresentation.\_instances()} \equiv \text{source.rows}() \wedge \text{cols()} \equiv \text{source.cols}()) & \{
\begin{align*}
\text{theRepresentation} & \leftarrow *\text{(source.theRepresentation)}; \\
\text{return } *\text{this};
\end{align*}
\}
\text{if } (\text{theRepresentation} \wedge \neg \text{theRepresentation.\_instances}() \equiv \text{0}) & \ \text{delete } \text{theRepresentation}; \\
\text{theRepresentation} & \leftarrow \text{new rep\_type}(*\text{(source.theRepresentation)}); \\
\text{return } *\text{this};
\end{align*}
\]

46. On occasion we may want to copy a matrix without sharing the representation (for example, when we know the matrix will be modified right away so that a new representation will be created anyway if we share the instance). So here’s what we do: if we don’t have a representation yet we simply create a new one. If we do have one and it has the same dimensions (we know it has the same type), we assign a copy of the old one to it (this can be very fast). If none of this happens we behave just like the assignment operator, except we always will create a new representation.

\langle\text{Matrix\ methods } 19\rangle +\equiv

\[
\text{matrix\_type} & \&\text{copyfrom}(\text{const matrix\_type} & \text{source})
\]

\[
\begin{align*}
\text{if } (\text{theRepresentation} \equiv \text{source.theRepresentation}) & \ \text{return } *\text{this}; \\
\text{if } (\text{theRepresentation} \wedge \text{theRepresentation.\_\_instances}() \equiv \text{1} \wedge \text{rows}() \equiv \text{source.rows}() \wedge \text{cols()} \equiv \text{source.cols}()) & \{
\begin{align*}
\text{theRepresentation} & \leftarrow *\text{(source.theRepresentation)}; \\
\text{return } *\text{this};
\end{align*}
\}
\text{if } (\text{theRepresentation} \wedge \neg \text{theRepresentation.\_\_instances}() \equiv \text{0}) & \ \text{delete } \text{theRepresentation}; \\
\text{theRepresentation} & \leftarrow \text{new rep\_type}(*\text{(source.theRepresentation)}); \\
\text{return } *\text{this};
\end{align*}
\]
47. In order to define the \emph{copyfrom} method we created a representation based on an existing one. We will need to do the same thing when setting elements. For this task we will define a copy representation constructor, which in turn will require copy constructors for both the dense and unstructured classes.

\begin{verbatim}
representation(const representation &source)
{
    theStructure ← new structure(T)(*(source.theStructure));
    theStorage ← new storage(T)(*(source.theStorage));
    num_instances ← 1;
    num_rows ← source.rows();
    num_cols ← source.cols();
}
\end{verbatim}

48. \begin{verbatim}
dense(const dense &source)
    : elements(0), data(0) {
    init(source.num_rows, source.num_cols);
    memcpy(elements, source.elements, num_rows * num_cols * sizeof(element_type));
}
\end{verbatim}

49. \begin{verbatim}
unstructured(const unstructured &)
{
}
\end{verbatim}

50. We also used assignment operators in \emph{copyfrom}. In order to simplify things, we only allow assignment operators to be called when the sources have the same dimensions. This may change in the future, but for now it’s good enough (since the user will never use these operators directly).

\begin{verbatim}
typedef representation(T, structure, storage) rep_type;
\end{verbatim}

51. \begin{verbatim}
rep_type &operator=(const rep_type &source)
{
    if (source.rows() ≠ rows() ∨ source.cols() ≠ cols())
        throw error::generic("Can only assign representation to same dimension.");
    theStorage ← *(source.theStorage);
    theStructure ← *(source.theStructure);
    return *this;
}
\end{verbatim}

52. Here we see why \emph{copyfrom} can be much faster: the assignment operator for the representation didn’t involve any memory allocation. Now, the \emph{dense} assignment operator will not allocate memory too, and will use an optimized routine for copying elements. We can not use a single \texttt{memcpy} because there is a possibility that the source storage has a different size of allocated memory (because of max\_rows and max\_cols).

\begin{verbatim}
dense(T) &operator=(const dense(T) &source)
{
    if (source.num_rows < num_rows ∨ source.num_cols < num_cols)
        throw error::generic("Incompatible\_dimension\_in\_dense\_assignment\_operator.");
    for (index i ← 1; i ≤ num_cols; ++i)
        memcpy(data[i] + 1, source.data[i] + 1, num_rows * sizeof(element_type));
    return *this;
}
\end{verbatim}
53. **Destroying a matrix.** Destroying a matrix would be very simple if it wasn’t for the fact that another matrix can be sharing the same representation. This fact makes the destruction of a matrix to be downgraded to the “simple” category. When a matrix is deleted, it can happen that there is another matrix sharing the representation. We only delete the representation if we are the sole matrix using it. If we are not, we only update the number of matrices sharing the representation.

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡

\[\text{Matrix methods}\] +≡
57. **Setting and getting elements.** At this point we are able to create and destroy matrices. The next step is to allow the user to set individual elements. Since the behavior of the set operation depends heavily on the matrix structure, the matrix type calls the representation to perform the structure-storage communication protocol. The matrix must, however, create a new representation in case the current one is shared, otherwise we will modify other matrices too. Since this operation is useful in other situations (for example when calling LAPACK routines), we define a method that creates a new representation for the matrix if necessary.

\[ \text{\texttt{Matrix methods 19} \equiv} \]
\[
\begin{align*}
\text{\texttt{void detach(void)}} & \equiv \\
\text{\texttt{\{}} & \\
\text{\texttt{\texttt{if (\texttt{theRepresentation} \& \texttt{theRepresentation-\texttt{instances}( )} > 1) \{}} & \\
\text{\texttt{\quad \texttt{--theRepresentation-\texttt{instances( );}}}} & \\
\text{\texttt{\quad theRepresentation \leftarrow \texttt{new rep_type(*theRepresentation);}}} & \\
\text{\texttt{\}}}} & \\
\text{\texttt{\}}} & \\
\end{align*}
\]

58. We are now in position to set an element.

\[ \text{\texttt{Matrix methods 19} \equiv} \]
\[
\begin{align*}
\text{\texttt{element_type set(index row, index col, element_type value)}} & \equiv \\
\text{\texttt{\{}} & \\
\text{\texttt{\quad detach();}} & \\
\text{\texttt{\quad return theRepresentation-set(row, col, value);}} & \\
\text{\texttt{\}}} & \\
\end{align*}
\]

59. The representation performs the handling. First it calls the structure preprocess method. The structure will then modify the index accordingly so that a call to the storage set method will modify the correct element. If the structure returns false it means the element is not assignable.

\[ \text{\texttt{Matrix representation methods 18} \equiv} \]
\[
\begin{align*}
\text{\texttt{element_type set(index row, index col, element_type value)}} & \equiv \\
\text{\texttt{\{}} & \\
\text{\texttt{\quad if (theStructure-preprocess(&row, &col)) return theStorage-set(row, col, value);}} & \\
\text{\texttt{\quad return theStorage-get(row, col);}} & \\
\text{\texttt{\}}} & \\
\end{align*}
\]

60. For unstructured structures the preprocess method is empty, and the dense set method is trivial (remember the way the data is stored).

\[ \text{\texttt{Unstructured structure methods 27 \equiv} \]
\[
\begin{align*}
\text{\texttt{bool preprocess(index *, index *) \{ return true; \}}} & \equiv \\
\end{align*}
\]

61. (Dense storage methods 29 \equiv \]
\[
\begin{align*}
\text{\texttt{element_type set(const index &row, const index &col, const element_type &value)}} & \equiv \\
\text{\texttt{\{}} & \\
\text{\texttt{\quad return data[col][row] \leftarrow value; \}}} & \\
\text{\texttt{\}}} & \\
\end{align*}
\]
62. Wow! We are now able to assign elements to matrices! Next I guess you will want to retrieve them, and the most intuitive way to do that is to use the parenthesis operator. For reasons that will become clear, we should not allow the parenthesis operator to be used for assignment, that is, we should not allow something like \( A(i, j) = 0 \) (if you insist then yes, we can allow it, but with a huge price in performance). For this reason we do not return a reference, but the value itself. Now to the protocol. Again, a matrix doesn’t know how to retrieve the value, since the way to retrieve something depends on the structure, so it calls the representation get method to do the job. For consistency reasons we also define a get method (to pair the set one).

\[
\begin{align*}
\text{const element type operator}() & \text{ (const index row, const index col ← 1) const} \\
\{ & \text{return theRepresentation-get(row, col); } \\
\text{const element type get} & \text{(const index row, const index col) const} \\
\{ & \text{return theRepresentation-get(row, col); } \\
\end{align*}
\]

63. The representation needs now to perform the communication again. First it calls the structure preprocess to get the right index, and then retrieve the element from the storage. Strictly speaking we would need a postprocessing on the element value for Hermitian matrices, but in this feature will wait a little more.

\[
\begin{align*}
\text{const element type get} & \text{(index row, index col) const} \\
\{ & \text{theStructure-preprocess(&row, &col); } \\
 & \text{return theStorage-get(row, col); } \\
\end{align*}
\]

64. A storage get method must return zero if either the column or the row indexes are zero. In this way it is possible for a structure to force certain entries to be zero. In a diagonal matrix, for example, the structure preprocess method should assign zero to the index whenever \( row \neq col \) (and also return \( false \)).

\[
\begin{align*}
\text{const element type get} & \text{(const index &row, const index &col) const} \\
\{ & \text{if } (\neg row \lor \neg col) \text{ return element type(0); } \\
 & \text{return data[col][row]; } \\
\end{align*}
\]

65. Now to the main issue. How to deal with \( A(i, j) = x ? \) Assigning a zero element to a sparse matrix means deleting it, so we cannot return a reference to the data. There are two possible solutions: either we force the user to use the set method or we return a structure. The first solution is cumbersome and hard to read, hence we choose the second, and define a method that will return an element. An element works like a pointer to a matrix entry. When you assign an element a value, it calls the matrix set method accordingly.

\[
\begin{align*}
\text{template} & \text{(class matrix_type)} \\
\text{class element} & \{} \\
\{ & \text{Matrix element internal variables 66} \\
\text{public:} & \} \\
\{ & \text{Matrix element methods 68} \\
\}
\end{align*}
\]

This code is used in section 3.
66. An element doesn’t need to know about it’s value, since, as said before, it works like a reference to a particular index of a particular matrix. These are, then, the only data an element must have.

⟨Matrix element internal variables 66⟩ ≡

\[
\begin{align*}
\text{matrix\_type} & \,*\text{theMatrix}; \\
\text{index} & \,i, \,j;
\end{align*}
\]

See also section 67.

This code is used in section 65.

67. The numeric type of the element is retrieved from the matrix. This type is useful (essential!) for readability in the sequel.

⟨Matrix element internal variables 66⟩ +≡

\[
\begin{align*}
typedef \text{typename} \text{matrix\_type}::\text{element\_type};
\end{align*}
\]

68. An element is not intended to be used as a user variable, and is also not intended to be constructed without arguments. Hence, we define the default constructor to throw an error. An element should be constructed with all necessary arguments, which are the matrix instance and the index to the matrix entry.

⟨Matrix element methods 68⟩ ≡

\[
\begin{align*}
\text{element} & \,(\text{void}):\text{theMatrix}(0), i(0), j(0) \\
& \{ \text{throw error}\::\text{generic}(\"Default\_constructor\_of\_element\_should\_not\_be\_used!\); \}
\end{align*}
\]

See also sections 69, 70, 95, 100, 103, and 107.

This code is used in section 65.

69. ⟨Matrix element methods 68⟩ +≡

\[
\begin{align*}
\text{element}(\text{matrix\_type} \,*\text{mat}, \,\text{const index} \,\text{row}, \,\text{const index} \,\text{col}):\text{theMatrix}(\text{mat}), i(\text{row}), j(\text{col}) \{ \}
\end{align*}
\]

70. As aligned before, the main purpose of an element is to call the set method of it’s matrix when assigned a value. This will ensure the correct processing of the assignment taking into account the matrix structure.

⟨Matrix element methods 68⟩ +≡

\[
\begin{align*}
\text{inline element\_type} & \,\text{operator←}(\text{const element\_type} \,\&\,\text{value}) \,\text{const} \{ \\
& \text{return} \,\text{theMatrix}-\text{set}(i, j, \text{value}); \}
\end{align*}
\]

71. Now we are finally able to define the matrix method to be used for readable assignments. The method returns an element which you can assign a value in a friendly way. We name the method entry, so that it makes logical sense. In a program, you would write \(A.\text{entry}(1, 1) \leftarrow 0\) or something similar. As you can see, that construction is much better, in visual terms, than the equivalent set method. Depending on the compiler, however, it is slower. But you can’t always get what you want.

⟨Matrix methods 19⟩ +≡

\[
\begin{align*}
\text{element}(\text{matrix\_type}) & \,\text{entry}(\text{const index} \,\text{row}, \,\text{const index} \,\text{col} \leftarrow 1) \\
& \{ \\
& \text{return} \,\text{element}(\text{matrix\_type})(\text{this}, \,\text{row}, \,\text{col}); \\
& \}
\end{align*}
\]

72. By now we are able to set and get individual elements values. It is, however, useful sometimes to be able to have direct access to the storage elements (for example, when interfacing with another library such as LAPACK). In this case, we also detach the matrix so that there is no danger of messing around with shared representations.

⟨Matrix representation methods 18⟩ +≡

\[
\begin{align*}
\text{storage}(\text{T}) & \,*\text{story}(\text{void}) \\
& \{ \text{return} \,\text{theStorage}; \}
\end{align*}
\]
73. \langle \text{Matrix methods } 19 \rangle \equiv

```
storage_type *story(void)
{
    detach();
    return (theRepresentation ? theRepresentation->story() : 0);
}
```

74. And sometimes we may want the representation too.

\langle \text{Matrix methods } 19 \rangle \equiv

```
rep_type *rep(void)
{
    detach();
    return theRepresentation;
}
```

75. Now that we have a means to get the matrix storage instance, we provide a means to have access to
the storage elements themselves. Since this method will be used only when we know which kind of storage
we have, it will differ between every storage type, and it is not required for one.

\langle \text{Dense storage methods } 29 \rangle \equiv

```
element_type *memory(void) { return elements; }
```
76. Resizing. We define a resize operation as one that changes the size of the matrix but leaves the matrix elements in the same place. You can’t assume anything about the value of elements that were not present in the original matrix. A matrix that has no representation simply creates one with the required dimensions. Otherwise, the matrix detaches itself from others and asks for a representation resizing.

\[
\text{Matrix methods 19} + \equiv \\
\begin{align*}
\text{void } & \text{resize(} \text{const index } \text{rows, const index } \text{cols)} \\
\{ & \\
\text{if } & (\neg \text{theRepresentation}) \{ \\
\text{init(} \text{rows, cols);} \\
\text{return;} \\
\} \\
\text{detach();} \\
\text{theRepresentation->resize(rows, cols);} \\
\}
\end{align*}
\]

77. The protocol is as follows: first the representation calls the structure resize method. The structure returns the new dimensions for the storage (or throws a \textbf{dimension} error in case the new size is not compatible). Since the storage will throw an error if the new dimensions are wrong, we can safely update the representation variables before resizing. Next, the representation calls the storage resize method, which will take care of everything else.

\[
\text{Matrix representation methods 18} + \equiv \\
\begin{align*}
\text{void } & \text{resize(} \text{index } \text{rows, index } \text{cols)} \\
\{ & \\
\text{num_rows } & \leftarrow \text{rows;} \\
\text{num_cols } & \leftarrow \text{cols;} \\
\text{theStructure->resize(} & \text{&rows, &cols);} \\
\text{theStorage->resize(rows, cols);} \\
\}
\end{align*}
\]

78. (Unstructured structure methods 27) +\equiv
\[
\begin{align*}
\text{void } & \text{resize(} \text{const index *}, \text{const index *}) \text{ const } \{ \} \\
\end{align*}
\]

79. For the dense resize we will use some tricks. If the matrix can be resized without any memory allocation, then that’s what will be done. This means that in those cases the resizing operation is very fast and that it can happen that resizing a big matrix to a small one doesn’t free any memory. If we need to allocate memory we make sure that \textit{init} won’t erase the old data. We do that by copying it to another dense instance and zeroing out our data pointers.

\[
\text{Dense storage methods 29} + \equiv \\
\begin{align*}
\text{void } & \text{resize(} \text{const index } \text{rows, const index } \text{cols);} \\
\end{align*}
\]
80. (export-waiting big definitions 33) +≡

```cpp
template<class T> void dense⟨T⟩::resize(const index rows, const index cols)
{
    if (cols ≤ max_cols ∧ rows ≤ max_rows) {
        num_rows ← rows;
        num_cols ← cols;
        return;
    }
    dense backup(*this);
    elements ← 0;
    data ← 0;
    init(rows, cols);
    for (index j ← 1; j ≤ cols ∧ j ≤ backup.num_cols; ++j)
        for (index i ← 1; i ≤ rows ∧ i ≤ backup.num_rows; ++i) data[j][i] ← backup.data[j][i];
}
```
81. **Submatrices.** Our next task is to provide a way to access submatrices, an operation that is very handy in many cases. A submatrix behaves like an element in the sense that it does not have any storage of its own, only a pointer to a matrix. The difference is that, while an element is a pointer to only one matrix entry, a submatrix is a pointer to a matrix block. The internal variables are the matrix the block refers to and the range, in the form (using MATLAB notation) \( A(i1: i2, j1: j2) \). The basic design decision is that a submatrix, although a pointer to a generic matrix, produces only unstructured matrices when algebraic operations are performed. An example will clarify this necessity: suppose you have a submatrix of a symmetric matrix, and that this submatrix is not symmetric. What happens when you multiply this submatrix by a number is that you should return a unstructured submatrix.

```c++
#define submatrix_template(M) class SUBM {}).M
⟨Submatrix definition 81⟩ ≡
  template<class T>
  class submatrix {
  ⟨Submatrix internal variables 83⟩
  public:
  ⟨Submatrix methods 85⟩
  }

This code is used in section 3.
```

82. In order to be able to use submatrices from matrices (recall that the submatrix definition comes after the matrix one in the header file) we need to predeclare the submatrix template. We could have done this the other way around, but this way seems to me to be simpler.

```c++
⟨Basic definitions 4⟩ +≡
  template<class T> class submatrix;
```

83. In order to be able to use submatrices and matrices with the same template functions, we need to provide a means of detecting the matrix type, element type and so on. For a submatrix, the matrix_type is always unstructured, so we have an additional definition for the matrix the submatrix is referring to.

```c++
⟨Submatrix internal variables 83⟩ ≡
  typedef T internal_matrix_type;
  typedef typename T::element_type element_type;
  typedef submatrix⟨T⟩ submatrix_type;
  typedef matrix⟨element_type, unstructured, dense⟩ matrix_type;

See also section 84.
This code is used in section 81.
```

84. The only data that is needed is a pointer to the matrix and the submatrix corners coordinates.

```c++
⟨Submatrix internal variables 83⟩ +≡
  internal_matrix_type *theMatrix;
  index i1, i2, j1, j2;
```
85. A submatrix is not intended to be used as a user variable, and is also not intended to be constructed
without arguments. This “not intended” is strong, in the sense that no care is currently taken to insure
that the pointer a submatrix stores is valid in any sense. A submatrix is intended to use immediately
as a shorthand for assignments and one-line formula references. We will adopt, for submatrices, a (0,0)
internal origin standard in order to make assignments more efficient. One of the annoying things in defining
constructors is that we have to use const_cast in some cases (if you are passing a submatrix as a
const argument of some function you’ll need this unless you want your screen filled with “discards const” warnings).

\{
\text{Submatrix methods \(85\)} \equiv
\begin{align*}
\text{submatrix}(\text{void}): & \text{theMatrix}() \\
& \text{\{ throw error::generic("Default\_constructor\_of\_submatrix\_should\_not\_be\_used!"); \}} \\
\text{submatrix}(\text{const internal\_matrix\_type } \ast \text{mat, const index row1, const index row2, const index col1 } ← 1, \text{const index col2 } ← 1) \\
& \text{\{ } \\
& \quad \text{theMatrix } ← \text{const\_cast}\langle\text{internal\_matrix\_type } \ast \rangle (\text{mat}); \\
& \quad i1 ← \text{row1} - 1; \\
& \quad i2 ← \text{row2} - 1; \\
& \quad j1 ← \text{col1} - 1; \\
& \quad j2 ← \text{max}(\text{col2}, \text{col1}) - 1; \\
& \text{\}}
\end{align*}
\}

See also sections 86, 87, 90, 91, 93, 98, 101, and 104.

This code is used in section 81.

86. A submatrix can be assigned values in two ways: by a matrix or by a submatrix. We deal with the
former first for simplicity.

\{
\text{Submatrix methods \(85\)} \equiv
\begin{align*}
\text{submatrix}(\text{const internal\_matrix\_type } \& \text{mat}) \{ \ast\text{this } ← \text{mat}; \} \\
\text{submatrix\_type } \& \text{operator}\langle\text{const internal\_matrix\_type } \& \text{mat}\rangle \\
& \text{\{ } \\
& \quad \text{if } (\text{mat.rows()} \neq i2 - i1 + 1 \lor \text{mat.cols()} \neq j2 - j1 + 1) \text{ throw error::dimension(); } \\
& \quad \text{for } (\text{index } i ← 1; \ i \leq \text{mat.rows()}; \ ++i) \\
& \quad \quad \text{for } (\text{index } j ← 1; \ j \leq \text{mat.cols()}; \ ++j) \ \text{theMatrix}-\text{set}(i1 + i, j1 + j, \text{mat.get}(i, j)); \\
& \quad \text{return } \ast\text{this}; \\
& \text{\}}
\end{align*}
\}

87. For the later type we need to make sure we are protected against things like \(A(1 : 2, 1 : 2) = A(2 : 3, 2 : 3)\). In order to do that we will need to create a matrix from a submatrix. The thing to be aware of is
that in assigning a submatrix to a matrix we don’t handle the shared representations anymore, instead we
really create a new matrix; but before doing that we need some auxiliary methods from submatrix (note
that the parenthesis operator obeys the (1,1) origin default). These auxiliary methods, together with others
that will be defined later, will make a submatrix behave like a matrix for the basic operations.

\{
\text{Submatrix methods \(85\)} \equiv
\begin{align*}
\text{index rows(}\text{void}) \text{const } \{ \ \text{return } i2 - i1 + 1; \} \\
\text{index cols(}\text{void}) \text{const } \{ \ \text{return } j2 - j1 + 1; \} \\
\text{element\_type } \text{operator}\langle\text{const index i, const index j } ← 1\rangle \text{ const} \\
& \text{\{ } \\
& \quad \text{return } \text{theMatrix}-\text{get}(i1 + i, j1 + j); \} \\
& \text{\}}
\end{align*}
\}

88. \text{\{Matrix internal types \(14\)} \equiv
\begin{align*}
\text{typedef submatrix(matrix\_type) submatrix\_type;}
\end{align*}
\}

89. The following matrix methods are auxiliary to submatrix, but useful in their own right.

\[ \text{Matrix methods} \] +≡

```cpp
template<class SUBM> matrix(const submatrix<SUBM> &mat) : theRepresentation(0) { *this ← mat; }
```

```cpp
template<class SUBM> matrix_type &operator←(const submatrix<SUBM> &mat)
{
    init(mat.rows(), mat.cols());
    for (index i ← 1; i ≤ rows(); ++i)
        for (index j ← 1; j ≤ cols(); ++j) set(i, j, mat(i, j));
    return *this;
}
```

90. Now to the last type of assignment. We need to check if our matrix is the same as the one being assigned, and if so if there is intersection between the source and destination elements. If that’s the case we create a new matrix based on the source and copy from this matrix.

\[ \text{Submatrix methods} \] +≡

```cpp
submatrix(const submatrix_type &mat) { *this ← mat; }
```

```cpp
submatrix_type &operator←(const submatrix_type &mat)
{
    if (theMatrix ≡ mat.theMatrix ∧ j1 ≤ mat.j2 ∧ j2 ≥ mat.j1 ∧ i1 ≤ mat.i2 ∧ i2 ≥ mat.i1)
        *this ← internal_matrix_type(mat);
    else
        for (index i ← 1; i ≤ rows(); ++i)
            for (index j ← 1; j ≤ cols(); ++j) theMatrix.set(i1 + i, j1 + j, mat(i, j));
    return *this;
}
```

91. In order to make the submatrix type to resemble a matrix we define set and get methods for it.

\[ \text{Submatrix methods} \] +≡

```cpp
void set(index row, index col, element_type value) { theMatrix.set(row + i1, col + j1, value); }
```

```cpp
const element_type get(index row, index col ← 1) { return theMatrix.get(row + i1, col + j1); }
```

92. All we need now is a means to get a submatrix from a matrix. Since we can’t overload the ‘:’ operator, there is no good way to create one except by a specialized method (instead of overloading the parenthesis operator). We maintain some similarity with MATLAB in the sense that the first two arguments define the row range (and not the upper corner). The default behavior for the last column (which is to get the value of the starting column if that is greater) is handy sometimes. Note that the method is declared const even if it isn’t. The reason is the same as the one described when defining submatrix constructors.

\[ \text{Matrix methods} \] +≡

```cpp
submatrix_type subm(const index row1, const index row2, const index col1 ← 1, index 
                     col2 ← 1) const
{
    col2 ← max(col2, col1);
    return submatrix_type(this, row1, row2, col1, col2);
}
```
Again we provide a `subm` method for submatrices in order to make them resemble an actual matrix.

```cpp
submatrix_type subm(const index row1, const index row2, const index col1 ← 1, index col2 ← 1) const
{
    col2 ← max(col2, col1);
    return theMatrix-subm(i1 + row1, i1 + row2, j1 + col1, j1 + col2);
}
```
94. Basic algebraic operations. We are now able to define the most basic algebraic operations, such as sum, multiplication by scalars, and so on. We begin the unary operators. The methods are straightforward, but it is not as elegant as one might want it to be because we have to use the set and get methods: it is possible to use the already defined entry, but it could be slower, and the parenthesis operator is very cumbersome to use with a pointer (this). A common characteristic of the methods that follow is that the outer loop is generally the loop on the columns. We do that because we know that the unstructured matrix stores by column, so that it is more likely that the computer will make better use of the fast cache memory if we access the matrix this way. Let’s start with scalar multiplication:

\[\text{matrix type } \& \text{operator } *=(\text{const T } \& \text{value})\]

\[
\begin{align*}
&\quad \text{for (index } j \leftarrow 1; j \leq \text{cols()}; ++j) \\
&\quad \quad \text{for (index } i \leftarrow 1; i \leq \text{rows()}; ++i) \text{ set}(i, j, \text{value } \times \text{get}(i, j)); \\
&\quad \text{return } \ast \text{this};
\end{align*}
\]

95. (Matrix element methods 68) \(+\equiv\)

\[\text{element type operator } *= (\text{const element type } \& \text{val})\]

\[
\begin{align*}
&\quad \ast \text{this } \leftarrow \text{theMatrix-\text{get}(i, j) } \times \text{val}; \\
&\quad \text{return } \text{value();}
\end{align*}
\]

96. (Matrix methods 19) \(+\equiv\)

\[\text{matrix type } \& \text{operator } /= (\text{const T } \& \text{value})\]

\[
\begin{align*}
&\quad \text{for (index } j \leftarrow 1; j \leq \text{cols()}; ++j) \\
&\quad \quad \text{for (index } i \leftarrow 1; i \leq \text{rows()}; ++i) \text{ set}(i, j, \text{get}(i, j)/\text{value}); \\
&\quad \text{return } \ast \text{this};
\end{align*}
\]

97. (Matrix element methods 68) \(+\equiv\)

\[\text{element type operator } /= (\text{const element type } \& \text{val})\]

\[
\begin{align*}
&\quad \ast \text{this } \leftarrow \text{theMatrix-\text{get}(i, j)/\text{val};} \\
&\quad \text{return } \text{value();}
\end{align*}
\]

98. (Submatrix methods 85) \(+\equiv\)

\[\text{submatrix type } \& \text{operator } /= (\text{const element type } \& \text{value}) \{
\]

\[
\begin{align*}
&\quad \text{for (index } j \leftarrow 1; j \leq \text{cols()}; ++j) \\
&\quad \quad \text{for (index } i \leftarrow 1; i \leq \text{rows()}; ++i) \text{ theMatrix-\text{entry}(i1 } + \text{i, j1 } + j ) /= \text{value;} \\
&\quad \text{return } \ast \text{this;}
\end{align*}
\]
99. Next we define the unary addition operator. In this case we test for dimension - we assume that the computational overhead in doing this is negligible when compared to the sum itself.

\[ \langle \text{Matrix methods 19} \rangle \quad + \equiv \]

\[
\text{template}<\text{submatrix}_\text{template}(A)>
\]

\[
\text{matrix}_\text{type} \ &\text{operator}+=(\text{const} \ \text{SUBMA} \ &\text{value})
\]

\[
\quad \{
\quad \quad \text{if} \ (\text{rows}() \neq \text{value}.\text{rows}() \lor \text{cols}() \neq \text{value}.\text{cols}()) \ \text{throw} \ \text{error}::\text{dimension}();
\quad \quad \text{for} \ (\text{index} \ j \leftarrow 1; \ j \leq \text{cols}(); \ ++j)
\quad \quad \quad \text{for} \ (\text{index} \ i \leftarrow 1; \ i \leq \text{rows}(); \ ++i) \ \text{set}(i, j, \text{get}(i, j) + \text{value}(i, j));
\quad \quad \text{return} \ \ast\text{this};
\quad \}
\]

100. \[ \langle \text{Matrix element methods 68} \rangle \quad + \equiv \]

\[
\text{element}_\text{type} \ &\text{operator}+=(\text{const} \ \text{element}_\text{type} \ &\text{val})
\]

\[
\quad \{
\quad \quad \ast\text{this} \leftarrow \text{theMatrix}.\text{get}(i, j) + \text{val};
\quad \quad \text{return} \ \text{value}();
\quad \}
\]

101. \[ \langle \text{Submatrix methods 85} \rangle \quad + \equiv \]

\[
\text{submatrix}_\text{type} \ &\text{operator}+=(\text{const} \ \text{matrix}_\text{type} \ &\text{value})
\]

\[
\quad \{
\quad \quad \text{if} \ (\text{value}.\text{rows}() \neq \text{rows}() \lor \text{value}.\text{cols}() \neq \text{cols}()) \ \text{throw} \ \text{error}::\text{dimension}();
\quad \quad \text{for} \ (\text{index} \ j \leftarrow 1; \ j \leq \text{cols}(); \ ++j)
\quad \quad \quad \text{for} \ (\text{index} \ i \leftarrow 1; \ i \leq \text{rows}(); \ ++i) \ \text{set}(i1 + i, j1 + j) += \text{value}.\text{get}(i, j);
\quad \quad \text{return} \ \ast\text{this};
\quad \}
\]

102. \[ \langle \text{Matrix methods 19} \rangle \quad = \equiv \]

\[
\text{template}<\text{submatrix}_\text{template}(A)>
\]

\[
\text{matrix}_\text{type} \ &\text{operator}--=(\text{const} \ \text{SUBMA} \ &\text{value})
\]

\[
\quad \{
\quad \quad \text{if} \ (\text{rows}() \neq \text{value}.\text{rows}() \lor \text{cols}() \neq \text{value}.\text{cols}()) \ \text{throw} \ \text{error}::\text{dimension}();
\quad \quad \text{for} \ (\text{index} \ j \leftarrow 1; \ j \leq \text{cols}(); \ ++j)
\quad \quad \quad \text{for} \ (\text{index} \ i \leftarrow 1; \ i \leq \text{rows}(); \ ++i) \ \text{set}(i, j, \text{get}(i, j) - \text{value}(i, j));
\quad \quad \text{return} \ \ast\text{this};
\quad \}
\]

103. \[ \langle \text{Matrix element methods 68} \rangle \quad = \equiv \]

\[
\text{element}_\text{type} \ &\text{operator}--=(\text{const} \ \text{element}_\text{type} \ &\text{val})
\]

\[
\quad \{
\quad \quad \ast\text{this} \leftarrow \text{theMatrix}.\text{get}(i, j) - \text{val};
\quad \quad \text{return} \ \text{value}();
\quad \}
\]
104. ⟨Submatrix methods 85⟩ ⊆

\[
\text{submatrix}_\text{type} \& \text{operator} \equiv (\text{const matrix}_\text{type} \& \text{value})
\]

\[
\text{if } (\text{value}.\text{rows()} \neq \text{rows()} \lor \text{value}.\text{cols()} \neq \text{cols()}) \text{ throw error}::\text{dimension}();
\]

\[
\text{for } (\text{index } j \leftarrow 1; j \leq \text{cols(); } ++j)
\]

\[
\text{for } (\text{index } i \leftarrow 1; i \leq \text{rows(); } ++i) \text{ theMatrix-} \text{entry}(i1 + i, j1 + j) \equiv \text{value}.\text{get}(i, j);
\]

\[
\text{return } *\text{this};
\]

105. The transpose operator has to be defined outside, because we’ll want to specialize it (for example, for a symmetric matrix it’s a void method, for a upper triangular it returns another type of matrix, and so on).

⟨Basic algebraic operations 105⟩ ≡

\[
\text{template}<\text{matrix_simple_template}>
\]

\[
\text{matrix}\langle\text{T, structure, storage}\rangle \text{ transpose}(\text{const matrix}\langle\text{T, structure, storage}\rangle \& x)
\]

\[
\{
\text{matrix}\langle\text{T, structure, storage}\rangle \text{ result}(x.\text{cols()}, x.\text{rows()});
\]

\[
\text{for } (\text{index } j \leftarrow 1; j \leq x.\text{cols(); } ++j)
\]

\[
\text{for } (\text{index } i \leftarrow 1; i \leq x.\text{rows(); } ++i) \text{ result.set}(j, i, x(i, j));
\]

\[
\text{return result};
\]

\[
\text{template}<\text{matrix_simple_template} \rangle \text{ matrix}\langle\text{T, structure, storage}\rangle \text{ transpose(\text{const submatrix}\langle\text{T, structure, storage}\rangle \& x)}
\]

\[
\{
\text{matrix}\langle\text{T, structure, storage}\rangle \text{ result}(x.\text{cols()}, x.\text{rows()});
\]

\[
\text{for } (\text{index } j \leftarrow 1; j \leq x.\text{cols(); } ++j)
\]

\[
\text{for } (\text{index } i \leftarrow 1; i \leq x.\text{rows(); } ++i) \text{ result.set}(j, i, x(i, j));
\]

\[
\text{return result};
\]

This code is used in section 3.
106. **Basic specializations and utilities.** We're done with the matrix type and the basic structure and storage types. You could, using what we wrote, code anything from a matrix multiplication procedure to a complete eigenvalue/eigenvector decomposition. Sometimes, however, it is good to have some utilities predefined, and that’s what this section is about.

While writing code for LU decomposition I felt the need for a swap algorithm for matrix elements. Since we cannot use the parenthesis operator for assignment, the STL's swap function doesn’t work. The reason is interesting: a swap function must store one value in a temporary variable. The STL's swap assumes that this temporary variable has the same type as the function arguments, so it does something like \( \text{element aux} \leftarrow A \). But, for our purposes, this is useless – the value of the element was not saved, only the matrix instance and the index to the entry! Hence, the need to specialize.

\[
\langle \text{Specializations 106} \rangle \equiv \\
\begin{align*}
\text{template} & \langle \text{class T} \rangle \ \text{inline void swap}(\text{element}\langle T \rangle \ x, \text{element}\langle T \rangle \ y) \\
& \{ \\
& \quad \text{typename T::element_type aux} \leftarrow x.\text{value}(); \\
& \quad x \leftarrow y.\text{value}(); \\
& \quad y \leftarrow aux;
\}
\end{align*}
\]

See also sections 109, 110, 111, and 116.

This code is used in section 3.

107. \( \langle \text{Matrix element methods 68} \rangle +\equiv \)

\[
\begin{align*}
\text{inline element\_type value}(\text{void}) \ &\text{const} \ \{ \ \text{return theMatrix-get}(i,j); \ \}
\end{align*}
\]

108. Of course, I needed a swap method because I wanted to swap rows. This operation is useful in many other places, so I add a method that does exactly this.

\[
\langle \text{Matrix methods 19} \rangle +\equiv
\]

\[
\begin{align*}
\text{void swaprows}(\text{const index} \ i, \text{const index} \ k) \\
& \{ \\
& \quad \text{for (index } j \leftarrow 1; \ j \leq \text{cols}(); \ ++j) \ \text{swap(entry}(i,j),\text{entry}(k,j)); \\
& \}
\end{align*}
\]

109. Comparisons between matrices can be made in two ways: either we compare their elements one by one or we check if they share the same representation. The next method does exactly that, and it’s useful in many contexts (there’s a similar function in Lisp, for example, where we can have shared objects).

\[
\langle \text{Specializations 106} \rangle +\equiv
\]

\[
\begin{align*}
\text{template} & \langle \text{matrix\_simple\_template} \rangle \\
\text{bool} & \ \text{same}(\text{const matrix}\langle T, \text{structure, storage} \rangle \ &x, \text{const matrix}\langle T, \text{structure, storage} \rangle \ &y) \\
& \{ \\
& \quad \text{return } (x.\text{theRepresentation} \equiv y.\text{theRepresentation}); \\
& \}
\end{align*}
\]

110. Next we define a new type of pair, a pair of indexes. We can use this pair for using stl's map with a less comparison function.

\[
\langle \text{Specializations 106} \rangle +\equiv
\]

\[
\begin{align*}
\text{typedef} & \ \text{pair}(\text{math::index}, \text{math::index}) \text{ pair};
\end{align*}
\]

111. \( \langle \text{Specializations 106} \rangle +\equiv \)

\[
\begin{align*}
\text{bool} & \ \text{operator}< (\text{const math::pair} \ &x, \text{const math::pair} \ &y);
\end{align*}
\]
§112. \( \langle \text{Big definitions} 112 \rangle \equiv \)

\[
\begin{align*}
\text{bool operator } < (\text{const math::pair } &x, \text{const math::pair } &y) \\
\{ \\
\text{if } (x._\text{first} > y._\text{first}) \text{ return } \text{false}; \\
\text{if } (x._\text{first} < y._\text{first}) \text{ return } \text{true}; \\
\text{return } (x._\text{second} < y._\text{second}); \\
\}
\end{align*}
\]

This code is used in section 2.

§113. \( \langle \text{Include files} \text{ math 6} \rangle \equiv \)

\[
\text{\#include <utility>}
\]

§114. We will also interface this library with LAPACK at some point. In order to define the necessary Fortran stuff we include a file automatically generated during the library build. The file contains some LAPACK function prototypes and some definitions to take care of system-dependent LAPACK features (in some systems the function names need underscore, for example). This file will also have the \text{LAPACK} definition built-in.

\( \langle \text{Include files} \text{ math 6} \rangle \equiv \)

\[
\text{\#include <math/private/fortran.h>}
\]

§115. While we are at it, some sparse structures used in other programs (such as PCx) have a field with the number of nonzero elements in the matrix. In order to make interface easier we define a method that returns this number.

\( \langle \text{Matrix methods} 19 \rangle \equiv \)

\[
\begin{align*}
\text{index } \text{numnonzeros}(\text{void}) \text{ const} \\
\{ \\
\text{index } \text{result} \leftarrow 0; \\
\text{for } (\text{index } j \leftarrow 1; j \leq \text{cols}(); ++j) \\
\text{for } (\text{index } i \leftarrow 1; i \leq \text{rows}(); ++i) \\
\text{if } (\text{get}(i,j)) \text{ result}++; \\
\text{return } \text{result}; \\
\}
\end{align*}
\]

§116. The \text{reshape} function works exactly as in Matlab (that is, the new matrix has the same elements taken columnwise from the original). We define it only for \text{unstructured} and \text{dense} matrices for now. We use our knowledge about the \text{dense} storage to make this operation very fast. The source matrix is overwritten with the new one.

\( \langle \text{Specializations} 106 \rangle \equiv \)

\[
\begin{align*}
\text{template<class } T \text{> } \\
\text{matrix(} T, \text{unstructured, dense} \text{) } \&\text{reshape(matrix(} T, \text{unstructured, dense} \text{) } \ast x, \text{index } \text{rows, index } \text{cols}) \\
\{ \\
\text{if } (x._\text{rows}(\text{)} \ast x._\text{cols}(\text{)} \neq \text{rows} \ast \text{cols}) \text{ throw error::dimension();} \\
\text{x-stor}()\text{-init(rows, cols);} \\
\text{x-rep}()\text{-rows()} \leftarrow \text{rows}; \\
\text{x-rep}()\text{-cols()} \leftarrow \text{cols}; \\
\text{return } \ast x; \\
\}
\end{align*}
\]
117. **Algebraic operations.** We now reach the point where serious optimization begins: basic algebraic operations are the heart of any matrix package, and if your basic operations suck your entire package will suck too. Since we’re defining generic interfaces, we can’t possibly reach state-of-the-art performance – but we can get pretty close.

```c
/
Empty, waiting for export */
```

118. ⟨algebra.h 118⟩ ≡
```
 ifndef __MATH_ALGEBRA__
 define __MATH_ALGEBRA__ 1.0
 include <math.h> /* For sqrt. */
 include <math/math.h>
 namespace math {
 ⟨Algebraic operations 119⟩
 }
 endif
```

119. We begin our definitions with binary algebraic operators. Unary operators were defined already. These are a little trickier than unary operators because we have structures: the matrix returned can have a different type of structure than that of the arguments (for example, a symmetric matrix times a symmetric matrix is not necessarily symmetric). We therefore start with scalar operations because this problem does not arise.

```c
 ⟨Algebraic operations 119⟩ ≡
 template(matrix_simple_template) matrix⟨T,structure,storage⟩ operator∗(const
 matrix⟨T,structure,storage⟩ &x, const T &y)
 {
 matrix⟨T,structure,storage⟩ result ← x;
 return result *= y;
 }
```

See also sections 120, 121, 122, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, and 134.
This code is used in section 118.

120. ⟨Algebraic operations 119⟩ +≡
```
 template(matrix_simple_template) matrix⟨T,structure,storage⟩ operator+(const
 submatrix⟨matrix⟨T,structure,storage⟩⟩ &x, const T &y)
 {
 matrix⟨T,structure,storage⟩ result ← x;
 return result += y;
 }
```

121. ⟨Algebraic operations 119⟩ +≡
```
 template(matrix_simple_template) matrix⟨T,structure,storage⟩ operator+(const
 matrix⟨T,structure,storage⟩ &x, const matrix⟨T,structure,storage⟩ &y)
 {
 if (x.rows() ̸= y.rows() ∨ x.cols() ̸= y.cols()) throw error::dimension();
 matrix⟨T,structure,storage⟩ result ← x;
 return result += y;
 }
```
122. The first function we define is one that overwrites a matrix $Y$ with the value of $\alpha x + y$. The name saxpy is of common use (it’s used, for example, in Golub and Van Loan and in the BLAS package), so we keep it. Note that with this method the variable $y$ is always overwritten, although some uses can be visually misleading. Also, it is the programmer who has to be sure that the structure of $y$ and $\alpha x + y$ is the same (or compatible), since $y$ is overwritten.

\begin{verbatim}
template<class T, submatrix_template(X), submatrix_template(Y)>
SUBMY &saxpy(const T &a, const SUBMX &x, SUBMY *y)
{
    if (y->rows() != x->rows() || y->cols() != x->cols()) throw error::dimension();
    for (index j ← 1; j ≤ y->cols(); ++j)
        for (index i ← 1; i ≤ y->rows(); ++i) y->set(i, j, a * x(i, j) + y->get(i, j));
    return *y;
}
\end{verbatim}

123. Next we define the dot product. This method only makes sense for vectors, and we test the dimensions before proceeding.

\begin{verbatim}
template<submatrix_template(X), submatrix_template(Y)>
typename SUBMX::element_type dot(const SUBMX &x, const SUBMY &y)
{
    if (x.cols() != 1 || y.cols() != 1 || x.rows() != y.rows()) throw error::dimension();
    typename SUBMX::element_type result ← 0;
    for (index i ← 1; i ≤ x.rows(); ++i) result += x(i) * y(i);
    return result;
}
\end{verbatim}

124. Sometimes we want to multiply a row vector by a column vector. We call this operation the “transposed dot” operation. Here $x$ must be a row vector.

\begin{verbatim}
template<submatrix_template(X), submatrix_template(Y)>
typename SUBMX::element_type tdot(const SUBMX &x, const SUBMY &y)
{
    if (x.rows() != 1 || y.cols() != 1 || x.cols() != y.rows()) throw error::dimension();
    typename SUBMX::element_type result ← 0;
    for (index i ← 1; i ≤ x.cols(); ++i) result += x(1, i) * y(i);
    return result;
}
\end{verbatim}
And, just to make you nervous, sometimes we want to take the “dot product” of two row vectors. We call this operation the “transposed-transposed dot” operation. Here x and y must be row vectors.

```cpp
template<submatrix_template(X), submatrix_template(Y)>
typename SUBMX::element_type ttdot(const SUBMX &x, const SUBMY &y)
{
    if (x.rows() != 1 || y.rows() != 1 || x.cols() != y.cols()) throw error::dimension();
    typename SUBMX::element_type result = 0;
    for (index i = 1; i <= x.cols(); ++i) result += x(i) * y(i);
    return result;
}
```

Instead of defining the inner product of a vector we define the 2-norm operator. Since we’re calling the dot function we don’t need to check dimensions here.

```cpp
template<submatrix_template(X)>
typename SUBMX::element_type norm2(const SUBMX &x)
{
    return (sqrt(dot(x, x)));
}
```

The following method returns the result of the operation Ax, where x is a vector or a matrix. The third argument is the destination matrix.

```cpp
template<submatrix_template(A), submatrix_template(X), matrix_simple_template>
matrix(T, structure, storage) &axmul(const SUBMA &A, const SUBMX &x, matrix(T, structure, storage) *dest)
{
    index n = A.rows();
    index m = x.cols();
    index p = x.rows();
    if (A.cols() != p) throw error::dimension();
    dest->resize(n, m);
    for (index i = 1; i <= n; ++i)
    {
        for (index j = 1; j <= m; ++j) {
            T aux = 0.0;
            for (index k = 1; k <= p; ++k) aux += A(i, k) * x(k, j);
            dest->entry(i, j) = aux;
        }
    }
    return *dest;
}
```
128. And this one returns the result of the operation $A^T x$, where $x$ can be a matrix. The third argument is the destination matrix.

Algebraic operations 119) +≡

```cpp
template<submatrix_template(A), submatrix_template(X), matrix_simple_template>
matrix(T, structure, storage) &atmul(const SUBMA &A, const SUBMX &x, matrix(T, structure, storage) *dest)
{
    index n ← A.cols();
    index m ← x.cols();
    index p ← x.rows();
    if (A.rows() ≠ p) throw error::dimension();
    dest.resize(n, m);
    for (index i ← 1; i ≤ n; ++i)
        for (index j ← 1; j ≤ m; ++j) {
            T aux ← 0.0;
            for (index k ← 1; k ≤ p; ++k) aux += A(k, i) * x(k, j);
            dest.entry(i, j) ← aux;
        }
    return *dest;
}
```

129. Next we present the generalized saxpy. In this method, the value $a$ of saxpy is substituted by a matrix, that is, now we have $Y = AX + Y$. This method is normally heavily used, and it involves a matrix multiplication – it’s a target of serious optimization in some libraries. Our generic version must be, well, generic. It shouldn’t be much worse than routines for dense and unstructured matrices, however.

Algebraic operations 119) +≡

```cpp
template<submatrix_template(A), submatrix_template(X), submatrix_template(Y)>
SUBMY &gatxpy(const SUBMA &A, const SUBMX &x, SUBMY *y)
{
    if (y.rows() ≠ A.rows() ∨ y.cols() ≠ x.cols() ∨ x.rows() ≠ A.cols()) throw error::dimension();
    for (index j ← 1; j ≤ y.cols(); ++j)
        for (index i ← 1; i ≤ y.rows(); ++i)
            y.set(i, j, y.get(i, j) + tdot(A.subm(i, i, 1, A.cols()), x.subm(1, x.rows(), j)));
    return *y;
}
```

130. We also define a method in case what we want to do is to compute $Y = A^T X + Y$.

Algebraic operations 119) +≡

```cpp
template<submatrix_template(A), submatrix_template(X), submatrix_template(Y)>
SUBMY &gatxpy(const SUBMA &A, const SUBMX &x, SUBMY *y)
{
    if (y.rows() ≠ A.cols() ∨ y.cols() ≠ x.cols() ∨ x.rows() ≠ A.rows()) throw error::dimension();
    for (index j ← 1; j ≤ y.cols(); ++j)
        for (index i ← 1; i ≤ y.rows(); ++i)
            y.set(i, j, y.get(i, j) + dot(A.subm(1, A.rows(), i), x.subm(1, A.rows(), j)));
    return *y;
}
```
131. The outer product operation (returns $X \cdot X^T$) is sometimes useful.

```cpp
template<typename T, std::vector<int> &rows, std::vector<int> &cols, T *dest, T *aux>
void outerp(const SUBMX &x, T *dest) {
    dest->resize(x.cols(), x.rows());
    T aux;
    for (index i ← 1; i ≤ x.rows(); ++i)
        for (index j ← i; j ≤ x.cols(); ++j) {
            aux ← tdot(x.subm(i, i, 1, x.cols()), x.subm(j, j, 1, x.cols()));
            dest-entry(i, j) ← aux;
        }
    return *dest;
}
```

132. And just to facilitate, we enable the operation $X^T \cdot X$ too.

```cpp
template<typename T, std::vector<int> &rows, std::vector<int> &cols, T *dest, T *aux>
void outerp(const SUBMX &x, T *dest) {
    dest->resize(x.cols(), x.cols());
    T aux;
    for (index i ← 1; i ≤ x.cols(); ++i)
        for (index j ← i; j ≤ x.cols(); ++j) {
            aux ← dot(x.subm(1, x.rows(), i, i), x.subm(1, x.rows(), j, j));
            dest-entry(i, j) ← aux;
        }
    return *dest;
}
```

133. And finally a function to compute $xy^T + yx^T$, where $x$ and $y$ are vectors.

```cpp
template<typename Y, std::vector<int> &rows, std::vector<int> &cols, T *dest, T *aux, Y *y>
void outyxyx(const SUBMX &x, const SUBMY &y, T *dest) {
    if (y.cols() ≠ 1 ∨ x.cols() ≠ 1 ∨ x.rows() ≠ y.rows()) throw error::dimension();
    index n ← x.rows();
    dest->resize(n, n);
    for (index i ← 1; i ≤ n; ++i)
        for (index j ← 1; j ≤ n; ++j) dest-entry(i, j) ← x(i) * y(j) + x(j) * y(i);
    return *dest;
}
```
134. Now to the outer product update. The function overwrites the matrix $A$ with $A + \beta xy^T$.

\[
\text{template} \langle \text{submatrix\_template}(A), \text{submatrix\_template}(X), \text{submatrix\_template}(Y) \rangle \text{ SUBMA}
\]
\[
& \text{outerp\_update}(\text{SUBMA} \ast A, \text{typename SUBMA}\_::\text{element\_type} \ beta, \text{const SUBMX} \ & x, \text{const SUBMY} \ & y)
\]

\[
\text{if } (y.\text{cols}() \neq 1 \lor x.\text{cols}() \neq 1 \lor A\_\text{rows}() \neq x.\text{rows}() \lor A\_\text{cols}() \neq y.\text{rows}())
\]
\[
\text{throw error} :: \text{dimension}();
\]
\[
\text{index} \ n \leftarrow A\_\text{rows}();
\]
\[
\text{index} \ m \leftarrow A\_\text{cols}();
\]
\[
\text{for} (\text{index} \ j \leftarrow 1; j \leq m; ++j)
\]
\[
\text{for} (\text{index} \ i \leftarrow 1; i \leq n; ++i) \ A\_\text{set}(i, j, A\_\text{get}(i, j) + \beta \ast x(i) \ast y(j));
\]
\[
\text{return } *A;
\]
135. **The sparse storage.** Now that we have the basics defined, let us exemplify how to create new types of matrices by defining a new storage type.

/* Empty, waiting for export */

136. \( \langle \text{sparse.h} \ 136 \rangle \equiv \)

```cpp
#ifndef __MATH_SPARSE__
define __MATH_SPARSE__ 1.0

\( \langle \text{Include files sparse} \ 139 \rangle \)
namespace math {
  \( \langle \text{Sparse storage definition} \ 137 \rangle \)
}
#endif

137. \( \langle \text{Sparse storage definition} \ 137 \rangle \equiv \)

```cpp
template<class T>
class sparse {
  \( \langle \text{Sparse storage internal variables} \ 138 \rangle \)
  public:
    \( \langle \text{Sparse storage methods} \ 140 \rangle \)
};
```

This code is used in section 136.

138. A sparse matrix stores its elements in a map in which the key is the element index. To simplify notation we define a new variable for the map type.

\( \langle \text{Sparse storage internal variables} \ 138 \rangle \equiv \)

```cpp
typef map<math::pair, T, less<math::pair>> storage;
storage *elements;
```

This code is used in section 137.

139. \( \langle \text{Include files sparse} \ 139 \rangle \equiv \)

```cpp
#include <math/math.h>  /* For math::pair – otherwise unnecessary. */
#include <map>
```

This code is used in section 136.

140. Now to the interesting part, that is, to the definition of methods that a matrix storage must have. We begin by the constructors and destructor: as with the dense case, we define a default constructor, a “dimension” constructor and a copy constructor. The interesting part is that we don’t need to initialize any data or keep any information whatsoever in the creation phase.

\( \langle \text{Sparse storage methods} \ 140 \rangle \equiv \)

```cpp
sparse(void):elements(0) { }
sparse(const index &rows, const index &cols):elements(0) { elements ← new storage; }
sparse(const sparse &source):elements(0) {
  elements ← new storage;
  elements.insert(source.elements.begin(), source.elements.end());
}
```

See also sections 141, 142, 143, and 144.
This code is used in section 137.

141. \( \langle \text{Sparse storage methods} \ 140 \rangle \equiv \)

```cpp
~sparse() { if (elements) delete elements; }
```

This code is used in section 137.
142. We have now to define how to set and get elements. When setting elements we have to consider setting an element to zero – if the element exists we remove it, otherwise we don’t set anything.

\[ \text{(Sparse storage methods 140) } +\equiv \]

```cpp
    void set(const index &row, const index &col, const T &value)
    {
        math::pair i(row, col);
        if (value != T(0)) (*elements)[i] = value;
        else if (elements->find(i) != elements->end()) elements->erase(i);
    }
```

143. Getting elements is not trivial only because if the default behavior of maps is to create non-existing entry when accessing it. Recall that if either \texttt{row} or \texttt{col} are zero we are supposed to return \texttt{0} – which will happen because there is no element with index \texttt{0,0} stored in a sparse matrix.

\[ \text{(Sparse storage methods 140) } +\equiv \]

```cpp
    const T get(const index &row, const index &col) const
    {
        math::pair i(row, col);
        if (elements->find(i) != elements->end()) return (*elements)[i];
        return T(0);
    }
```

144. When resizing we let elements outside the new matrix to stay there just to make resizing fast (very fast, just an empty method). It is assumed that you won’t be resizing a sparse matrix to very different sizes.

\[ \text{(Sparse storage methods 140) } +\equiv \]

```cpp
    void resize(const index &rows, const index &cols) { }
```

145. And we are done. The methods listed for this class are the only required methods for a valid matrix storage type. Now that we have the dense and sparse types, however, there is little need for other types, so I consider this work done. The only other kind of storage I can imagine is the block-diagonal storage, but then we have some difficulties (we can have dense and sparse block-diagonal matrices and so on).
The symmetric structure. Now that we have a new storage defined, let us end the lesson on how to define new matrices by creating a new structure.

/* Empty, waiting for export */

⟨symmetric.h 147⟩ ≡
ifndef __MATH_SYMMETRIC__
define __MATH_SYMMETRIC__ 1.0
#include <math/math.h>
namespace math {
  ⟨Symmetric structure definition 148⟩
}
endif

⟨Symmetric structure definition 148⟩ ≡
template ⟨class T⟩
class symmetric {
  public:
    ⟨Symmetric structure methods 149⟩
};

This code is used in section 147.

We start by defining the constructor that takes the dimensions as arguments. A symmetric matrix has to be square, and it can be represented with only $n(n+1)/2$ elements. We will store them as a vector as follows: if we have a matrix

$$A = \begin{bmatrix}
a_{11} & a_{12} & a_{13} \\
* & a_{22} & a_{23} \\
* & * & a_{33}
\end{bmatrix},$$

then we store is as the vector

$$[a_{11} \quad a_{12} \quad a_{22} \quad a_{13} \quad a_{23} \quad a_{33}].$$

This scheme is the same one used in the LAPACK package, so we will use it to facilitate interfacing with it.

⟨Symmetric structure methods 149⟩ ≡
void resize(math::index *rows, math::index *cols)
{
  if (*rows != *cols) throw error::nonsquare();
  *rows ← (*rows) * (*rows + 1)/2;
  *cols ← 1;
}
symmetric(math::index *rows, math::index *cols) { resize(rows, cols); }

This code is used in section 148.
Next we define the preprocess method. After some algebra, you will see that the index of the element \((i, j)\) is given by

\[
\frac{j(j - 1)}{2} + i
\]

if \(j \geq i\), that is, if the element is on the upper triangle of the matrix. In this class these are the elements that are stored. The elements of the lower triangle are not assignable.

```cpp
bool preprocess(math::index *row, math::index *col) const
{
    if (*col >= *row) {
        *row = (*col - 1) * (*col) / 2 + (*row);
        *col = 1;
        return true; /* Assignable. */
    }
    *row = (*row - 1) * (*row) / 2 + (*col);
    *col = 1;
    return false; /* Not assignable. */
}
```

We need a copy constructor, which is empty.

```cpp
symmetric(const symmetric &) {}
```

And we are done. I cannot help feeling proud of the flexibility of this system. In less than two pages we defined a whole new matrix structure, and this is taken into account documentation!
153. **File streams: input.** Files are the most convenient interface with other programs. I chose the MATLAB file format for simple interface with a widely used program. Since MATLAB’s version 4 file format is public we will use it against the new format, which is proprietary and secret and sucks.

```cpp
#include "fstream.h"
```

154. ⟨fstream.h 154⟩ ≡

```cpp
#ifndef __MATH_FSTREAM__
#define __MATH_FSTREAM__ 1.0

#include <math/math.h>
namespace math {
  #include <fstream structures 162>
  #include <fstream declarations 155>
}
#endif
```

155. We will first define the input stream, so that you will be able to load matrices from a MATLAB file. Almost all the work is already done by the standard library `ifstream` class, the only thing we need to do is to specialize it a little bit. We will want, for example, to find a matrix by name.

```cpp
#include <fstream>
```

156. ⟨fstream declarations 155⟩ ≡

```cpp
class ifstream : public std::ifstream {
  public:
    (ifstream methods 157)
};
```

157. The first thing to do is to overload some constructors for compatibility with the standard `ifstream` class.

```cpp
ifstream() :
  std::ifstream() {}
DMETHOD ifstream(int fd) :
  std::ifstream(fd) {}
```

158. Now to the opening of files. A matrix file is binary, so we change the default `ifstream` mode on both the constructor and the `open` method accordingly.

```cpp
ifstream(const char *name, int mode ← ios::in | ios::binary, int prot ← °664)
  :
  std::ifstream(name, mode, prot) {}
```
159. \langle \text{ifstream methods 157} \rangle \equiv
\begin{verbatim}
void open(const char *name, int mode \leftarrow \text{ios::in | ios::binary, int prot \leftarrow \text{\^664}})
{
    std::ifstream::open(name, mode, prot);
}
\end{verbatim}

160. Unfortunately, the binary format of my library does not work for \texttt{long} or \texttt{double}, that is, I cannot use the operator \texttt{\gg} to get these value types from a file. But we can fix it very easily by defining new operators. The trick, in both cases, is to declare an union of \texttt{unsigned char} and \texttt{long} (or \texttt{double}) and read \texttt{unsigned char}s from the file. Since we have the desired type in the union, what we are actually doing is filling in the value. We will only overload the \texttt{\gg} operator for the \texttt{long} case, because it will be convenient when reading matrices headers. For the \texttt{double} case it is simpler to do the trick when reading the matrix data itself.

\begin{verbatim}
ifstream &operator\gg(long &);  
\end{verbatim}

161. \texttt{math::ifstream \&math::ifstream::operator\gg(long \&dest)}
\begin{verbatim}
{
    union {
        long Long;
        unsigned char Char[sizeof(long)];
    } tricky;
    for (int i \leftarrow 0; i \neq sizeof(long); \texttt{this-get(tricky.Char[i++])} ;
        dest \leftarrow tricky.Long;
        return \texttt{*this};
}
\end{verbatim}

162. Now we define a method that will read a matrix header. In MATLAB files, a matrix header consists of a sequence of fields that we will read in a structure as described below.

\begin{verbatim}
struct fheader {
    long type;   /* Type of the object. */
    long rows;   /* Number of rows. */
    long cols;   /* Number of cols. */
    bool iflag;  /* true if matrix has imaginary part. */
    string name; /* Matrix name. */
};
\end{verbatim}

This code is used in section 154.

163. \langle \text{Include files fstream 156} \rangle \equiv
\#include <string>

164. The actual data on a MATLAB file is somewhat different. The three first fields from \texttt{fheader} are the same. The \texttt{iflag} field is a \texttt{long} on the original file, and after it there is a \texttt{long} field with the number of characters in the name (including the terminating '\0') followed by the matrix name. If we encounter an error while parsing the header we return, instead of throwing an error, since this method is not intended for end users.

\begin{verbatim}
void parse_header(fheader &header);
\end{verbatim}
### 165. 

`void math::ifstream::parse_header(math::fheader &header)`

```cpp
{ 
    long name_length;
    long file_flag;
    "this" >> header.type >> header.rows >> header.cols >> file_flag >> name_length;
    if (!rdstate() || !goodbit) return;
    header.iflag = bool(file_flag);
    header.name.assign(name_length, 0); // Reserve space for matrix name.
    for (int i = 0; i < name_length; this->get(header.name[i++]) ); 
}
```

#### 166.

We are now in position to load a matrix. In a MATLAB file the matrix data is stored by columns, in two separate blocks for real and imaginary parts. If the matrix does not have an imaginary part there is only the real block in the file. There is only one complication: if we are to define a single method, then we must call, in case of a complex matrix, the method `set` with a complex argument. Even if this method is not called it must be compiled. Now, if the matrix is of type, say, `double`, then there will be a compiler error, because there is no conversion from `complex` to `double`. The solution to this problem is to define two functions, specializing for complex matrices. First we will define the method for non-complex matrices. In this case we simply disconsider the eventual imaginary part of the file matrix.

```cpp
<ifstream methods 157> +=
    template(matrix_simple_template)
    ifstream &operator>>(matrix(T, structure, storage) &dest)

<ifstream methods 157> +=
{ 
    math::fheader header;
    parse_header(header);
    if (!rdstate() || !goodbit) throw math::error::filerr();
    dest.init(header.rows, header.cols);
    for (int ipart = 0; ipart < header.iflag; ipart++)
        for (int j = 1; j <= header.cols; j++)
            for (int i = 1; i <= header.rows; i++) {
                double number;
                \{ Get number from file, using the same tricky method as for longs 168 \}
                if (!ipart) dest.set(i, j, number);
            }
    return *this;
}
```

#### 167.

Until the compiler accepts `export` we are stuck with the inline method.

```cpp
<ifstream methods 157> +=
{ 
    math::fheader header;
    parse_header(header);
    if (!rdstate() || !goodbit) throw math::error::filerr();
    dest.init(header.rows, header.cols);
    for (int ipart = 0; ipart < header.iflag; ipart++)
        for (int j = 1; j <= header.cols; j++)
            for (int i = 1; i <= header.rows; i++) {
                double number;
                \{ Get number from file, using the same tricky method as for longs 168 \}
                if (!ipart) dest.set(i, j, number);
            }
    return *this;
}
```

#### 168.

```cpp
<ifstream methods 157> +=
{ 
    union {
        double Double;
        unsigned char Char[sizeof(double)];
    } tricky;
    for (int k = 0; k < sizeof(double); tricky.get(Char[k++]) );
    number = tricky.Double;
}
```

This code is used in sections 167 and 170.
Next we do the same thing for complex matrices. In this case there is nothing to worry about: if the file matrix does not have an imaginary part, the matrix itself will have the imaginary part zeroed.

$$\text{ifstream methods 157} \equiv$$

```cpp
template<typename T, Allocator<>>
ifstream &operator>>(matrix<complex<T>, structure, storage> &dest)
```

Until the compiler accepts `export` we are stuck with the inline method.

```cpp
{
math::fheader header;
parse_header(header);
if (rdstate() != goodbit) throw math::error::filerr();
dest.init(header.rows, header.cols);
for (int ipart = 0; ipart < header.iflag; ipart++)
  for (int j = 1; j <= header.cols; j++)
    for (int i = 1; i <= header.rows; i++)
      double number;
      (Get number from file, using the same tricky method as for longs 168)
      if (ipart) dest.set(i, j, number);
      else dest.set(i, j, dest(i, j) + number * std::complex<double>(0, 1));
}
return *this;
}
```

At this point we are able to load a matrix. Sometimes, however, we will need to skip matrices (to load the second one, for example) or to find matrices by name. We begin by defining a method to skip matrices. First we parse the current matrix header and then skip the data using a `skip_data` method.

```cpp
{
math::fheader header;
while (num_matrices--) {
  parse_header(header);
  if (rdstate() != goodbit)
    throw math::error::generic("Too many matrices to skip.");
  skip_data(header);
}
```

Skipping the data is a simple matter of reading the necessary amount of bytes from the body of the matrix.

```cpp
void skip_data(const fheader &header);
```
175. void math::ifstream::skip_data(const math::fheader &header)
{
    long data_size ← header.cols * header.rows * sizeof(double);
    if (header.iflag) data_size *= 2;
    for (unsigned char garbage; data_size; data_size--) this→get(garbage);
}

176. With the above definitions we can jump to an arbitrary matrix on the file. The first matrix has position 1.
⟨ifstream methods 157⟩ ≡
void skipto(const int position);

177. void math::ifstream::skipto(int position)
{
    seekg(0);
    if (−−position > 0) skip(position);
}

178. Now to matrix names. What we do when we want to skip to a certain matrix name is to first go to the beginning of the file and search for the matrix with the correct name. If we succeed, then the file will be after the header, so that the input operator cannot be used. The solution is to rewind the file again and skip to the correct matrix.
⟨ifstream methods 157⟩ ≡
void skipto(const char *matrix_name);

179. void math::ifstream::skipto(const char *matrix_name)
{
    int position ← 0;
    seekg(0);
    ⟨Search for correct matrix name 180⟩
    skipto(position);
}

180. ⟨Search for correct matrix name 180⟩ ≡
math::fheader header;
header.name ← "";
while (strcmp(header.name.c_str(), matrix_name)) {
    if (position) skip_data(header);    /* Skip data of previous matrix */
    parse_header(header);
    if (rdstate() ≠ goodbit)
        throw math::error::generic("No matrix with supplied name in the file.");
    position ++;
}

This code is used in section 179.
181. **File streams: output.** The input part is complete, so let us go on the next part. Again, the biggest job is already done by the standard library `ofstream` class, we just need to specialize it a little. The problem now is that we must give matrices a name before writing them to a file. We will do that by keeping a private string for the matrix name in the class and providing a means to modify it.

```cpp
class ofstream : public std::ofstream {
    string matrix_name;
    public:
        ofstream(string name, int mode ← ios::out | ios::binary, int prot ← 664) {
            std::ofstream::open(name, mode, prot);
        }
}
```

182. We overload some constructors for compatibility with the standard `ofstream` library.

```cpp
math::ofstream &operator<<(const char *name) {
    matrix_name ← name;
    return *this;
}
```

183. Now to the opening of files. A matrix file is binary, so we change the default `ofstream` mode on both the constructor and the `open` method accordingly.

```cpp
void open(const char *name, int mode ← ios::in | ios::binary, int prot ← 664) {
    std::ofstream::open(name, mode, prot);
}
```

184. Now to the writing algorithms: since we normally write matrices in sequence, providing a method like the `ifstream`'s `skipto` is unhandy and cumbersome to use. The solution is to “write the matrix name” to the stream before writing the matrix itself. This operation will assign the matrix name field of the header to the specified name.

```cpp
math::ofstream &operator<<(const char *name) {
    matrix_name ← name;
    return *this;
}
```

185. Like with the input stream case, we define a method to output `longs`. The trick used is exactly the same.

```cpp
math::ofstream &operator<<(const long &);
188. \texttt{math::ofstream} \&\texttt{math::ofstream::operator\ll(const long \&source)}

\begin{verbatim}
    
    union {
        long Long;
        unsigned char Char[sizeof(long)];
    } tricky;
    tricky.Long ← source;
    for (int i ← 0; i ≠ sizeof(long); this→put(tricky.Char[i++]));
    return *this;

189. We are now ready to write matrices. We have here the same problem with complex types as we had in the input stream case, and we adopt the same specialization solution.

\begin{verbatim}
    \texttt{ofstream methods 182} \+=
    template<\texttt{matrix\_simple\_template}>
    ofstream \&operator\ll(const matrix\langle T, structure, storage\rangle \&source)

190. Waiting for export...

\begin{verbatim}
    \texttt{ofstream methods 182} \+=
    {
        long iflag ← 0;

        \{ Write matrix header 191 \}
        for (math::index j ← 1; j ≤ source.cols(); ++j)
            for (math::index i ← 1; i ≤ source.rows(); ++i) {
                double number;
                number ← double(source(i, j));

                \{ Write number to file, using the same tricky method as for longs 192 \}
            }
        return *this;
    }

191. \{ Write matrix header 191 \} ≡

    *this \ll long(0) \ll long(source.rows()) \ll long(source.cols()) \ll iflag \ll long(matrix\_name.size() + 1);
    for (unsigned int i ← 0; i ≤ matrix\_name.size(); this→put(matrix\_name\_c\_str(\[]i++\)));

This code is used in sections 190 and 194.

192. \{ Write number to file, using the same tricky method as for longs 192 \} ≡

\begin{verbatim}
    union {
        double Double;
        unsigned char Char[sizeof(double)];
    } tricky;
    tricky.Double ← number;
    for (int k ← 0; k ≠ sizeof(double); this→put(tricky.Char[k++]));

This code is used in sections 190 and 194.

193. Now to the complex specialization.

\begin{verbatim}
    \texttt{ofstream methods 182} \+=
    template<\texttt{matrix\_simple\_template}>
    ofstream \&operator\ll(const matrix\langle complex\langle T\rangle, structure, storage\rangle \&source)
\end{verbatim}
\end{verbatim}
194. Waiting for export...

```cpp
{ ofstream methods 182 } +≡
  
  long iflag ← 1;
  { Write matrix header 191 }
  for (int ipart ← 0; ipart ≤ 1; ++ipart)
    for (math::index j ← 1; j ≤ source.cols(); ++j)
      for (math::index i ← 1; i ≤ source.rows(); ++i) {
        double number;
        complex<double> aux;
        aux ← complex<double>(source(i, j));
        if (~ipart) number ← aux.real();
        else number ← aux.imag();
        { Write number to file, using the same tricky method as for longs 192 }
      }
  return *this;
```

195. With the end of the file streams classes we finished the main body of the library. With the definitions so far provided, you are able to do everything you want, including interfacing with MATLAB. Of course, what we have is too basic for complex algorithms, so our task now is to provide common functions, such as various types of matrix decompositions, solution of systems, and functions like determinant, inverse and so on. The point is that the main job is done, that is, the definition of the structures and methods necessary to develop algorithms. From now on there is really nothing really new, no impacting decisions to be made. The fun is not gone, but instead is replaced by the fun of using the power of the matrix class to write beautiful code.
196. The LU decomposition. We now begin to build the backbone of any linear algebra library: the matrix decomposition functions. We begin with the most simple, the LU decomposition. The goal is to decompose a matrix $A$ into a product of a lower-triangular matrix $L$ and an upper-triangular matrix $U$, that is, we want to have $A = LU$. The main purpose of this decomposition is to solve linear systems: if we want to solve $Ax = b$, we decompose matrix $A$ and then solve two systems, $Ly = b$ and $Ux = y$ to find the solution. The point is that solving triangular systems is very easy by back or forward substitution.

/* Empty, waiting for export */

197. ⟨lu.h 197⟩ ≡

#ifndef _MATH_LU_
#define _MATH_LU_ 1.0
#include <math/math.h>
#include <vector>
#include <complex>

namespace math {
    namespace lu {
        ⟨LU prototypes 199⟩
    }
}
#endif

198. We can understand the way the decomposition works by considering the first step in a $3 \times 3$ matrix. We build a matrix $M_1$ such that the first column of $M_1A$ is zero, except for the first element. The construction is

$$M_1 A = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{a_{21}}{a_{11}} & 1 & 0 \\ -\frac{a_{31}}{a_{11}} & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} - \frac{a_{21}a_{12}}{a_{11}} & a_{23} - \frac{a_{21}a_{13}}{a_{11}} \\ 0 & a_{32} - \frac{a_{31}a_{12}}{a_{11}} & a_{33} - \frac{a_{31}a_{13}}{a_{11}} \end{bmatrix}.$$  

Now we can repeat the procedure with the remaining blocks of the matrix so that

$$M_2 M_1 A = U,$$

where $U$ is upper-triangular. Since $M_i$ is lower-triangular, so it is its inverse, and defining $L = M_1^{-1} M_2^{-1}$ we finally have $A = LU$. A matrix of the form $M_i$ is called a Gauss transformation, and the $i$th column of $M_i$ is represented by $m_i = (0, 0, \ldots, 1, -\tau_{i+1}, \ldots, -\tau_n)$. The vector $\tau = (\tau_{i+1}, \ldots, \tau_n)$ is called the Gauss vector. In this way, a Gauss transformation $M_i$ can be represented as $M_i = (I - \tau^T e_i)$.

There is only one possible problem with the algorithm. Looking at the first step of the $3 \times 3$ decomposition example we can see that, if $a_{11}$ is small, we have numerical problems because the Gauss vector could have too large elements. This effect can propagate through the rest of the algorithm. The solution is to find the decomposition for a permuted version of $A$, such that at every step we have the largest possible denominator in the Gauss vector. The denominator is called the pivot, and the method of permutations is called pivoting. The LU decomposition has a workload of $O(n^3)$ flops (actually exactly $2n^3/3$) and, if with pivoting, $O(n^2)$ comparisons.
Our basic function will be one that will overwrite the input matrix with $U$ on the upper part and $L$ on the lower part (it can be easily shown – see Golub and Van Loan – that $L$ is a row permutation of the Gauss vectors). A vector $pivots$ will store the data about the permutations, such that at step $k$ we multiply $A$ by the identity matrix with rows $k$ and $pivots(k)$ swapped. The function returns $1$ if the number of permutations is even and $-1$ otherwise. Since we will overwrite the matrix at will, we require it to have no structure.

\[ \langle \text{LU prototypes } 199 \rangle \equiv \]

```cpp
template<class T, template<class> class storage>
int decompose(matrix<T, unstructured, storage> *A, vector<index> *pivots)
{
    int permutations ← 1; /* No permutations yet. */
    index n ← A.rows();
    if ($n \neq A.cols()$) throw error :: nonsquare();
    pivots.resize($n-1, 0$); /* Reserve space for permutation data. */
    for (index k ← 1; k ≠ n; ++k) {
        T pivot;
        ⟨Search for pivot and swap rows if necessary 200⟩
        if (pivot ≠ 0) ⟨Apply Gauss transformation 201⟩
    } // end for
    return permutations;
}
```

See also sections 202, 203, and 204.

This code is used in section 197.

200. ⟨Search for pivot and swap rows if necessary 200⟩ ≡

```cpp
pivot ← A.get(k, k);
(*pivots)[k-1] ← k; /* STL vector has base index 0. */
for (index i ← k + 1; i ≤ n; ++i)
    if (abs(A.get(i, k)) > abs(pivot)) {
        pivot ← A.get(i, k);
        (*pivots)[k-1] ← i;
    } // end if
if ((*pivots)[k-1] ≠ k) /* If need to swap. */
    {
        permutations *= -1;
        A.swaprows(k, (*pivots)[k-1]);
    } // end if
```

This code is used in section 199.

201. ⟨Apply Gauss transformation 201⟩ ≡

```cpp
for (index i ← k + 1; i ≤ n; ++i) {
    if (!finite(A.entry(i, k) ← A.get(i, k)/A.get(k, k))) throw error :: singular();
    for (index j ← k + 1; j ≤ n; ++j) A.entry(i, j) ← A.get(i, j) − A.get(i, k) * A.get(k, j);
}
```

This code is used in section 199.
202. **Solving a linear system.** As hinted before, the principal application of the LU decomposition is the solution of linear systems. The function we will define will solve a matrix linear system $AX = B$. The decomposition just defined gives us information to compute $P_{n-1} \ldots P_1 A = LU$, where $P_i$ are permutation matrices defined by the *pivots* vector. Having the LU decomposition, we solve the system $Ax = b$ by first solving $Ly = P_{n-1} \ldots P_1 b$ and then $Ux = y$. By doing this to all columns of $X$ and $B$ we are able to solve $AX = B$. We define now a function that, given a decomposition and a matrix $B$, overwrite in $B$ the solution to $AX = B$. We define this function separately because if you want to solve $A^kX = B$, you need to perform only one decomposition and call this function $k$ times.

```cpp
template<class T, template<class> class storage>
void finish(const matrix<T, unstructured, storage> &A, const vector<index>
&pivots, matrix<T, unstructured, storage> *B)
{
    index n ← A.rows();

    for (index i ← 1; i < n; ++i) /* B ← $P_{n-1} \ldots P_1 B$ */
        B.swaprows(i, pivots[i - 1]);

    for (index k ← B.cols(); k ≠ 0; --k) {
        for (index i ← 1; i ≤ n; ++i) /* Solve $Ly = B(:, k)$ */
            {
                T inner ← 0;

                for (index j ← 1; j ≠ i; ++j) inner += A(i, j) * B.get(j, k);

                B.entry(i, k) ← B.get(i, k) - inner;
            }

        for (index i ← n; i ≥ 1; --i) /* Solve $Ux = B(:, k)$ */
            {
                T inner ← 0;

                for (index j ← i + 1; j ≤ n; ++j) inner += A(i, j) * B.get(j, k);

                if (not finite(B.entry(i, k) ← (B.get(i, k) - inner) / A(i, i))) throw error::singular();
            }
    }
}
```

203. Finally, we provide an interface (that will destroy the original matrix, by the way). Note that we can compute the inverse of a matrix by calling this function with $B = I_n$.

```cpp
template<class T, template<class> class storage>
matrix<T, unstructured, storage> &solve(matrix<T, unstructured, storage>
*A, matrix<T, unstructured, storage> *B)
{
    vector<index> pivots;

decompose(A, &pivots);

finish(*A, pivots, &B);

return *B;
}
```
204. Interfacing with LAPACK. The routines just written enable you to compute the LU decomposition of any type of matrix. For certain types, however, we have extremely efficient decomposition functions already coded in the LAPACK package. It only makes sense to call these routines when possible, and the MATH library provides the ideal transparent interface: you call the exact same function, and if there is a LAPACK function to do the job, then the function will be called.

\[\text{LU prototypes} +\equiv\]

```c
#ifndef LAPACK
// (LU lapack interface)
#endif
```

The key for the transparent operation is specialization. The trick here is that we need to make sure that the matrix does not share the representation, otherwise the call to the LAPACK routine will modify all matrices sharing it.

\[\text{LU lapack interface} \equiv\]

```c
// Prepare for lapack LU
int decompose(matrix<fortran::double_precision, unstructured, dense> *A, vector<index> *pivots)
{
    // Prepare for lapack LU
    dgetrf(&m, &n, A->memory(), &m, (int *) pivots->begin(), &status);
    // Check for lapack LU errors
    // Compute number of permutations and store it in status
    return status;
}
```

See also section 209. This code is used in section 204.

205. The key for the transparent operation is specialization. The trick here is that we need to make sure that the matrix does not share the representation, otherwise the call to the LAPACK routine will modify all matrices sharing it.

\[\text{LU lapack interface} \equiv\]

```c
// Prepare for lapack LU
fortran::integer m ← fortran::integer(A->rows());
fortran::integer n ← fortran::integer(A->cols());
fortran::integer status ← 0;
pivots->resize(min(m, n), 0);
```

This code is used in sections 205 and 209.

206. (Prepare for lapack LU) \equiv

```c
if (status > 0) throw error::singular();
if (status < 0) throw error::generic();
```

This code is used in sections 205 and 209.

207. In order to be compatible with our own LU decomposition routine, we need to compute the number of permutations (1 means even number, -1 means odd).

\[\text{Compute number of permutations and store it in status} \equiv\]

```c
status ← 1;
for (index i ← 1; i < (index) min(m, n); i++)
    if ((pivots)[i - 1] != i) status *= -1;
```

This code is used in sections 205 and 209.
209. \[ \text{LU lapack interface 205} \] \text{+=} 

\begin{verbatim}
template()

int decompose(matrix(fortran::real, unstructured, dense) *A, vector(index) *pivots)
{
    \text{Prepare for lapack LU 206}
    sgetrf(&m, &n, A->memory(), &m, (int *) pivots->begin(), &status);
    \text{Check for lapack LU errors 207}
    \text{Compute number of permutations and store it in status 208}
    return status;
}
\end{verbatim}
210. The Cholesky decomposition. The LU decomposition works for all types of square matrices. If, however, a matrix is symmetric, then we will see that we can cut the work in half: first, instead of decomposing $A = LU$ we do $A = LDU$, where $D$ is diagonal and both $L$ and $U$ have unit diagonal (the matrix $L$ is already unit diagonal, and we can make the matrix $U$ the same by scaling it with a diagonal matrix $D$). Now, if $A$ is symmetric, then we have $L^{-1}AL^{-T} = DUL^{-T}$ is also symmetric, but this can only be true if $U = L^T$. Hence, if $A$ is symmetric we can decompose it such that $A = LL^T$, so we need to find only one matrix. If, in addition, $A$ is positive definite, then there is no need for pivoting, and the resulting decomposition is called the “Cholesky decomposition.”

211. Another way to see that we can decompose $A = LL^T$ if $A$ is symmetric is by establishing the equality

$$
\begin{bmatrix}
  a_{11} & \alpha^T \\
  \alpha & B
\end{bmatrix} =
\begin{bmatrix}
  \sqrt{a_{11}} & 0 \\
  \alpha/\sqrt{a_{11}} & I
\end{bmatrix}
\begin{bmatrix}
  1 & 0 \\
  0 & B - \alpha\alpha^T/a_{11}
\end{bmatrix}
\begin{bmatrix}
  \sqrt{a_{11}} & \alpha^T/\sqrt{a_{11}} \\
  0 & I
\end{bmatrix},
$$

which already hints the algorithm. One characteristic of the decomposition is that it is stable even without pivoting, and that during the entire algorithm all diagonal elements remain positive if the matrix itself is positive definite. The function we will define will overwrite the upper triangular part of $A$ with $L^T$. We do that because now we know that a symmetric matrix structure stores only the upper triangle elements.

212. Another way to see that we can decompose $A = LL^T$ if $A$ is symmetric is by establishing the equality

$$
\begin{bmatrix}
  a_{11} & \alpha^T \\
  \alpha & B
\end{bmatrix} =
\begin{bmatrix}
  \sqrt{a_{11}} & 0 \\
  \alpha/\sqrt{a_{11}} & I
\end{bmatrix}
\begin{bmatrix}
  1 & 0 \\
  0 & B - \alpha\alpha^T/a_{11}
\end{bmatrix}
\begin{bmatrix}
  \sqrt{a_{11}} & \alpha^T/\sqrt{a_{11}} \\
  0 & I
\end{bmatrix},
$$

which already hints the algorithm. One characteristic of the decomposition is that it is stable even without pivoting, and that during the entire algorithm all diagonal elements remain positive if the matrix itself is positive definite. The function we will define will overwrite the upper triangular part of $A$ with $L^T$. We do that because now we know that a symmetric matrix structure stores only the upper triangle elements.

213. Another way to see that we can decompose $A = LL^T$ if $A$ is symmetric is by establishing the equality

$$
\begin{bmatrix}
  a_{11} & \alpha^T \\
  \alpha & B
\end{bmatrix} =
\begin{bmatrix}
  \sqrt{a_{11}} & 0 \\
  \alpha/\sqrt{a_{11}} & I
\end{bmatrix}
\begin{bmatrix}
  1 & 0 \\
  0 & B - \alpha\alpha^T/a_{11}
\end{bmatrix}
\begin{bmatrix}
  \sqrt{a_{11}} & \alpha^T/\sqrt{a_{11}} \\
  0 & I
\end{bmatrix},
$$

which already hints the algorithm. One characteristic of the decomposition is that it is stable even without pivoting, and that during the entire algorithm all diagonal elements remain positive if the matrix itself is positive definite. The function we will define will overwrite the upper triangular part of $A$ with $L^T$. We do that because now we know that a symmetric matrix structure stores only the upper triangle elements.
214. \{ Apply Cholesky transformation 214 \} ≡

\[
\begin{array}{l}
\text{for (index } i \leftarrow k + 1; \ i \leq n; \ + + i) \\
\text{ for (index } j \leftarrow i; \ j \leq n; \ + + j) \ A\text{-entry}(i, j) \leftarrow A\text{-get}(i, j) - A\text{-get}(k, i) \ast A\text{-get}(k, j);
\end{array}
\]

This code is used in section 212.

215. **Solving a linear system.** The Cholesky decomposition uses half the number of flops as the LU, and in addition there is no pivoting overhead. It is only advisable to use it to solve linear equations with positive definite matrices. We solve a system \( AX = B \) vector by vector of \( X \) by first solving \( Ly = b \) and then \( L^T x = y \). As with the LU decomposition we define a function that solves the system given a previously computed decomposition. The solution \( X \) is overwritten in matrix \( B \).

\[
\langle \text{Cholesky prototypes 212} \rangle + \equiv
\]

\[
\begin{array}{l}
\text{template\langle class } T, \text{template\langle class } \text{ class structure}_A, \text{template\langle class } \text{ class structure}_B, \text{template\langle class } \text{ class storage} \rangle \\
\text{ void finish\langle const matrix\langle T, \text{structure}_A, \text{storage} \rangle & A, \text{matrix\langle T, \text{structure}_B, \text{storage} \rangle *B \rangle}
\end{array}
\]

\[
\{ \\
\text{ index } n \leftarrow A\text{.rows}(); \\
\text{ for (index } k \leftarrow B\text{-cols}(); \ k \neq 0; \ --k) \{ \\
\text{ for (index } i \leftarrow 1; \ i \leq n; \ + + i) \ /* \text{ Solve } Ly = B(:, k). */
\end{array}
\]

\[
\begin{array}{l}
\text{ T inner } \leftarrow 0; \\
\text{ for (index } j \leftarrow 1; \ j < i; \ + + j) \ /* \text{ Here we have to remember that we overwrote only the upper triangular part of } A, \text{ so that now we have to get the element } A(j, i) \text{ instead of } A(i, j). */
\end{array}
\]

\[
\begin{array}{l}
\text{ inner +} = A(j, i) \ast B\text{-get}(j, k); \\
\text{ if (} \langle \text{finite}(B\text{-entry}(i, k) \leftarrow (B\text{-get}(i, k) - inner)/A(i, i)) \rangle \text{ throw error::singular();}
\end{array}
\]

\[
\begin{array}{l}
\text{ for (index } i \leftarrow n; \ i \geq 1; \ --i) \ /* \text{ Solve } L^T x = y. */
\end{array}
\]

\[
\begin{array}{l}
\text{ T inner } \leftarrow 0; \\
\text{ for (index } j \leftarrow i + 1; \ j \leq n; \ + + j) \text{ inner +} = A(i, j) \ast B\text{-get}(j, k); \\
\text{ if (} \langle \text{finite}(B\text{-entry}(i, k) \leftarrow (B\text{-get}(i, k) - inner)/A(i, i)) \rangle \text{ throw error::singular();}
\end{array}
\]

\[
\begin{array}{l}
\} \\
\text{ for (index } j \leftarrow i + 1; \ j \leq n; \ + + j) \text{ inner +} = A(i, j) \ast B\text{-get}(j, k); \\
\text{ if (} \langle \text{finite}(B\text{-entry}(i, k) \leftarrow (B\text{-get}(i, k) - inner)/A(i, i)) \rangle \text{ throw error::singular();}
\end{array}
\]

\[
\begin{array}{l}
\} \\
\text{ for (index } j \leftarrow i + 1; \ j \leq n; \ + + j) \text{ inner +} = A(i, j) \ast B\text{-get}(j, k); \\
\text{ if (} \langle \text{finite}(B\text{-entry}(i, k) \leftarrow (B\text{-get}(i, k) - inner)/A(i, i)) \rangle \text{ throw error::singular();}
\end{array}
\]

216. Finally, we provide an interface for the linear solver. The big warning is that the \( A \) matrix will be overwritten too, not only \( B \)!!

\[
\langle \text{Cholesky prototypes 212} \rangle + \equiv
\]

\[
\begin{array}{l}
\text{template\langle class } T, \text{template\langle class } \text{ class structure}_A, \text{template\langle class } \text{ class structure}_B, \text{template\langle class } \text{ class storage} \rangle \\
\text{ matrix\langle T, \text{structure}_B, \text{storage} \rangle & solve\langle \text{matrix\langle T, \text{structure}_A, \text{storage} \rangle *A, \text{matrix\langle T, \text{structure}_B, \text{storage} \rangle *B \rangle}
\end{array}
\]

\[
\{ \\
\text{ decompose}(A); \\
\text{ finish}(A, B); \\
\text{ return } *B;
\}
\]
217. **Interfacing with LAPACK.** Again, the routines we just defined can compute the Cholesky decomposition of any matrix. If you have LAPACK installed, however, you can take advantage of years and years of laborious code optimization – why not use it?

\[\text{Cholesky prototypes} \quad \equiv \quad \#	ext{include} \quad \text{<math/private/fortran.h>}\]
\[\#\text{ifdef} \quad \text{HAVE_LIBLAPACK}\]
\[\text{(Cholesky lapack interface} \quad 218\text{)}\]
\[\#	ext{endif}\]

218. We begin with the double precision symmetric matrix case. The others are very similar, we just have to replace the function call.

\[\text{Cholesky lapack interface} \quad 218\text{)} \quad \equiv \quad \text{template}\{\}
\[\text{void} \quad \text{decompose} \quad (\text{matrix} \quad \text{<fortran::double\_precision, symmetric, dense}> \quad *A)\]
\[\{\]
\[\text{\{Prepare for lapack Cholesky} \quad 219\text{)}\]
\[\text{dpptrf} \quad (\text{&mode}, \text{&n}, \text{A\_storg()\_memory()}, \&\text{status});\]
\[\text{(Check lapack Cholesky errors} \quad 220\text{)}\]
\[\}\]

See also sections 221, 222, and 223.
This code is used in section 217.

219. \[\text{\{Prepare for lapack Cholesky} \quad 219\text{)} \quad \equiv \quad \text{fortran::integer} \quad n \quad \leftarrow \quad \text{A\_rows()};\]
\[\text{fortran::integer} \quad m \quad \leftarrow \quad \text{A\_cols()};\]
\[\text{fortran::integer} \quad \text{status} \quad \leftarrow \quad 0;\]
\[\text{fortran::character} \quad \text{mode} \quad \leftarrow \quad \text{'U'};\]
\[\text{if} \quad (n \neq n) \quad \text{throw} \quad \text{error}::\text{nonsquare}();\]
This code is used in sections 218, 221, 222, and 223.

220. \[\text{\{Check lapack Cholesky errors} \quad 220\text{)} \quad \equiv \quad \text{if} \quad (\text{status} \quad > \quad 0) \quad \text{throw} \quad \text{error}::\text{nonpositivedef}();\]
\[\text{if} \quad (\text{status} \quad < \quad 0) \quad \text{throw} \quad \text{error}::\text{generic}();\]
This code is used in sections 218, 221, 222, and 223.

221. As said before, the rest of the routines is essentially the same, except that for unstructured matrices we need to pass also the number of columns to the LAPACK routines.

\[\text{Cholesky lapack interface} \quad 218\text{)} \quad \equiv \quad \text{template}\{\}
\[\text{void} \quad \text{decompose} \quad (\text{matrix} \quad \text{<fortran::real, symmetric, dense}> \quad *A)\]
\[\{\]
\[\text{\{Prepare for lapack Cholesky} \quad 219\text{)}\]
\[\text{sptrf} \quad (\text{&mode}, \text{&n}, \text{A\_storg()\_memory()}, \&\text{status});\]
\[\text{(Check lapack Cholesky errors} \quad 220\text{)}\]
\[\} \]
222. We begin with the symmetric matrix cases.

\begin{verbatim}
  template<typename A>
  void decompose(matrix<fortran::double_precision, unstructured, dense> *A)
  {
    // Prepare for lapack Cholesky
    dpotrf(&mode, &n, A->memory(), &m, &status);
    // Check lapack Cholesky errors
  }
\end{verbatim}

223. \begin{verbatim}
  template<typename A>
  void decompose(matrix<fortran::real, unstructured, dense> *A)
  {
    // Prepare for lapack Cholesky
    spotrf(&mode, &n, A->memory(), &m, &status);
    // Check lapack Cholesky errors
  }
\end{verbatim}
224. The QR decomposition. The last two decompositions worked with square matrices, decomposing them into matrices with some structure that made easy the task of solving linear systems. The QR decomposition, on the other hand, produces $A = QR$, where $R$ is upper triangular and $Q$ is orthogonal, that is, $Q'Q = QQ' = I$. This decomposition provides a straightforward way to solve the problem $\min_x \|Ax - b\|_2$, and is also the basis for one of the most widely used algorithms for computing the SVD decomposition and the eigenvalues of a matrix.

```c
/* Empty, waiting for export. */
```

225. ⟨qr.h 225⟩ ≡
```c
#ifndef __MATH_QR__
define __MATH_QR__ 1.0
#include <math/math.h>
#include <math/algebra.h>
namespace math {
 namespace qr {
 ⟨QR prototypes 226⟩
 } }
#endif
```

226. The QR decomposition algorithm we will implement works by finding orthogonal matrices $H_i$ such that $H_n \cdots H_1 A = R$, where $R$ is upper triangular. The $H_i$ matrices are called Householder matrices, and what they do is to selectively zero out elements of a column of $A$. We’ll see how to compute these matrices later on. A straightforward algorithm would then be one that repeatedly computed and applied Householder transformations to the original matrix. This is exactly the algorithm we implement here until necessity arrives for pivoting.

```c
⟨QR prototypes 226⟩ ≡
```
227. A Householder reflection can in fact be represented by a single vector $v$ and a scalar $\beta$. In fact, the definition of a Householder matrix is any $n - \times - n$ matrix $P$ of the form

$$I - \frac{2}{v^T v} v v^T.$$ 

Synonyms are Householder reflection, Householder matrix. The vector $v$ is called a Householder vector. Householder matrices are easily verified to be symmetric and orthogonal. If a vector $x$ is multiplied by a Householder matrix, then it is reflected in the hyperplane defined by span $\{v\}^\perp$. In particular, suppose we have a vector $x$ and want to zero out all but the first component, that is, we want $P x$ to be a multiple of $e_1$. After some algebra we can see that

$$v = x \pm P \beta x^0_2 e_1$$

gives the desired transformation. The following piece of code will compute a Householder vector such that $v(1) = 1$, $\beta = 2/v^T v$, and $P x = P \beta x^0_2 e_1$. The normalization of the first entry of the Householder vector is desirable because we can store it with one less component (which enables us to store them directly in the $A$ matrix together with $R$). Also, we don’t use the above formula because of numerical problems. We use instead

$$v_1 = x_1 - P \beta x^0_2 = \frac{- (x_2^2 + \cdots + x_n^2)}{x_1 + P \beta x^0_2}$$

which is numerically more stable when $x_1 > 0$.

```plaintext
(Compute Householder vector for column j 227) \equiv
v.resize(A.rows() - j + 1, 1);
v.entry(1) \leftarrow T(0.0);
v.subm(2,v.rows()) \leftarrow A.subm(j + 1, A.rows(), j, j);
T sigma \leftarrow dot(v,v);
v.entry(1) \leftarrow T(1.0);
if (sigma \equiv 0) beta \leftarrow 0;
else {
    T x \leftarrow A.get(j,j);
    T mu \leftarrow sqrt(sigma + x * x);
    v.entry(1) \leftarrow (x \leq 0 ? x - mu : -sigma / (x + mu));
    T v1 \leftarrow v(1);
    beta \leftarrow 2 * v1 * v1 / (sigma + v1 * v1);
    v /\ = v1;
}
```

This code is used in section 226.
228. In order to apply the Householder transformation we need to be careful. We actually don’t need any matrix-matrix products if we realize that

\[ PA = (I - \beta vv^T) A = A - \beta vv^T A, \]

and \( vv^T Av^T = v(A^T v)^T \), which consists only of matrix-vector and vector-vector products. Thus, the Householder update requires only a vector-matrix multiplication and an outer product update. In our case, we can also make use of some facts: we know that \( v(1) = 1 \) and that a block of the \( A \) matrix is already zeroed out.

229. Solving linear equations. We now are in position to solve two important problems. The first is the so-called least-squares problem, where we find \( x \) that solves \( \min \| Ax - b \|_2 \), where \( A \in \mathbb{R}^{m \times n} \) and \( m > n \). In this case the system is overdetermined, and an exact solution to \( Ax = b \) may not exist.

The second problem occurs when \( A \in \mathbb{R}^{m \times n} \) and \( n > m \). The system \( Ax = b \) either has an infinite number of solutions or none. If it does have one, we compute the minimum norm solution.

230. Least squares. The least squares problem is solved via a direct matrix decomposition. The solution matrix is guaranteed to be smaller than \( B \) and has the same number of columns, so we overwrite \( B \) with the solution and resize it accordingly before returning.
231. To solve \( Qy = b \) we have to remember that \( Q \) is orthogonal, so all we have to do is to compute \( y = Q^Tb \).

\[
\langle \text{Solve } Qy = b \text{ 231} \rangle \equiv \\
\text{matrix}(T, \text{unstructured, storage}) v(m, 1);
\]

\[
\begin{align*}
\text{v.entry}(1) & \leftarrow T(1); \\
\text{for } (\text{index } j \leftarrow 1; j \leq n; ++j) & \{ \\
\text{v.resize}(m - j + 1, 1); \\
\text{v.subm}(2, v.rows()) & \leftarrow A\text{-subm}(j + 1, m, j); \\
\text{submatrix}(\text{matrix}(T, \text{unstructured, dense})) & \text{ } B\text{block}(B, j, m, 1, l); \\
w & \leftarrow \text{atxml}(B\text{block}, v, \&w); \\
\text{outerp_update}(\&\text{Bblock}, -2/\text{dot}(v, v), v, w); \\
\}
\end{align*}
\]

This code is used in section 230.

232. The last part is just back substitution with an upper triangular matrix.

\[
\langle \text{Solve } Rx = y \text{ 232} \rangle \equiv \\
\begin{align*}
\text{for } (\text{index } k \leftarrow 1; k \leq l; ++k) & \{ \\
\text{for } (\text{index } i \leftarrow n; i \geq 1; --i) & \{ \\
T & \text{inner } \leftarrow 0; \\
\text{for } (\text{index } j \leftarrow i + 1; j \leq n; ++j) & \text{ inner } += A\text{-get}(i, j) \ast B\text{-get}(j, k); \\
\text{if } (\neg \text{finite}(B\text{-entry}(i, k) \leftarrow (B\text{-get}(i, k) \ast \text{inner})/A\text{-get}(i, i))) & \text{ throw error::rankdeficient()}; \\
\}
\}
\end{align*}
\]

This code is used in section 230.

233. Minimum norm. The minimum norm problem is solved through a decomposition of the transposed matrix. If \( Ax = b \), then we take the decomposition of \( A^T \) and solve the system \((QR)^T x = b\). The matrix dimensions now will be \( Q \in \mathbb{R}^{n \times n} \) and \( R \in \mathbb{R}^{n \times m} \). The matrix we’ll work with is \( At \), which is in \( \mathbb{R}^{n \times m} \).

\[
\langle \text{Solve minimum norm problem 233} \rangle \equiv \\
\text{matrix}(T, \text{unstructured, storage}) At \leftarrow \text{transpose}(\ast A); \\
(\text{void}) \text{decompose}(\&At); \\
(\text{Solve } R^T y = b \text{ 234}); \\
(\text{Solve } Q^T x = y \text{ 235});
\]

This code is used in section 229.

234. Our system consists of \( R^T Q^T x = b \). We first solve \( R^T y = b \). This equation actually translates to \([R_1 \ 0] [y_1 \ y_2] = b\), so we have no means of determining \( y_2 \). This poses no problem since we can determine \( y_1 \) and that’s all we need. We’ll overwrite \( y_1 \) on \( B \).

\[
\langle \text{Solve } R^T y = b \text{ 234} \rangle \equiv \\
\begin{align*}
\text{for } (\text{index } k \leftarrow 1; k \leq l; ++k) & \{ \\
\text{for } (\text{index } i \leftarrow 1; i \leq m; ++i) & \{ \\
T & \text{inner } \leftarrow 0; \\
\text{for } (\text{index } j \leftarrow 1; j < i; ++j) & \text{ inner } += At(j, i) \ast B\text{-get}(j, k); \\
\text{if } (\neg \text{finite}(B\text{-entry}(i, k) \leftarrow (B\text{-get}(i, k) \ast \text{inner})/At.(i, i))) & \text{ throw error::rankdeficient()}; \\
\}
\}
\end{align*}
\]

This code is used in section 233.
235. Now we use the fact that $Q$ is orthogonal, so that $Q^T x = y$ translates to $x = Qy$. The only problem is that we don’t have $y$, just $y_1$, so that we can’t use the outerp_update function here because the matrix dimensions wouldn’t match.

\[
\langle \text{Solve } Q^T x = y \rangle \equiv \\
B\text{-resize}(n, l); \\
B\text{-subm}(m + 1, n, 1, l) \leftarrow \text{matrix}(T, \text{unstructured}, \text{storage})(n - m, l); \\
\text{matrix}(T, \text{unstructured}, \text{storage}) \ v(n, 1); \\
v.\text{entry}(1) \leftarrow T(1); \\
\text{for } (\text{index } k \leftarrow m; \ k \geq 1; \ --k) \ \{ \\
\ \text{index } nrows \leftarrow n - k + 1; \\
\ \ v.\text{resize}(nrows, 1); \\
\ \ v.\text{subm}(2, nrows) \leftarrow A.\text{subm}(k + 1, n, k, k); \\
\ \ \text{submatrix(\text{matrix}(T, \text{unstructured}, \text{dense})) } B\text{block}(B, k, n, 1, l); \\
\ \ w \leftarrow \text{atxmul}(B\text{block}, v, \&w); \\
\ \ \text{outerp_update}(\&B\text{block}, -2/\text{dot}(v, v), v, w); \\
\ \}\]

This code is used in section 233.
236. **Matrix creation functions.** We have now the power of the definitions, but using only the basics is unhandy. If we want to create an identity matrix, for example, we should not expect to have to set the diagonal elements to 1 manually. In this part we define functions to create matrices of common use.

237. **Eye.** Following MATLAB syntax, the function that creates the identity matrix is `eye`. How nice.

```cpp
/* Empty, waiting for export */
```

238. (eye.h 238) ≡

```cpp
#include <algorithm> /* For min. */
#include <math/math.h>
namespace math {
    template<matrix_simple_template> matrix<T,structure,storage> eye(const math::index rows, const math::index cols)
    {
        matrix<T,structure,storage> dest(rows,cols);
        for (math::index i ← 1; i ≤ min(rows,cols); ++i) dest.entry(i,i) ← T(1);
        return dest;
    }
    #endif
```
239. Ones. Following MATLAB syntax, the function that creates the matrix with ones all around is `ones`. How nice, again. Since we have a matrix constructor that sets values, the function itself is nothing more than a matrix creation. For people used with MATLAB (or Scilab, or Octave, and so on) the use of `ones` in a program may seem more intuitive.

`/* Empty, waiting for export */`

240. `<ones.h>`

```cpp
 ifndef __MATH_ONES__
 define __MATH_ONES__ 1.0
 include <math/math.h>
 namespace math {
 template<\template{matrix_simple_template}>
 matrix<\template{T,structure,storage}> ones(const math::index rows, const math::index cols)
 {
     return matrix<\template{T,structure,storage}>(rows, cols, T(1));
 }
 }
 endif
```
241. Matrix functions.

242. Determinant. The determinant of a matrix $A \in \mathbb{R}^{n \times n}$ is given by

$$\det(A) = \sum_{i=1}^{n} (-1)^{i+1} a_{1j} \det(A_{1j}),$$

where $A_{1j}$ is an $(n-1)$-by-$(n-1)$ matrix obtained by deleting the first row and $j$th column of $A$. Computing the determinant this way would require $O(n!)$ operations, which is unacceptable. Fortunately, we have $\det(AB) = \det(A) \det(B)$, and for a upper or lower triangular matrix $\det(A) = \prod a_{ii}$. With this in mind we are able to compute the determinant in $2n^3/3$ operations via the LU decomposition: we have $\det(A) = \det(LU) = \det(L) \det(U) = \det(U) = \prod u_{ii}$. The only trick is that we have a permuted version, so we have to take into account the number of permutations. The LU decomposition function returns the necessary information.

/* Empty, waiting for export */

243. ⟨det.h 243⟩ ≡

```c
#ifndef __MATH_DET__
#define __MATH_DET__ 1.00
#include <math/math.h>
#include <math/lu.h>
namespace math {
    template(matrix_simple_template)
    T det(const matrix(T,structure,storage) &A)
    {
        matrix(T,unstructured,storage) aux ← A;
        vector(math::index) pivots;
        T determinant ← 0;
        try {
            determinant ← math::lu::decompose(&aux,&pivots);
            math::index i ← 0;
            while (++i ≤ aux.rows()) determinant *= aux(i,i)
        } catch(math::error::singular e) {}
        return determinant;
    }
}#endif
```
244. Functions. Functions are things that take a matrix as argument and return a matrix. We will define them before functionals (which take vectors and return scalars) because, for optimization purposes, functionals are more useful but they need functions in order to be of use. Here we define the basic interface in a base class from which specific functionals will be derived.

\[\begin{array}{l}
/* \text{Not until} \text{export} */
\end{array}\]

245. \langle\text{functionbase.h} \ 245\rangle \equiv

```
#ifndef __MATH_FUNCTION__
#define __MATH_FUNCTION__ 1.00
#include <math/math.h>
#include <math/symmetric.h>
#include <math/sparse.h>
namespace math {
    namespace function {
        template<class T>
        class base {
        public:
            \{ Function base class methods 246 \}
        };
    }
}
#endif
```

246. The first thing we do is to declare a virtual destructor, so that we can derive classes from the base class.

\langle\text{Function base class methods} \ 246\rangle \equiv

```
virtual ~base(void) {};
```

See also sections 247, 248, and 249.

This code is used in section 245.

247. Before defining the interface we need to consider some points about functions (this rant will be repeated when defining functionals): first, a function should be able to get any kind of matrix as the point, that is, the \( x \) in \( f(x) \) could be sparse, dense and so on. Also, we should be able to create a list containing various types of functions. The impossibility of this ideal situation is summarized by Stroustrup: “A member template cannot be virtual.” This is a design decision, and a wrong one in my point of view. We have then only two possibilities: either we make the matrix class derived, so we can pass pointers, or we fix the type of the parameter. The first option results in performance degradation, so we’re stuck with the second. Fortunately most functions take vectors as arguments, so that matrix structures are not a big deal. We will assume that dense vectors are the best compromise. Also, we fix the return value to a dense and unstructured matrix because we consider that in most cases we’ll be returning vectors. Storage for the result must be provided by the user.

\langle\text{Function base class methods} \ 246\rangle +≡

```
matrix\langle T, \text{unstructured}, \text{dense} \rangle \& \text{operator}(\text{const matrix}\langle T, \text{unstructured}, \text{dense} \rangle \& x, \text{matrix}\langle T, \text{unstructured}, \text{dense} \rangle \& dest) 
\{ \text{return eval}(x, dest); \}
```

```
virtual matrix\langle T, \text{unstructured}, \text{dense} \rangle \& \text{eval}(\text{const matrix}\langle T, \text{unstructured}, \text{dense} \rangle \& x, \text{matrix}\langle T, \text{unstructured}, \text{dense} \rangle \& dest) \leftarrow 0;
```
248. If possible, a function should compute the Jacobian. If \( f : \mathbb{R}^n \to \mathbb{R}^m \), then the Jacobian is a matrix \( J \in \mathbb{R}^{m \times n} \) defined by \( J_{ij} = \frac{df_i}{dx_j} \). Here we face another interface decision: functions are supposed to be used to compute lots of points, that is, when you create a function, usually you want to evaluate it in a set of points, not just only one. If a evaluation method created storage for the result each time it was called, then we would be facing a serious performance threat. For this reason we require the user to pass the result variable as an argument. This requirement, in turn, makes the type of the result matrix fixed for the same reasons discussed above. The method returns the result (that is, \( \text{dest} \)), which is a useful behavior in certain applications, as for example in function compositions like \( f(g(x)) \). The size of the \( \text{dest} \) matrix at input should be checked by the method. This will be normally performed via the \text{matrix}::\text{resize} method. Since this method checks for same dimensions, the performance does not suffer too much.

249. The same thing is valid for the Hessian, except here we require the Hessian to be symmetric (which it is by definition). The same remarks about \( \text{dest} \) size made above are valid here. We provide two methods, one for \text{sparse} matrices (for large optimization problems it may be crucial to get a sparse version).

250. The gaxpy function. Our first functional is arguably the most simple. We define a function that computes the scalar gaxpy operation, that is, \( f(x) = Ax + b \). This is also one of the most useful functions for optimization – think of \( Ax \prec b \).

/* Empty, waiting for export */
252. To compute a gaxpy operation we need two parameters, the $A$ and $b$ matrices. We assume $b$ will be a vector most times.

To compute a gaxpy operation we need two parameters, the $A$ and $b$ matrices. We assume $b$ will be a vector most times.

$$\langle \text{Gaxpy function internal variables 252} \rangle \equiv$$

matrix $T$, structure, storage $ay$;

matrix $T$, unstructured, dense $bee$;

This code is used in section 251.

253. We provide a means to modify these values in two ways: by returning a reference to them and at the time of construction.

$$\langle \text{Gaxpy function class methods 253} \rangle \equiv$$

matrix $T$, structure, storage $&A(\text{void})$ { return $ay$; }

matrix $T$, unstructured, dense $&b(\text{void})$ { return $bee$; }

gaxpy(const matrix $T$, structure, storage) $&a$, const $T &B$)

: $ay(a)$, $bee(B)$ { }

See also sections 254, 255, and 256.

This code is used in section 251.

254. Now to the fun stuff. Evaluating is a simple task of calling some algebra functions.

$$\langle \text{Gaxpy function class methods 253} \rangle +=$$

virtual matrix $T$, unstructured, dense $&eval$(const matrix $T$, unstructured, dense)

&$x$, matrix $T$, unstructured, dense) *dest

{ return (gaxpy($ay, x, &$(dest-copyfrom($b$)))) ; }

255. The Jacobian is simply $A$.

$$\langle \text{Gaxpy function class methods 253} \rangle +=$$

virtual matrix $T$, unstructured, dense $&jacobian$(const matrix $T$, unstructured, dense)

&$x$, matrix $T$, unstructured, dense) *dest

{ return (*dest) $\leftarrow ay$; }

256. The hessian of an affine function is zero.

$$\langle \text{Gaxpy function class methods 253} \rangle +=$$

virtual matrix $T$, symmetric, dense $&hess$(const matrix $T$, unstructured, dense)

&$x$, matrix $T$, symmetric, dense) *dest, const index, const index

{ dest resize($x$.rows(), $x$.rows());

dest fillwith(0);

return *dest; }

virtual matrix $T$, symmetric, sparse $&hess$(const matrix $T$, unstructured, dense)

&$x$, matrix $T$, symmetric, sparse) *dest, const index, const index

{ *dest $\leftarrow$ matrix $T$, symmetric, sparse)($x$.rows(), $x$.rows());

return *dest; }


257. **Functionals.** Functionals are the MATH library representation of functionals. In the most simple situation, functionals take arguments and return the value of a function at some point. In other situations, such as optimization algorithms, we may want the gradient and/or the Hessian of the function at some point. Here we define the basic interface in a base class from which specific functionals will be derived.

```cpp
/* Not until export */
```

258. \[ \langle \text{functionalbase.h} \rangle \equiv \]

```cpp
#ifndef __MATH_FN__
#define __MATH_FN__ 1.00
#include <math/math.h>
#include <math/symmetric.h>
#include <math/sparse.h>
namespace math {
namespace functional {

template<class T>
class base {
public:
    // Functional base class methods 259

};
}
}
#endif
```

259. The first thing we do is to declare a virtual destructor, so that we can derive classes from the base class.

```cpp
virtual ~base(void) {} 
```

See also sections 260, 261, and 262.

This code is used in section 258.

260. Before defining the interface we need to consider some points about functionals: first, a functional should be able to get any kind of matrix as the point, that is, the \( x \) in \( f(x) \) could be sparse, dense and so on. Also, we should be able to create a list containing various types of functionals. The impossibility of this ideal situation is summarized by Stroustrup: “A member template cannot be virtual.” This is a design decision, and a wrong one in my point of view. We have then only two possibilities: either we make the matrix class derived, so we can pass pointers, or we fix the type of the parameter. The first option results in performance degradation, so we’re stuck with the second. Fortunately most functionals take vectors as arguments, so that matrix structures are not a big deal. We will assume that dense vectors are the best compromise.

```cpp
T operator()(const matrix<T, unstructured, dense> &x) { return eval(x); }
virtual T eval(const matrix<T, unstructured, dense> &x) ← 0;
```
261. If possible, a functional should compute the gradient at a point. Here we face another interface decision: functionals are supposed to be used to compute lots of points, that is, when you create a functional, usually you want to evaluate it in a set of points, not just only one. If a evaluation method created storage for the result each time it was called, then we would be facing a serious performance threat. For this reason we require the user to pass the result variable as an argument. This requirement, in turn, makes the type of the result matrix fixed for the same reasons discussed above. The method returns the result (that is, dest), which is a useful behavior in certain applications, as for example in function compositions like \( f(g(x)) \). The size of the dest matrix at input should be checked by the method. This will be normally performed via the matrix::init method. Since this method checks for same dimensions, the performance does not suffer too much.

262. The same thing is valid for the Hessian, except here we require the Hessian to be symmetric (which it is by definition). The same remarks about dest size made above are valid here. We provide two methods, one for sparse matrices (for large optimization problems it may be crucial to get a sparse version).

263. The gaxpy functional. Our first functional is arguably the most simple. We define a functional that computes the scalar gaxpy operation, that is, \( f(x) = a^T x + b \).

/∗ Empty, waiting for export ∗/

264. ⟨functional/gaxpy.h 264⟩ ≡

#include <math/algebra.h>
#include <math/functionalbase.h>
namespace math {
namespace functional {


virtual matrix⟨T, unstructured, dense⟩ &grad(const matrix⟨T, unstructured, dense⟩ &x, matrix⟨T, unstructured, dense⟩ &dest) ← 0;

virtual matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩ &x, matrix⟨T, symmetric, dense⟩ &dest) ← 0;

virtual matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩ &x, matrix⟨T, symmetric, sparse⟩ &dest) ← 0;

To compute a gaxpy operation we need two parameters, the \( a \) vector and the scalar \( b \).

matrix⟨T, structure, storage⟩ ay;
T bee;

This code is used in section 264.
266. We provide a means to modify these values in two ways: by returning a reference to them and at the time of construction.

\[
\text{Gaxpy functional class methods (266)} \equiv
\]

\[
\begin{align*}
\text{matrix}(\mathbf{T}, \text{structure, storage}) & \& a(\text{void}) \{ \text{return } ay; \} \\
\mathbf{T} & \& b(\text{void}) \{ \text{return } bee; \} \\
gaxpy(\text{const matrix}(\mathbf{T}, \text{structure, storage}) & A, \text{const } \mathbf{T} & B) \\
& : ay(A), bee(B) \{ \}
\end{align*}
\]

See also sections 267, 268, and 269.

This code is used in section 264.

267. Now to the fun stuff. Evaluating is a simple task of calling some algebra functions.

\[
\text{Gaxpy functional class methods (266)} + \equiv
\]

\[
\begin{align*}
\text{virtual } \mathbf{T} \text{ eval(const matrix}(\mathbf{T}, \text{unstructured, dense}) & x) \\
& \{ \text{return } \text{dot}(ay, x) + bee; \}
\end{align*}
\]

268. The gradient is simply \( a \).

\[
\text{Gaxpy functional class methods (266)} + \equiv
\]

\[
\begin{align*}
\text{virtual matrix}(\mathbf{T}, \text{unstructured, dense}) & \text{grad(const matrix}(\mathbf{T}, \text{unstructured, dense}) \\
& & \& x, \text{matrix}(\mathbf{T}, \text{unstructured, dense}) & \text{dest}) \\
& \{ \text{return } (\text{dest } \leftarrow ay); \}
\end{align*}
\]

269. The hessian of an affine function is zero.

\[
\text{Gaxpy functional class methods (266)} + \equiv
\]

\[
\begin{align*}
\text{virtual matrix}(\mathbf{T}, \text{symmetric, dense}) & \text{hess(const matrix}(\mathbf{T}, \text{unstructured, dense}) \\
& & \& x, \text{matrix}(\mathbf{T}, \text{symmetric, dense}) & \text{dest}) \\
& \{ \\
& \quad \text{dest.init}(x.\text{rows}(), x.\text{rows}()); \\
& \quad \text{dest.fillwith}(0); \\
& \quad \text{return } \text{dest}; \\
& \}
\end{align*}
\]

\[
\begin{align*}
\text{virtual matrix}(\mathbf{T}, \text{symmetric, sparse}) & \text{hess(const matrix}(\mathbf{T}, \text{unstructured, dense}) \\
& & \& x, \text{matrix}(\mathbf{T}, \text{symmetric, sparse}) & \text{dest}) \\
& \{ \\
& \quad dest \leftarrow \text{matrix}(\mathbf{T}, \text{symmetric, sparse})(x.\text{rows}(), x.\text{rows}()); \\
& \quad \text{return } \text{dest}; \\
& \}
\end{align*}
\]

270. The linear combination of functionals. Our next definition is a functional that is the weighed sum of other specified functionals. It will be useful, for example, in optimization methods (when summing barrier functions).

\[
/* \text{Empty, waiting for export} */
\]
271. \{ functional/linear.h \} \equiv

```c++
#ifndef __MATH_LINEAR_FUNCTIONAL__
#define __MATH_LINEAR_FUNCTIONAL__ 1.00
#include <algorithm>
#include <math/functionalbase.h>
namespace math {
    namespace functional {
        template<class T>
        class linear : public base<T> {
            // Linear combination of functionals internal variables
            typedef std::pair<T, base<T>*> element_type;
            vector<element_type> elements;
            typedef typename vector<element_type>::iterator iterator;
        };
    }
}
#endif
```

272. The functionals to be added are stored in a vector of pairs of weights and functionals.

\{ Linear combination of functionals internal variables \} \equiv

```c++
typedef std::pair<T, base<T>*> element_type;
vector<element_type> elements;
typedef typename vector<element_type>::iterator iterator;
```

See also sections 278 and 280.
This code is used in section 271.

273. Adding elements to the sum is a matter of passing weights and functional pointers to the `add` function. We also allow the user to specify initial values at the time of construction.

\{ Linear combination of functionals class methods \} \equiv

```c++
linear<void> {
    linear(base<T>* elment, T weight ← 1) { add(elment, weight); }
    linear(const vector<element_type>& elem) { elements ← elem; }
    void add(base<T>* elment, T weight ← 1)
    {
        if (weight ≡ 0) return;
        elements.push_back(element_type(weight, elment));
    }
    void add(const linear<T>* elment, T weight ← 1)
    {
        for (int i ← 0; i ≠ elment-size(); ++i) add(elment-get_term(i), weight * elment-get_weight(i));
    }
}
```

See also sections 274, 275, 276, 277, 279, and 281.
This code is used in section 271.

274. We can remove functionals from the end of the list.

\{ Linear combination of functionals class methods \} \equiv

```c++
void pop_back(void)
{
    elements.pop_back();
}
```
275. We will need to modify weights

```cpp
void set_weight(int i, T value) { elements[i].first ← value; }
T get_weight(int i) const { return elements[i].first; }
void set_term(int i, base<T> *value) { elements[i].second ← value; }
base<T> *get_term(int i) const { return elements[i].second; }
int size(void) const { return elements.size(); }
```

276. The pointers will be erased, but not the functionals they point to, when the linear is deleted.

```cpp
∼linear(void) { elements.erase(elements.begin(), elements.end()); }
```

277. Now to evaluation. Pretty simple, as you may expect. The default behavior is to return zero if there are no functionals in the list.

```cpp
T eval(const matrix<T, unstructured, dense> &x) {
    T result ← 0;
    for (iterator i ← elements.begin(); i ≠ elements.end(); ++i) result += i.first * i.second*eval(x);
    return result;
}
```

278. For the gradient, the default when no elements are present is also to return zero. We set up an internal variable aux so that we don’t need to create/resize vectors each time a new gradient is computed. Also, we test for unity weights in order to save computation.

```cpp
matrix(T, unstructured, dense) aux;
```

279. `Linear combination of functionals class methods 273} +≡

```cpp
matrix(T, unstructured, dense) &grad(const matrix<T, unstructured, dense> &x, matrix(T, unstructured, dense) &dest) {
    if (elements.size() == 0) {
        dest.resize(x.rows(), 1);
        dest.fillwith(0.0);
        return dest;
    }
    iterator i ← elements.begin();
    i.second−grad(x, dest);
    if (i.first ≠ 1) dest *= i.first;
    for (i++; i ≠ elements.end(); ++i) {
        i.second−grad(x, aux);
        if (i.first ≠ 1) aux *= i.first;
        dest += aux;
    }
    return dest;
}
```
280. We follow the same algorithm for the Hessian, again testing for unity weights in order to save computation.

\[
\text{matrix}(T, \text{symmetric, dense}) \ H_{\text{dense}};
\]

\[
\text{matrix}(T, \text{symmetric, sparse}) \ H_{\text{sparse}};
\]

281. \text{matrix}(T, \text{symmetric, dense}) \& \text{hess}(\text{const matrix}(T, \text{unstructured, dense})

\[
&\times, \text{matrix}(T, \text{symmetric, dense}) \& \text{dest}
\]

\{
if (\text{elements.size()} \equiv 0) \{
\text{dest.resize}(\text{x.rows()}, \text{x.rows()});
\text{dest.fillwith}(0.0);
return dest;
\}

\text{iterator} i \leftarrow \text{elements.begin}();
\text{i-second-hess}(\text{x, dest});
if (i-vrst \neq 1) \text{dest} \ast= \text{i-vrst};
for (i++; i \neq \text{elements.end}(); ++i) \{
\text{i-second-hess}(\text{x, H_{\text{dense}}});
if (i-vrst \neq 1) \text{H_{\text{dense}}} \ast= i-vrst;
\text{dest} += \text{H_{\text{dense}}};
\}
return dest;
\}

\text{matrix}(T, \text{symmetric, sparse}) \& \text{hess}(\text{const matrix}(T, \text{unstructured, dense})

\[
&\times, \text{matrix}(T, \text{symmetric, sparse}) \& \text{dest}
\]

\{
if (\text{elements.size()} \equiv 0) \{
\text{dest} \leftarrow \text{matrix}(T, \text{symmetric, sparse})(\text{x.rows()}, \text{x.rows()});
return dest;
\}

\text{iterator} i \leftarrow \text{elements.begin}();
\text{i-second-hess}(\text{x, dest});
if (i-vrst \neq 1) \text{dest} \ast= i-vrst;
for (i++; i \neq \text{elements.end}(); ++i) \{
\text{i-second-hess}(\text{x, H_{\text{sparse}}});
if (i-vrst \neq 1) \text{H_{\text{sparse}}} \ast= i-vrst;
\text{dest} += \text{H_{\text{sparse}}};
\}
return dest;
\}

282. The quadratic functional. The next functional we implement is the quadratic. It computes the value of \( f(x) = x^T P x + p^T x + \pi \). We will require \( P \) to be symmetric. If it is not, it is always possible to find a new \( P \) that results in the same functional. To the code:

\/*/ Empty, waiting for \text{export} */
We first provide storage for the functional parameters. As a design decision, the vector \( p \) will be dense and unstructured.

\[
\begin{align*}
\text{matrix} \langle T, \text{symmetric, storage} \rangle & \quad \text{Pee;} \\
\text{matrix} \langle T, \text{unstructured, dense} \rangle & \quad \text{pee;} \\
T & \quad \text{Pi;}
\end{align*}
\]

This code is used in section 283.

We provide means to change the parameters in the same way we did with the \text{gaxpy} functional.

\[
\begin{align*}
\text{matrix} \langle T, \text{symmetric, storage} \rangle & \quad \&P(\text{void}) \quad \{ \text{return} \text{Pee;} \} \\
\text{matrix} \langle T, \text{unstructured, dense} \rangle & \quad \&p(\text{void}) \quad \{ \text{return} \text{pee;} \} \\
T & \quad \&\text{pi}(\text{void}) \quad \{ \text{return} \text{Pi;} \} \\
\text{quadratic} \langle \text{const matrix} \langle \text{T, symmetric, storage} \rangle & \quad \&\text{newP}, \text{const matrix} \langle \text{T, unstructured, dense} \rangle \\
& \quad \&\text{newp}, \text{const T} & \text{newpi} \rangle \\
& \quad : \text{Pee} \langle \text{newP}, \text{pee} \langle \text{newp} \rangle, \text{Pi} \langle \text{newpi} \rangle \} \}
\end{align*}
\]

See also sections 286, 287, and 288. This code is used in section 283.

Evaluating: we use the \text{axmul} function for now, but it’s a good idea to change it in the future. Probably the best thing to do is to define a good \(*\) operator, but I don’t have one right now.

\[
\begin{align*}
\text{virtual T} & \quad \text{eval} \langle \text{const matrix} \langle \text{T, unstructured, dense} \rangle & \quad \&x \rangle \\
& \quad \{ \\
& \quad \text{T result} & \quad \text{dot} \langle \text{pee}, \text{x} \rangle + \text{Pi;} \\
& \quad \text{matrix} \langle \text{T, unstructured, dense} \rangle & \quad y \langle \text{Pee.rows}(), 1 \rangle; \\
& \quad y & \quad \text{axmul} \langle \text{Pee}, \text{x}, \&y \rangle; \\
& \quad \text{result} & \quad += \text{dot} \langle \text{x}, \text{y} \rangle; \\
& \quad \text{return} \text{ result;} \\
& \quad \}
\end{align*}
\]
287. The gradient is given by $2Px + p$.

\[
\langle \text{Quadratic functional methods 285} \rangle \equiv \\
\text{virtual matrix}(T, \text{unstructured, dense}) & \text{grad}(\text{const matrix}(T, \text{unstructured, dense}) \\
& x, \text{matrix}(T, \text{unstructured, dense}) & \text{dest}) \\
\{ \\
\text{return gaxpy}(Pee, x * 2, &(\text{dest.copyfrom}(pee))); \\
\}
\]

288. And the Hessian is simply $2P$.

\[
\langle \text{Quadratic functional methods 285} \rangle \equiv \\
\text{virtual matrix}(T, \text{symmetric, dense}) & \text{hess}(\text{const matrix}(T, \text{unstructured, dense}) \\
& x, \text{matrix}(T, \text{symmetric, dense}) & \text{dest}) \\
\{ \\
\text{dest} \leftarrow \text{Pee}; \\
\text{return} \text{dest} *= 2; \\
\}
\]

289. The norm-2 error. This functional computes, for a given function $f$ and a vector $y$, the value of

\[
\|y_i - f_i(x)\|^2.
\]

This is an extremely useful functional for optimization problems: suppose you trying to approximate some set of data $y$ to a function $f$ that depends on the parameters $x$ – then minimizing this functional is what you want to do. Note the confusing naming (of which we cannot escape). Normally the set of data is $(x_i, y_i)$, but in our case the $x_i$ is stored in the function $f$ itself – they are not the argument to $f$.

\[
/ * \text{ Empty, waiting for export. } */
\]

290. \langle \text{functional/norm2err.h 290} \rangle \equiv 

\#ifndef __MATH_NORM2ERR_FUNCTIONAL__
\#define __MATH_NORM2ERR_FUNCTIONAL__
\#include <math/algebra.h>
\#include <math/function.h>
\#include <math/functionalbase.h>
namespace math {
    namespace functional {
        template<class T, template<class> class storage>
        class norm2err:public base<T> {
            \langle \text{Norm-2 error functional internal variables 291} \rangle 
        public:
            \langle \text{Norm-2 error functional class methods 292} \rangle 
        }
    }
}
\#endif
The internal matrix variables are the vector \( y \), two auxiliary vectors for computing the value and gradient of the function, and an auxiliary matrix for computing the Hessian. The other internal variable is the approximating function.

\[
\langle \text{Norm-2 error functional internal variables} \rangle \equiv \begin{align*}
\text{matrix} \langle \mathbf{T}, \text{unstructured, dense} \rangle & \ Y, \ aux, \ fval; \\
\text{matrix} \langle \mathbf{T}, \text{symmetric, storage} \rangle & \ H; \\
\text{function} : \text{base} \langle \mathbf{T} \rangle * & \ F;
\end{align*}
\]

This code is used in section 290.

We can change all the values at any time or at construction.

\[
\langle \text{Norm-2 error functional class methods} \rangle \equiv \begin{align*}
\text{matrix} \langle \mathbf{T}, \text{unstructured, dense} \rangle \ & \& (\text{void}) \ {\text{return Y;}} \\
\text{function} : \text{base} \langle \mathbf{T} \rangle * & \& f(\text{void}) \ {\text{return F;}} \\
\text{norm2err} \langle \text{const matrix} \langle \mathbf{T}, \text{unstructured, dense} \rangle \ & \& \text{newy}, \text{function} : \text{base} \langle \mathbf{T} \rangle * & \& \text{newf} \leftarrow 0 \rangle \\
: \ Y(\text{newy}), \ F(\text{newf}) \ {\text{}} \\
\text{norm2err} \langle \text{function} : \text{base} \langle \mathbf{T} \rangle * & \& \text{newf} \leftarrow 0 \rangle \\
: \ F(\text{newf}) \ {\text{}}
\end{align*}
\]

See also sections 293, 294, 295, and 296.

This code is used in section 290.

Now to the eval function.

\[
\langle \text{Norm-2 error functional class methods} \rangle \equiv \begin{align*}
\text{virtual T} \ & \text{eval} \langle \text{const matrix} \langle \mathbf{T}, \text{unstructured, dense} \rangle \ & \& x \rangle \\
\{ \\
\text{fval} \leftarrow F-\text{eval}(x, \ & \& \text{fval}); \\
\text{fval} \leftarrow Y; \\
\text{return} \ \text{dot}(\text{fval}, \text{fval});
\}
\end{align*}
\]

The gradient is given by

\[
2Df(x)^T(f - y)
\]

\[
\langle \text{Norm-2 error functional class methods} \rangle \equiv \begin{align*}
\text{virtual matrix} \langle \mathbf{T}, \text{unstructured, dense} \rangle \ & \text{&grad} \langle \text{const matrix} \langle \mathbf{T}, \text{unstructured, dense} \rangle \ & \& x, \text{matrix} \langle \mathbf{T}, \text{unstructured, dense} \rangle \ & \& \text{dest} \rangle \\
\{ \\
\text{fval} \leftarrow F-\text{eval}(x, \ & \& \text{fval}); \\
\text{fval} \leftarrow Y; \\
F-\text{jacobian}(x, \ & \& \text{aux}); \\
\text{atxmul}(\text{aux}, \text{fval}, \ & \& \text{dest}); \\
\text{dest} *= 2; \\
\text{return dest;}
\}
\end{align*}
\]
The Hessian is a little bit more complicated. It is given by

\[ 2 \sum_i (f_i(x) - y_i) \nabla^2 f_i(x) + 2 Df(x)^T Df(x). \]

We have all the necessary algebraic functions already defined, though.

```cpp
virtual matrix<T, symmetric, dense> &hess(const matrix<T, unstructured, dense> &x, matrix<T, symmetric, dense> &dest)
{
    outerp(F->jacobian(x, &aux), &dest);
    fval ← F->eval(x, &fval);
    fval ← Y;
    for (index i ← 1; i ≤ Y.rows(); ++i) {
        H ← F->hess(x, &H, i);
        H *= fval(i);
        dest += H;
    }
    dest *= 2;
    return dest;
}
```

The product of two functionals.

/* Empty, waiting for export */
298. ⟨functional/prod.h  298⟩ ≡
   #ifndef __MATH_PROD_FUNCTIONAL__
   #define __MATH_PROD_FUNCTIONAL__  1.00
   #include <math/functionalbase.h>
   namespace math {
     namespace functional {
       template <class T>
       class prod : public base<T> {
         ⟨Product of functionals internal variables 299⟩
       public:
         ⟨Product of functionals class methods 300⟩
       };
     }
   }
   #endif

299. ⟨Product of functionals internal variables 299⟩ ≡
   typedef base<T> *element_type;

300. ⟨Product of functionals class methods 300⟩ ≡
   prod(element_type new_f, element_type new_g):f(new_f), g(new_g) { }

301. ⟨Product of functionals class methods 300⟩ +≡
   T eval(const matrix<T, unstructured, dense> &x) { return f-val(x) * g-val(x); }

302. The gradient of the product is given by \( f \nabla g + g \nabla f \).
   ⟨Product of functionals internal variables 299⟩ +≡
   matrix(T, unstructured, dense) aux;

303. ⟨Product of functionals class methods 300⟩ +≡
   matrix(T, unstructured, dense) &grad(const matrix<T, unstructured, dense>
   &x, matrix<T, unstructured, dense> &dest)
   { f-grad(x, aux);
     aux *= g-val(x);
     g-grad(x, dest);
     dest *= f-val(x);
     return dest += aux;
   }

304. The Hessian is given by \( g \nabla^2 f + \nabla f \nabla^T g + \nabla g \nabla^T f + f \nabla^2 g \).
   ⟨Product of functionals internal variables 299⟩ +≡
   matrix(T, unstructured, dense) xua;
   matrix(T, symmetric, dense) Hdense;
   matrix(T, symmetric, sparse) Hsparse;
305. 〈Product of functionals class methods 300〉 +≡

\[
\text{matrix}(T, \text{symmetric}, \text{dense}) \quad \& \quad \text{hess}(\text{const matrix}(T, \text{unstructured}, \text{dense}) \\
\quad \quad \quad \quad \& \quad \text{dest})
\]

\{
\begin{align*}
& f\text{-hess}(x, \text{dest}); \\
& \text{dest} \ast= g\text{-eval}(x); \\
& g\text{-hess}(x, H\text{dense}); \\
& H\text{dense} \ast= f\text{-eval}(x); \\
& \text{dest} += H\text{dense}; \\
& f\text{-grad}(x, aux); \\
& g\text{-grad}(x, xua); \\
& \text{dest} += xyyx(aux, xua, &H\text{dense}); \\
& \text{return} \text{ dest};
\end{align*}
\}

\[
\text{matrix}(T, \text{symmetric}, \text{sparse}) \quad \& \quad \text{hess}(\text{const matrix}(T, \text{unstructured}, \text{dense}) \\
\quad \quad \quad \quad \& \quad \text{dest})
\]

\{
\begin{align*}
& f\text{-hess}(x, \text{dest}); \\
& \text{dest} \ast= g\text{-eval}(x); \\
& g\text{-hess}(x, H\text{sparse}); \\
& H\text{sparse} \ast= f\text{-eval}(x); \\
& \text{dest} += H\text{sparse}; \\
& f\text{-grad}(x, aux); \\
& g\text{-grad}(x, xua); \\
& \text{dest} += xyyx(aux, xua, &H\text{sparse}); \\
& \text{return} \text{ dest};
\end{align*}
\}

306. The ratio of two functionals.

\[
\text{/} * \text{ Empty, waiting for export } * /\n\]

307. 〈functional/ratio.h 307〉 ≡

\[
\text{#ifndef } \_\_MATH\_RATIO\_FUNCTIONAL\_\_ \\
\text{#define } \_\_MATH\_RATIO\_FUNCTIONAL\_\_ 1.00 \\
\text{#include } <\text{math/functionalbase.h}>
\]

\[
\text{namespace math } \{
\text{namespace functional}
\]

\{
\begin{align*}
& \text{template<class T>}
& \text{class ratio:public base}(T) \{ \\
& \quad \{ \text{Ratio of functionals internal variables 308} \}
& \text{public:}
& \quad \{ \text{Ratio of functionals class methods 309} \}
& \};
\end{align*}
\}

\text{#endif}
308. ⟨Ratio of functionals internal variables 308⟩ ≡

```
typedef base(T) *element_type;
```

element_type f, g;

See also sections 310 and 312.

This code is used in section 307.

309. ⟨Ratio of functionals class methods 309⟩ ≡

```
ratio(element_type new_f, element_type new_g): f(new_f), g(new_g) { }
```

```
T eval(const matrix(T, unstructured, dense) &x) { return f-eval(x)/g-eval(x); }
```

See also sections 311 and 313.

This code is used in section 307.

310. The gradient of the ratio is given by $\nabla f/g - f \nabla g / g^2$.

⟨Ratio of functionals internal variables 308⟩ ⊔

```
matrix(T, unstructured, dense) aux;
```

311. ⟨Ratio of functionals class methods 309⟩ ⊔

```
matrix(T, unstructured, dense) &grad(const matrix(T, unstructured, dense)
&x, matrix(T, unstructured, dense) &dest)
```

```
{ 
    T result ← g-eval(x);
    g-grad(x, aux);
    aux *= -f-eval(x)/result;
    f-grad(x, dest);
    dest += aux;
    return dest /= result;
}
```

312. The Hessian is given by $\nabla^2 f/g - \nabla g \nabla^T f/g^2 - \nabla f \nabla^T g/g^2 + 2 f \nabla g \nabla^T g/g^3 - f \nabla^2 g/g^2$.

⟨Ratio of functionals internal variables 308⟩ ⊔

```
matrix(T, unstructured, dense) xua;
matrix(T, symmetric, dense) Hdense;
matrix(T, symmetric, sparse) Hsparse;
```
313. 〈Ratio of functionals class methods 309〉 ≡

\[
\text{matrix}(T, \text{symmetric, dense}) \ & \ hess(\text{const matrix}(T, \text{unstructured, dense}) \ & \ x, \text{matrix}(T, \text{symmetric, dense}) \ & \ dest)
\]

\{
   T \ result \leftarrow g-eval(x);
   T \ tluser \leftarrow f-eval(x);
   f-grad(x, aux);
   g-grad(x, xua);
   outerp(xua, &dest);
   dest **= 2 * tluser / result;
   dest \ -= xynn(aux, xua, &Hdense);
   g-hess(x, Hdense);
   Hdense **= tluser;
   dest \ -= Hdense;
   dest /= result;
   dest += f-hess(x, Hdense);
   return dest /= result;
\}

\text{matrix}(T, \text{symmetric, sparse}) \ & \ hess(\text{const matrix}(T, \text{unstructured, dense}) \ & \ x, \text{matrix}(T, \text{symmetric, sparse}) \ & \ dest)

\{
   T \ result \leftarrow g-eval(x);
   T \ tluser \leftarrow f-eval(x);
   f-grad(x, aux);
   g-grad(x, xua);
   outerp(xua, &dest);
   dest **= 2 * tluser / result;
   dest \ -= xynn(aux, xua, &Hsparse);
   g-hess(x, Hsparse);
   Hsparse **= tluser;
   dest \ -= Hsparse;
   dest /= result;
   dest += f-hess(x, Hsparse);
   return dest /= result;
\}

314.  The power functional.

\/* Empty, waiting for export */
315. \langle\text{functional/power.h} \ 315\rangle \equiv 

```
#ifndef __MATH_POWER_FUNCTIONAL__
#define __MATH_POWER_FUNCTIONAL__ 1.00
#include <math.h>
#include <math/functionalbase.h>
namespace math {
    namespace functional {
        template<class T>
        class power : public base<T> {
            (Power of a functional internal variables 316)
            public:
                (Power of a functional class methods 317)
            };
        }
    }
    #endif
```

316. \langle\text{Power of a functional internal variables 316}\rangle \equiv 
```
typedef base<T> *element_type;
element_type f;
double exponent;
```

See also section 321.
This code is used in section 315.

317. \langle\text{Power of a functional class methods 317}\rangle \equiv 
```
power(element_type new_f, double new_e): f(new_f), exponent(new_e) {
```
See also sections 318, 320, and 322.
This code is used in section 315.

318. \langle\text{Power of a functional class methods 317}\rangle +\equiv 
```
T eval(const matrix<T, unstructured, dense> &x) {
    T result ← f-eval(x);
    return pow(result, exponent);
}
```

319. The gradient of the power is given by \( nf^{n-1} \nabla f \).

320. \langle\text{Power of a functional class methods 317}\rangle +\equiv 
```
matrix(T, unstructured, dense) &grad(const matrix<T, unstructured, dense>
    &x, matrix(T, unstructured, dense) &dest) {
    T result ← f-eval(x);
    result ← exponent * pow(result, exponent - 1);
    f-grad(x, dest);
    return dest *= result;
}
```
321. The Hessian is given by
\[ nf^{n-1} \nabla^2 f + n(n - 1) f^{n-2} \nabla f \nabla f^T. \]

\[ \langle \text{Power of a functional internal variables } 316 \rangle + \equiv \]
\[ \text{matrix}(T, \text{symmetric, dense}) \ H_{\text{dense}}; \]
\[ \text{matrix}(T, \text{symmetric, sparse}) \ H_{\text{sparse}}; \]
\[ \text{matrix}(T, \text{unstructured, dense}) \ aux; \]

322. \( \langle \text{Power of a functional class methods } 317 \rangle + \equiv \)
\[ \text{matrix}(T, \text{symmetric, dense}) \ & \text{hess}(\text{const matrix}(T, \text{unstructured, dense}) \ & x, \text{matrix}(T, \text{symmetric, dense}) \ & dest) \]
\[ \{ \]
\[ T \ result \leftarrow f\text{-eval}(x); \]
\[ \text{if} \ (result \equiv 0 \land \text{exponent} \geq 2) \ {\}
\[ dest.\text{resize}(x.rows(), x.rows()); \]
\[ dest.\text{fillwith}(0.0); \]
\[ \text{return} \ dest; \]
\[ \} \]
\[ \text{if} \ (result \equiv 0 \land \text{exponent} < 2) \ \text{throw} \ \text{error} :: \text{domain}(); \]
\[ f\text{-grad}(x, aux); \]
\[ \text{outerp}(aux, \& H_{\text{dense}}); \]
\[ H_{\text{dense}} \leftarrow (\text{exponent} - 1) / result; \]
\[ f\text{-hess}(x, dest); \]
\[ dest + = H_{\text{dense}}; \]
\[ \text{return} \ dest \leftarrow \text{pow}(result, \text{exponent} - 1) \ast \text{exponent}; \]
\[ \} \]
\[ \text{matrix}(T, \text{symmetric, sparse}) \ & \text{hess}(\text{const matrix}(T, \text{unstructured, dense}) \ & x, \text{matrix}(T, \text{symmetric, sparse}) \ & dest) \]
\[ \{ \]
\[ T \ result \leftarrow f\text{-eval}(x); \]
\[ \text{if} \ (result \equiv 0 \land \text{exponent} \geq 2) \ {\}
\[ dest.\text{resize}(x.rows(), x.rows()); \]
\[ dest.\text{fillwith}(0.0); \]
\[ \text{return} \ dest; \]
\[ \} \]
\[ \text{if} \ (result \equiv 0 \land \text{exponent} < 2) \ \text{throw} \ \text{error} :: \text{domain}(); \]
\[ f\text{-grad}(x, aux); \]
\[ \text{outerp}(aux, \& H_{\text{sparse}}); \]
\[ H_{\text{sparse}} \leftarrow (\text{exponent} - 1) / result; \]
\[ f\text{-hess}(x, dest); \]
\[ dest + = H_{\text{sparse}}; \]
\[ \text{return} \ dest \leftarrow \text{pow}(result, \text{exponent} - 1) \ast \text{exponent}; \]
\[ \} \]

323. The entropy of a functional.

\[
\text{\textast{ Empty, waiting for export \textast{}}}
\]
324. \( \langle \text{functional/entr.h} \rangle \equiv \)

```c
#ifndef __MATH_ENTROPY_FUNCTIONAL__
#define __MATH_ENTROPY_FUNCTIONAL__ 1.00
#include <math.h>
#include <math/functionalbase.h>
namespace math {
    namespace functional {
        template \langle class T \rangle
            class entr:public base \langle T \rangle {
            \langle Entropy of a functional internal variables 325 \rangle
            public:
            \langle Entropy of a functional class methods 326 \rangle
        };
    }
}
#endif
```

325. \( \langle \text{Entropy of a functional internal variables 325} \rangle \equiv \)

```c
typedef base \langle T \rangle * element_type;
```

See also section 330.
This code is used in section 324.

326. \( \langle \text{Entropy of a functional class methods 326} \rangle \equiv \)

```c
entr(element_type new_f):f(new_f) \{ \}
```

See also sections 327, 329, and 331.
This code is used in section 324.

327. \( \langle \text{Entropy of a functional class methods 326} \rangle +\equiv \)

```c
T eval(const matrix \langle T, unstructured, dense \rangle &x) {
    T result ← f-eval(x);
    if (result ≤ 0) throw error::domain();
    return -result * ::log(result);
}
```

328. The gradient of the entropy is given by \( -(1 + \log f)\nabla f \).

329. \( \langle \text{Entropy of a functional class methods 326} \rangle +\equiv \)

```c
matrix \langle T, unstructured, dense \rangle & grad(const matrix \langle T, unstructured, dense \rangle &x, matrix \langle T, unstructured, dense \rangle & dest) {
    T result ← f-eval(x);
    if (result ≤ 0) throw error::domain();
    f-grad(x, dest);
    return dest *= -(::log(result) + 1);
}
```
330. The Hessian is given by

\[-(1 + \log f)\nabla^2 f - \frac{1}{f} \nabla f \nabla^T f.\]

\[\langle \text{Entropy of a functional internal variables} \rangle + \equiv \]
- matrix(T, symmetric, dense) \( H_{\text{dense}} \);
- matrix(T, symmetric, sparse) \( H_{\text{sparse}} \);
- matrix(T, unstructured, dense) \( \text{aux} \);

331. \( \langle \text{Entropy of a functional class methods} \rangle + \equiv \)
- matrix(T, symmetric, dense) \&hess(const matrix(T, unstructured, dense) \&x, matrix(T, symmetric, dense) \&dest)

\{ \[\]
- \( T \) result ← f-eval(x);
- if (result ≤ 0) throw error::domain();
- f-grad(x, aux);
- outerp(aux, &H_{\text{dense}});
- \( H_{\text{dense}} /= (-\text{result});
- f-hess(x, dest);
- dest *= -(::\log(result) + 1);
- return dest += H_{\text{dense}};
\}

- matrix(T, symmetric, sparse) \&hess(const matrix(T, unstructured, dense) \&x, matrix(T, symmetric, sparse) \&dest)

\{ \[\]
- \( T \) result ← f-eval(x);
- if (result ≤ 0) throw error::domain();
- f-grad(x, aux);
- outerp(aux, &H_{\text{sparse}});
- \( H_{\text{sparse}} /= (-\text{result});
- f-hess(x, dest);
- dest *= -(::\log(result) + 1);
- return dest += H_{\text{sparse}};
\}

332. Relative entropy. The relative entropy (also known as Kullback Leibler distance) function is given by \( x \log \frac{x}{y} \). The relative entropy is a measure of how much do we loose for assuming that the distribution of some random variable is \( y \) when the true distribution is \( x \). If we knew \( x \), then we could describe the random variable with a code with average description length \( \text{entr}(x) \). If we use \( y \), however, we would need \( \text{entr}(x) + \text{relentr}(x, y) \) to describe the variable.

\/* Empty, waiting for export */
333. \(\text{\{functional/relentr.h 333\} } \equiv \)

\#ifndef \_MATH\_RELATIVE\_ENTROPY\_FUNCTIONAL\_
\#define \_MATH\_RELATIVE\_ENTROPY\_FUNCTIONAL\_ 1.00
\#include <math.h>
\#include <math/functionalbase.h>

namespace math { namespace functional {
    template\langle class T\rangle
    class relentr : public base\langle T\rangle {
        \{ Relative entropy functional internal variables 334 \}
        public:\
        \{ Relative entropy functional class methods 335 \} ;
    } ;
} }
\#endif

334. \(\text{\{Relative entropy functional internal variables 334\} } \equiv \)

typedef base\langle T \rangle *element_type;

element_type f, g;

See also sections 337 and 339.
This code is used in section 333.

335. \(\text{\{Relative entropy functional class methods 335\} } \equiv \)

relentr\langle element_type new. f, element_type new. g \rangle :

\{ \}

See also sections 336, 338, and 340.
This code is used in section 333.

336. \(\text{\{Relative entropy functional class methods 335\} } + \equiv \)

T eval\langle const matrix\langle T, unstructured, dense \rangle &x \rangle {
    T fval ← f-eval\langle x \rangle ;
    T gval ← g-eval\langle x \rangle ;
    if \(gval ≤ 0 \text{\lor \ fval ≤ 0}\) throw error::domain\langle \rangle ;
    return fval * ::log\langle fval / gval \rangle ;
} 

337. The gradient of the relative entropy is given by \((1 + \log(f/g))\nabla f - (f/g)\nabla g.\)

\(\text{\{Relative entropy functional internal variables 334\} } + \equiv \)

matrix\langle T, unstructured, dense \rangle auxvec ;
338. (Relative entropy functional class methods \(335\)) \(\equiv\)

\[
\text{\text{matrix}}(\text{T, unstructured, dense}) \& \text{\text{grad}}(\text{const matrix}(\text{T, unstructured, dense}) \& \text{dest})
\]

\[
\begin{cases}
\text{T fval} ← f\text{-eval}(x);
\text{T gval} ← g\text{-eval}(x);
\text{if (gval} ≤ 0 \lor \text{fval} ≤ 0) \text{ throw error}::\text{domain}();
\text{g-grad}(x, \text{auxvec});
\text{auxvec} *= \text{fval}/\text{gval};
\text{f-grad}(x, \text{dest});
\text{dest} *= (:\log(\text{fval}/\text{gval}) + 1);
\text{return dest} ← \text{auxvec};
\end{cases}
\]

339. The Hessian is given by \((1 + \log(f/g))\nabla^2 f - (f/g)\nabla^2 g + f(\nabla g/g - \nabla f/f)(\nabla g/g - \nabla f/f)^T\).

\[
\text{\text{matrix}}(\text{T, unstructured, dense}) \text{ auxgrad};
\text{\text{matrix}}(\text{T, symmetric, dense}) \text{ auxdense};
\text{\text{matrix}}(\text{T, symmetric, sparse}) \text{ auxsparse};
\]

340. (Relative entropy functional class methods \(335\)) \(\equiv\)

\[
\text{\text{matrix}}(\text{T, symmetric, dense}) \& \text{\text{hess}}(\text{const matrix}(\text{T, unstructured, dense}) \& \text{dest})
\]

\[
\begin{cases}
\#define auxhess auxdense
\langle \text{relentr Hessian 341} \rangle;
\#undef auxhess
\end{cases}
\]

\[
\text{\text{matrix}}(\text{T, symmetric, sparse}) \& \text{\text{hess}}(\text{const matrix}(\text{T, unstructured, dense}) \& \text{dest})
\]

\[
\begin{cases}
\#define auxhess auxsparse
\langle \text{relentr Hessian 341} \rangle;
\#undef auxhess
\end{cases}
\]

\[
\]
341. \( \langle \text{relentr} \rangle \) Hessian 341 \( \equiv \)

\[
\begin{align*}
T \ fval & \leftarrow f-eval(x); \\
T \ gval & \leftarrow g-eval(x); \\
\text{if} \ (gval \leq 0 \lor fval \leq 0) \ & \text{throw error}::\text{domain}(); \\
g-\text{grad}(x, auxvec); \\
f-\text{grad}(x, auxgrad); \\
auxvec /= gval; \\
auxgrad /= fval; \\
\text{outerp}(auxvec \ -= auxgrad, &dest); \\
dest *= fval; \\
g-\text{hess}(x, auxhess); \\
auxhess *= fval/gval; \\
dest -= auxhess; \\
f-\text{hess}(x, auxhess); \\
auxhess *= (::\log(fval/gval) + 1); \\
\text{return} \ dest += auxhess;
\end{align*}
\]

This code is used in section 340.

342. The Error Function. The error function (erf) is defined as

\[
\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-u^2} du
\]

\[
/* \text{Empty, waiting for export} */
\]

343. \( \langle \text{functional/erf.h} \rangle \) \( \equiv \)

\[
\begin{align*}
\#ifndef \ _\text{MATH}_ERF\_FUNCTIONAL_\-
\#define \ _\text{MATH}_ERF\_FUNCTIONAL_\- \\
\#include <\text{math.h}> \\
\#include <\text{math/functionalbase.h}>
\text{namespace math} \ \{ \text{namespace functional} \ \\
\text{template}(<\text{class T}) \\
\text{class erf:public base}<\text{T}> \ \{ \\
\text{\langle Error function of a functional internal variables 344\rangle} \\
\text{public:} \\
\text{\langle Error function of a functional class methods 345\rangle} \\
\}; \\
\text{\}}
\#endif
\end{align*}
\]

344. \( \langle \text{Error function of a functional internal variables 344} \rangle \) \( \equiv \)

\[
\text{typedef base}<\text{T}> \ * \text{element_type};
\text{element_type} \ f;
\]

See also section 349.

This code is used in section 343.

345. \( \langle \text{Error function of a functional class methods 345} \rangle \) \( \equiv \)

\[
\text{erf(element_type} \ new_f):f(new_f) \ \{
\}
\]

See also sections 346, 348, and 350.

This code is used in section 343.
346. \(\langle\) Error function of a functional class methods 345 \(\rangle\) \(\equiv\)
\[
T\eval(const\ matrix\ (T,\ unstructured,\ dense)\ &x)
\{
return (::erf(f-eval(x)));\}
\]

347. The gradient of the error function is given by \(\frac{2}{\sqrt{\pi}}e^{-f^2}\nabla f\).

348. \(\langle\) Error function of a functional class methods 345 \(\rangle\) \(\equiv\)
\[
\text{matrix}\ (T,\ unstructured,\ dense)\ &\grad(const\ matrix\ (T,\ unstructured,\ dense))
\&x,\ matrix\ (T,\ unstructured,\ dense)\ &dest)
\{
T\ result \leftarrow f-eval(x);
\f\grad(x,\ dest);
return dest \leftarrow 2/::\sqrt{\text{M_PI}} \ast ::\exp(-result \ast result);
\}
\]

349. The Hessian is given by \(\frac{2}{\sqrt{\pi}}e^{-f^2}\nabla^2 f - \frac{4}{\sqrt{\pi}}f e^{-f^2}\nabla \nabla^T f\).

350. \(\langle\) Error function of a functional class methods 345 \(\rangle\) \(\equiv\)
\[
\text{matrix}\ (T,\ symmetric,\ dense)\ \text{dense}_\text{aux};
\text{matrix}\ (T,\ symmetric,\ sparse)\ \text{sparse}_\text{aux};
\text{matrix}\ (T,\ unstructured,\ dense)\ \text{aux};
\]

351. \(\langle\) Compute Hessian for erf functional 351 \(\rangle\) \(\equiv\)
\[
T\ result \leftarrow f-eval(x);
f\grad(x,\ aux);
f\hess(x,\ dest);
outerp(aux,\ &\hess\_aux);
hess\_aux \leftarrow -2 \ast result;
dest \leftarrow hess\_aux;
return dest \leftarrow 2 \ast ::\exp(-result \ast result)/::\sqrt{\text{M_PI}};
\]
This code is used in section 350.
352. The exponential of a functional.
/* Empty, waiting for export */

353. ⟨functional/exp.h 353⟩ ≡
2004 #ifndef __MATH_EXP_FUNCTIONAL__
2005 #define __MATH_EXP_FUNCTIONAL__ 1.00
2006 #include <math.h>
2007 #include <math/functionalbase.h>
2008 namespace math {
2009 namespace functional {
2010   template ⟨class T⟩
2011   class exp:public base ⟨T⟩ {
2012     ⟨Exponential of a functional internal variables 354⟩
2013     public:
2014     ⟨Exponential of a functional class methods 355⟩
2015   };  
2016 }
2017 }  
2018 #endif

354. ⟨Exponential of a functional internal variables 354⟩ ≡
2020 typedef base ⟨T⟩ *element_type;
2021 element_type f:
See also section 359.
This code is used in section 353.

355. ⟨Exponential of a functional class methods 355⟩ ≡
2022 exp(element_type new_f):f(new_f) {  }
See also sections 356, 358, and 360.
This code is used in section 353.

356. ⟨Exponential of a functional class methods 355⟩ +=
2023 T eval(const matrix ⟨T, unstructured, dense⟩ &x)
2024 {  
2025     return ::exp(f->eval(x));
2026 }

357. The gradient of the exponential is given by $e^f \nabla f$.

358. ⟨Exponential of a functional class methods 355⟩ +=
2027 matrix ⟨T, unstructured, dense⟩ &grad(const matrix ⟨T, unstructured, dense⟩
2028 &x, matrix ⟨T, unstructured, dense⟩ &dest)  
2029 {  
2030     f->grad(x, dest);
2031     return dest *= eval(x);  
2032 }
The Hessian is given by $e^f \nabla^2 f + e^f \nabla f \nabla^T f$. 

359. The Hessian is given by $e^f \nabla^2 f + e^f \nabla f \nabla^T f$.

360. The logarithm of a functional.

```cpp
#include <math.h>
#include <math/functionalbase.h>
namespace math {
    namespace functional {
        template <class T> class log : public base<T> {
            // Logarithm of a functional internal variables
            public:
                // Logarithm of a functional class methods
            }
    }
}
#endif
```

361. The logarithm of a functional.

```cpp
#include <math.h>
#include <math/functionalbase.h>
namespace math {
    namespace functional {
        template <class T> class log : public base<T> {
            // Logarithm of a functional internal variables
            public:
                // Logarithm of a functional class methods
            }
    }
}
#endif
```
363. The functionals to be added are stored in a vector of pairs of weights and functionals.

\[ \langle \text{Logarithm of a functional internal variables 363} \rangle \equiv \]

```cpp
typedef base(T) *element_type;
```

\[ \text{element_type f;} \]

See also section 368.

This code is used in section 362.

364.

\[ \langle \text{Logarithm of a functional class methods 364} \rangle \equiv \]

```cpp
log(element_type new_f):f(new_f) { }
```

See also sections 365, 367, and 369.

This code is used in section 362.

365.

\[ \langle \text{Logarithm of a functional class methods 364} \rangle + \equiv \]

```cpp
T eval(const matrix(T, unstructured, dense) &x) {
    T result ← ::log(f-eval(x));
    if (¬finite(result)) throw error::domain();
    return result;
}
```

366. The gradient of the logarithm is given by \( \nabla f / f \).

367. \[ \langle \text{Logarithm of a functional class methods 364} \rangle + \equiv \]

```cpp
matrix(T, unstructured, dense) &grad(const matrix(T, unstructured, dense) &x, matrix(T, unstructured, dense) &dest) {
    T result ← f-eval(x);
    f-grad(x, dest);
    return dest /= result;
}
```

368. The Hessian is given by \( \nabla^2 f / f - \nabla f \nabla^T f / f^2 \).

\[ \langle \text{Logarithm of a functional internal variables 363} \rangle + \equiv \]

```cpp
matrix(T, symmetric, dense) Hdense;
matrix(T, symmetric, sparse) Hsparse;
matrix(T, unstructured, dense) aux;
```


369. \langle \text{Logarithm of a functional class methods 364} \rangle \equiv 
\begin{align*}
\text{matrix}(T, \text{symmetric, dense}) & \& \text{hess}(\text{const matrix}(T, \text{unstructured, dense}) \\
& \& x, \text{matrix}(T, \text{symmetric, dense}) \& dest) \\
\end{align*}

\begin{verbatim}
{
    T result ← f-eval(x);
    f-grad(x, aux);
    outerp(aux, &Hdense);
    Hdense /= (-result);
    f-hess(x, dest);
    dest += Hdense;
    return dest /= result;
}
\end{verbatim}

\begin{verbatim}
{
    T result ← f-eval(x);
    f-grad(x, aux);
    outerp(aux, &Hsparse);
    Hdense /= (-result);
    f-hess(x, dest);
    dest += Hsparse;
    return dest /= result;
}
\end{verbatim}
370. **Line Searching.** We begin now the definitions that will enable us to perform various types of numerical optimization. The first task we face is the one of minimizing a functional along a line. There are many algorithms to perform this task, and which to use will depend heavily on the specific problem (for example, is it cheap to compute the gradient, to evaluate the function at a point and so on).

```c++
// Empty, waiting for export. */
```

371. ⟨linesearchbase.h 371⟩ ≡

```c++
#ifndef __MATH_LINESEARCH__
#define __MATH_LINESEARCH__ 1
#include <math/functionalbase.h>
namespace math {
    namespace linesearch {
        template⟨class T⟩
        class base {
            public:
                virtual ~base(void) { }
        }; // Line search base class methods 372
    }
}
#endif
```

372. The interface is very simple: there is one method that applies the minimization algorithm to a given functional, starting from a given point and searching in a given direction. The syntax is `minimize(functional, x0, dir)`. The direction does not always need to have unity norm, but you better check out the specific algorithm to be sure that this is the case. The `minimize` method returns the minimizing point. As a design decision, it would be beneficial in some cases (for example in the backtracking algorithm) to include the gradient in the list of arguments, since the gradient would probably be available from the computation of the search direction. Since we cannot require this (for example, in some cases the gradient would have to be computed via some expensive simulation), we provide an alternative method that takes the gradient at the starting point as the third argument (the search direction becomes the fourth).

```c++
virtual matrix⟨T, unstructured, dense⟩ minimize(functional::base⟨T⟩ *, const
matrix⟨T, unstructured, dense⟩ &, const matrix⟨T, unstructured, dense⟩ &) ← 0;
virtual matrix⟨T, unstructured, dense⟩ minimize(functional::base⟨T⟩ *, const
matrix⟨T, unstructured, dense⟩ &, const matrix⟨T, unstructured, dense⟩ &) ← 0;
```

This code is used in section 371.

373. **The bisection algorithm.** Although not really a true bisection algorithm, it works by halving the step size when appropriate as we will see. This is only an example of line searching, but this algorithm can fail miserably in practice: depending on the function and on the disposition of local minima, you can end up in a local minima that is bigger than the closest one. You’re advised do use other methods if possible. That said, we move on: As with many algorithms, we must provide a stopping criterion. In this case, the algorithm stops whenever `∥x − x∗∥ ≤ tol`, where `tol` is a given tolerance. Also, some people claim that halving the step size is not always the best thing to do. We then provide a means to alter this behavior via a `ratio` parameter, so that when the step size is changed, it changes via `step ← step * ratio`. Of course, we need `0 < ratio ⟨1 for the algorithm to work.

```c++
/* Empty, waiting for export. */
```
374. \langle \text{linesearch/bisection.h} \rangle \equiv

```cpp
#ifndef __MATH_BISECTION__
#define __MATH_BISECTION__ 1.0
#include <math.h>

const double bisection_infty ← HUGE_VAL;

#include <math/algebra.h>
#include <math/linesearchbase.h>
namespace math {
    namespace linesearch {
        template <class T>
        class bisection:public base<T> {
            double tol,
            ratio,
            unsigned long maxiter;  /* Stop in case its unbounded below or something. */
        public: ~bisection(void)
            {}
            (Bisection line search methods 375)
        }
            (bisection big definitions 378);
        }
```

```cpp```

375. First we provide a constructor that enable us to set all parameters at creation time, with some defaults in case we don’t care about it.

\langle Bisection line search methods 375 \rangle \equiv

```cpp
bisection(double t ← 1 \cdot 10^{-3}, double r ← 0.5, unsigned long i ← 1000)
: tol(t), ratio(r), maxiter(i) {}
```

See also sections 376 and 377.

This code is used in section 374.

376. The first thing we do is to define the method that takes the gradient as an argument. We don’t use the gradient for this algorithm, so the only thing we do is to call the other method.

\langle Bisection line search methods 375 \rangle +≡

```cpp
matrix(T, unstructured, dense) minimize(functional::base<T> \ast f,
    const matrix(T, unstructured, dense) \&x0, const matrix(T, unstructured, dense)
    \&y0, const matrix(T, unstructured, dense) \&dir)
    {}
    return minimize(f, x0, dir);
```
377. The algorithm works as follows: we make \( x \leftarrow x_0 \). At each step, we start from \( x \) and go in the descending direction until we pass the minimum, as indicated by an increase in the function. At this point, we know the descent direction will be to the other side, so we turn around and proceed slower, that is, with a smaller stepsize. The stopping criterion is independent of \( x^\ast \) and \( x \) since
\[
\| x - x^\ast \| \leq \| x - (x + \delta x) \| = \| \delta x \| ,
\]
where \( \| \delta x \| \) is a positive multiple of \( \| dir \| \). We consider that reaching the maximum number of iterations is an error. Note that the maximum number of iterations is with respect to the inner loop, since the outer loop always finishes. The algorithm follows:

This code is used in section 374.
379. ⟨Compute next function value for bisection 379⟩ ≡

```c
saxpy(stepsize, dir, &x);
try {
    newf ← func-eval(x);
} catch(error::domain e) {
    newf ← bisection_infty;
} while (¬finite(newf)) {
    saxpy(−stepsize, dir, &x); stepsize *= ratio;
    saxpy(stepsize, dir, &x);
    try {
        newf ← func-eval(x);
    } catch(error::domain e) {
        newf ← bisection_infty;
    }
} if (stepsize ≡ 0) return x;
```

This code is used in section 378.

380. **Backtracking.** This line search algorithm uses the gradient information. We begin with a unit step $\lambda$, and until we have

$$f(x_0 + \lambda \cdot dir) \leq f(x_0) + \lambda \alpha \nabla f(x_0)^T dir,$$

we update the step with $\beta \lambda$. The algorithm parameters are $\alpha$ and $\beta$, where $0 < \alpha < 0.5$ and $0 < \beta < 1$.

```c
/* Empty, waiting for export. */
```

381. ⟨linesearch/backtracking.h 381⟩ ≡

```c
ifndef __MATH_BACKTRACKING_LINESEARCH__
define __MATH_BACKTRACKING_LINESEARCH__ 1.0
const double backtracking_infty ← HUGE_VAL;
/* Some compilers have problems using HUGE_VAL. */
#include <math/algebra.h>
#include <math/linesearchbase.h>
namespace math {
    namespace linesearch {
        template<class T>
        class backtracking:public base<T> {
            double alpha, beta;
            unsigned long maxiter;
            public: ~backtracking{void }{
            }
            ⟨Backtracking line search methods 382⟩
        }
    }
}
#endif
```
382. First we provide a constructor that enable us to set all parameters at creation time, with some defaults in case we don’t care about it.

\[
\text{backtracking}(\text{double } a \leftarrow 0.3, \text{double } b \leftarrow 0.8, \text{unsigned long } m \leftarrow 1000)
\]

See also sections 383 and 384.

This code is used in section 381.

383. The first thing we do is to define the method that doesn’t take the gradient as an argument. Since we need the gradient, what this method does is to compute it and call the correct method.

\[
\text{matrix} \langle \text{T}, \text{unstructured}, \text{dense} \rangle \text{ minimize} \langle \text{functional} :: \text{base} \langle \text{T} \rangle \ast f, \text{const matrix} \langle \text{T}, \text{unstructured}, \text{dense} \rangle \& x0, \text{const matrix} \langle \text{T}, \text{unstructured}, \text{dense} \rangle \& dir \rangle
\]

\[
\text{matrix} \langle \text{T}, \text{unstructured}, \text{dense} \rangle g0;
\]

\[
g0 \leftarrow f\text{-grad}(x0, g0);
\]

\[
\text{return minimize}(f, x0, g0, dir);
\]

384.

\[
\text{matrix} \langle \text{T}, \text{unstructured}, \text{dense} \rangle \text{ minimize} \langle \text{functional} :: \text{base} \langle \text{T} \rangle \ast \text{func}, \text{const matrix} \langle \text{T}, \text{unstructured}, \text{dense} \rangle \& x0, \text{const matrix} \langle \text{T}, \text{unstructured}, \text{dense} \rangle \& g0, \text{const matrix} \langle \text{T}, \text{unstructured}, \text{dense} \rangle \& dir \rangle
\]

\[
x;
\]

\[
double \text{ ftreshold} \leftarrow \text{func-eval}(x0);
\]

\[
double \text{ gtreshold} \leftarrow \text{alpha} \ast \text{dot}(g0, dir);
\]

\[
double \text{ stepsize} \leftarrow 1/\text{beta};
\]

\[
\text{unsigned long } \text{ iter} \leftarrow 0;
\]

\[
double \text{ fval};
\]

\[
\#ifdef \_MATH\_DEBUG\_
\]

\[
cout \leftarrow "[\text{math}]::\text{backtracking}\_line\_search\_begin.\n"
\]

\[
cout \leftarrow "[\text{math}]::x0.\n"
\]

\[
\text{for (index } i \leftarrow 1; \ i \leq x0.\text{rows}(); \ ++i \) cout \leftarrow x0(i) \leftarrow ', n' \n"
\]

\[
cout \leftarrow ', n'
\]

\[
\text{for (index } i \leftarrow 1; \ i \leq dir.\text{rows}(); \ ++i \) cout \leftarrow dir(i) \leftarrow ', n' \n"
\]

\[
cout \leftarrow ', n'
\]

\[
\#endif
\]

\[
do {\ 
\text{\{Compute next function value for backtracking 385\};}
\]

\[
\text{if (} ++\text{iter }\equiv \text{maxiter} \text{) throw error::maxiterations();}
\]

\[
\#ifdef \_MATH\_DEBUG\_
\]

\[
cout \leftarrow "[\text{math}]::\text{backtracking}\_line\_search\_end.\n"
\]

\[
\#endif
\]

\[
\text{return } x;
\]
385. \(\{\text{Compute next function value for backtracking}\ 385\} \equiv\)

\[
\text{fval} \leftarrow \text{backtracking}_\text{infty};
\]

\[
\text{while } (\neg \text{finite}(\text{fval})) \{
\]

\[
 x \leftarrow \text{dir};
\]

\[
 x \ast= (\text{stepsize} \ast= \beta);\]

\[
 x \mathbin{+}= x_0;
\]

\[
\#\text{ifdef } _{\_}\_\text{MATH\_DEBUG}_\text{__}
\]

\[
\text{cout} \ll \"[\text{math}]:\backslash n_\text{x}\backslash n_\text{=}\backslash n\";
\]

\[
\text{for } (\text{index } i \leftarrow 1; \ i \leq x.\text{rows}(); \ ++i) \ \text{cout} \ll x(i) \ll \'\_\';
\]

\[
\text{cout} \ll \'\n\text{}\';
\]

\[
\#\text{endif}
\]

\[
\text{try } \{
\]

\[
\text{catch}(\text{error::domain } e)
\]

\[
\{
\]

\[
\text{fval} \leftarrow \text{backtracking}_\text{infty};
\]

\[
\}
\]

\[
\#\text{ifdef } _{\_}\_\text{MATH\_DEBUG}_\text{__}
\]

\[
\text{cout} \ll \"[\text{math}]:f(x)=\" \ll \text{fval} \ll \"\_\_\text{stepsize}=\" \ll \text{stepsize} \ll \"\_\_\text{treshold}=\" \ll \
\text{ftreshold} + \text{stepsize} \times \text{gtreshold} \ll \'\n\text{}\';
\]

\[
\#\text{endif}
\]

\[
\text{if } (\text{stepsize} \equiv 0.0) \ \text{fval} \leftarrow \text{ftreshold} + \text{stepsize} \ast \text{gtreshold};
\]

\[
\}
\]

This code is used in section 384.
386. **Computing a search direction.** Once you have a line minimization algorithm the only thing that you still need in order to define a local minimization algorithm is a method for computing the search direction.

```cpp
// Empty, waiting for export.
```

387. `(searchdirbase.h 387) ≡

```cpp
#ifndef __MATH_SEARCHDIRBASE__
#define __MATH_SEARCHDIRBASE__ 1.0

#include <math/math.h>
#include <math/functionalbase.h>

namespace math {
    namespace searchdir {
        template<class T>
        class base {
            public:
                virtual ~base(void) { }
        }; // Search direction base class methods 388
    }
}
#endif
```

388. Now to the interface: The only thing a search direction method has to do is to compute a descent direction based on a functional and a starting point. Our method will return the descent direction.

```cpp
virtual matrix<T, unstructured, dense> &dir(func: base<T> *f, const matrix<T, unstructured, dense> &xi, matrix<T, unstructured, dense> &dest) ← 0;
```

This code is used in section 387.

389. **The gradient direction.** If you’re able to compute the gradient easily (that is, not requiring simulation or something) and the Hessian is too costly for you, then the gradient search direction can be a good choice.

```cpp
// Empty, waiting for export.
```
390.  \{searchdir/gradient.h \} ≡

\begin{verbatim}
# ifndef __MATH_SEARCHDIR_GRADIENT__
# define __MATH_SEARCHDIR_GRADIENT__ 1.0
#include <math/searchdirbase.h>

namespace math {
    namespace searchdir {
        template <class T>
        class gradient: public base< T > {
            public:
                ~gradient( void )
                    {}  // Gradient search direction methods 391
            }
        };  // namespace
    }
}  // namespace
# endif
\end{verbatim}

391.  The search direction is basically \(-\nabla f\).

\begin{verbatim}
<Gradient search direction methods 391> ≡

matrix<T, unstructured, dense> &dir (functional :: base<T> *f, const matrix<T, unstructured, dense> &x, matrix<T, unstructured, dense> *dest)

    { f->grad(x, *dest);
      return (*dest) *= -1;
    }
\end{verbatim}

This code is used in section 390.

392.  The Newton direction.  If you are lucky enough that the Hessian computation is not a big deal and you can afford solving a linear system each time the search direction is needed, then the Newton direction is a no-brainer.  The only thing that needs to be defined is what type of storage you want to use for the Hessian.  For large, sparse problems, you probably want a sparse storage.

    /* Empty, waiting for export. */
393. \langle \text{searchdir/newton.h} 393 \rangle \equiv

```cpp
#ifndef __MATH_SEARCHDIR_NEWTON__
#define __MATH_SEARCHDIR_NEWTON__ 1.0
#include <math/searchdirbase.h>
#include <math/algebra.h>
#include <math/lu.h>
#include <math/cholesky.h>
namespace math {
  namespace searchdir {
    template <class T, template <class> class storage>
    class newton: public base<T> {
      (Newton search direction internal variables 394)
      public:
        (Newton search direction methods 395)
    };
  }
} #endif
```

394. We assume that you'll need to compute the search direction many times, so we reserve space for the Hessian.

\langle Newton search direction internal variables 394 \rangle \equiv

```cpp
matrix(T, symmetric, storage) hess;
```

This code is used in section 393.

395. The search direction is now \(- (\nabla^2 f)^{-1} \nabla f\). We first compute the gradient and store it in aux, following by a Hessian computation and Cholesky linear system solver. In case the Hessian is singular or close, we return the gradient direction (which is still a descent direction).

\langle Newton search direction methods 395 \rangle \equiv

```cpp
matrix(T, unstructured, dense) &dir(functional::base(T) *f, const matrix(T, unstructured, dense) &x, matrix(T, unstructured, dense) *dest)
{
  try {
    try {
      f-grad(x, *dest);
      f-hess(x, hess);
      cholesky::solve(&hess, dest);
    }
    catch(error::nonpositivedef e) {
      f-grad(x, *dest);
      f-hess(x, hess);
      lu::solve(&hess, dest);
    }
    catch(error::singular e) {
      return (*dest) *= -1;
    }
  }
}
```

This code is used in section 393.
396. Enforcing equality constraints. All the search direction classes defined above are not able to take into account the search direction must, in some cases, have zero component in some directions. The most obvious and common case is when we want to minimize something while enforcing an equality constraint $Ax = b$. We will define a class that, based on a search direction $d$ computed in any way, transforms it so that $A(x + tv) = b$ for any real $t$.

    /* Empty, waiting for export. */

397.  \{searchdir/equality.h \ 397\} ≡
\begin{verbatim}
#define __MATH_SEARCHDIR_EQUALITY__ 1
#include <math/searchdirbase.h>
#include <math/symmetric.h>
#include <math/lu.h>
namespace math {
    namespace searchdir {
        template<\langle matrix_simple,template \rangle>
        class equality : public base<\langle T \rangle> {
            \langle Equality search direction internal variables 398 \rangle
            public:
                \langle Equality search direction methods 399 \rangle
        };
    }
}
#endif
\end{verbatim}

398. First we need a way to let the user define which basic search direction he wants to use.
\langle Equality search direction internal variables 398 \rangle ≡
\begin{verbatim}
    base<\langle T \rangle> *dirf;
\end{verbatim}

See also section 400.
This code is used in section 397.

399. \langle Equality search direction methods 399 \rangle ≡
\begin{verbatim}
    equality(\langle void \rangle):dirf(0) {}
    equality(\langle base<\langle T \rangle> *newf \rangle):dirf(newf) {} 
    base<\langle T \rangle> **f(\langle void \rangle) { return dirf; }
\end{verbatim}

See also sections 401 and 403.
This code is used in section 397.
Now to the $A$ matrix. Suppose that our basic direction class provides us with a preferred direction $d_0$. What we do is to compute the projection of $d_0$ into the kernel of $A$. This task can be accomplished by solving the linear system

$$\begin{bmatrix} I & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} d \\ w \end{bmatrix} = \begin{bmatrix} d_0 \\ 0 \end{bmatrix},$$

which in the end will give

$$d = (I - A^T(AA^T)^{-1}A)d_0.$$

That's the only time where we use the $A$ matrix, so we can see that we don't need to store it: what we store is the LU decomposition (the matrix is not necessarily positive-definite) of the linear system matrix, so that no matter how many directions we compute, solving the linear system is a very fast operation (no decompositions needed!). The only catch here is that we may receive (and we will when barrier functions are defined) a preferred direction with the wrong dimension. We store the correct dimension in $xrows$ and deal with it later on.

401. 〈Equality search direction methods 399〉 $+$

```cpp
equality(const matrix<T, structure, storage> &A): dirf (0) { set_A(newa); }
void set_A(const matrix<T, structure, storage> &A)
{
  #ifdef __MATH_DEBUG__
  cout << "[math]::equality_search_constructor.\n";
  for (index i ← 1; i ≤ A.rows(); ++i) {
    for (index j ← 1; j ≤ A.cols(); ++j) cout << A(i, j) << ' ';
    cout << 'n';
  }
  #endif
    xrows ← A.cols();
    decomp.resize (A.cols () + A.rows (), A.cols () + A.rows ());
    decomp.fillwith(0.0);
    decomp.subm(1, A.cols (), 1 + A.cols (), decomp.cols ()) ← transpose(A);
    decomp.subm(1 + A.cols (), decomp.rows (), 1, A.cols ()) ← A;
    for (index i ← 1; i ≤ A.cols (); ++i) decomp.entry(i, i) ← 1;
    lu::decompose(&decomp, &pivots);
}
```

402. Now we're ready to compute the direction. As said before, we may receive a preferred direction with the wrong dimensions. This happens when barrier functions are in use and a new slack variable is introduced in the problem after the $A$ matrix is defined. Therefore, when $x$ has the wrong dimension, we know it has exactly one more component than the original number, and what we do is to project only the first components.
403. (Equality search direction methods 399) +≡

\[
\text{matrix}(T, \text{unstructured}, \text{dense}) \& \text{dir}(\text{functional} :: \text{base}(T) \ast f, \text{const} \text{matrix}(T, \text{unstructured}, \\
\text{dense}) \& x, \text{matrix}(T, \text{unstructured}, \text{dense}) \ast dest)
\]

\[
\text{\{ double extra\_component } \leftarrow 0; \\
\text{ dest\_resize(decomp.rows(), 1); } \\
\text{ dest\_fillwith(0.0); } \\
\text{ dir\_dir(f, x, dest); }
\]

```
#ifndef __MATH_DEBUG__

cout \ll \"[math]::\_equality\_search\_dir\_begin.\n[math]\::\_orig\_dir\_\_\_\_\_\";

for (index i \leftarrow 1; i \leq x.rows(); ++i) cout \ll (\ast dest)(i) \ll \'\\n';

cout \ll \'\n\';

#endif

if (x.rows() \neq x.rows()) extra\_component \leftarrow (\ast dest)(xrows + 1);

dest\_entry(xrows + 1) \leftarrow 0.0;

lu\_\_finish(decomp, pivots, dest);

dest\_resize(x.rows(), 1);

if (x.rows() \neq xrows) dest\_entry(xrows + 1) \leftarrow extra\_component;
```

```
#ifndef __MATH_DEBUG__

cout \ll \"[math]::\_new\_dir\_\_\_\_\_\";

for (index i \leftarrow 1; i \leq result.rows(); ++i) cout \ll (\ast dest)(i) \ll \'\\n';

cout \ll \"\n[math]::\_equality\_search\_dir\_\_\_end.\n\";

#endif

return \ast dest;
```

404. Newton direction with equality constraints.

"Empty, waiting for export */

405. (searchdir/equality/newton.h 405) ≡

```
#ifndef __MATH_SEARCHDIR_EQUALITY_NEWTON__
#define __MATH_SEARCHDIR_EQUALITY_NEWTON__
#include <math/math.h>
#include <math/searchdirbase.h>
#include <math/lu.h>

namespace math { namespace searchdir { namespace equality { template <class T, template <class > class storage > class newton : public base(T) {
    \{ Newton with equality internal variables 406 \};

    public: \{ Newton with equality functions 408 \};
}; } } }
```

406. Let us be wise memorywise. Unlike with the pure equality search direction, we will have to restore the system matrix every time we compute a new direction. So that’s what we do: we store the \( A \) matrix literallyi. This matrix will not change once its initialized. Next, we keep the system matrix in a matrix \( M \).

\[
\text{matrix}(T, \text{unstructured, storage}) A; \\
\text{matrix}(T, \text{symmetric, storage}) H; \\
\text{matrix}(T, \text{unstructured, storage}) M;
\]

See also section 407.

This code is used in section 405.
Of course, we also face the same auxiliary variable problem.

\begin{verbatim}
407. (Newton with equality internal variables 406) +
  bool has_t;
  index xrows;  /* Original number of variables. */
\end{verbatim}
Newton direction with equality constraints

```cpp
408. \{ Newton with equality functions 408 \} ≡

matrix(T, unstructured, dense) & dir(func::base(T) * f, const matrix(T, unstructured,
  dense) & x, matrix(T, unstructured, dense) * dest)

  \{ if (x.rows() ≠ xrows ∧ has_t ≡ false) \{
    has_t ← true;
    A.resize(A.rows(), xrows + 1);
    for (index i ← 1; i ≤ A.rows(); ++i) A.entry(i, xrows + 1) ← 0;
  } if (x.rows() ≡ xrows ∧ has_t ≡ true) \{
    has_t ← false;
    A.resize(A.rows(), xrows);
  } try \{
    f-grad(x, *dest);
    f-hess(x, H);
    \#ifdef __MATH_DEBUG__
    cout ≡ "[math]: ⃗g=Hessian\n"
    for (index i ← 1; i ≤ H.rows(); ++i) \{
      cout ≡ "row\n" "i" ≡ "\n";
      for (index j ← 1; j ≤ H.cols(); ++j) cout ≡ H(i, j) ≡ '\n';
    }\n    \#endif
    dest-resize(x.rows() + A.rows(), 1);
    M.resize(x.rows() + A.rows(), x.rows() + A.rows());
    for (index i ← 1; i ≤ A.rows(); ++i) \{
      dest-entry(i + x.rows(), 1) ← 0;
      for (index j ← 1; j ≤ A.rows(); ++j) M.entry(i + x.rows(), j + x.rows()) ← 0;
      for (index j ← 1; j ≤ A.cols(); ++j) \{
        M.entry(i + x.rows(), j) ← A(i, j);
        M.entry(j, i + x.rows()) ← A(i, j);
      }\n    }\n    for (index i ← 1; i ≤ x.rows(); ++i)
      for (index j ← i; j ≤ x.rows(); ++j) \{
        M.entry(i, j) ← H(i, j);
        M.entry(j, i) ← H(j, i);
      }\n    \#ifdef __MATH_DEBUG__
    cout ≡ "[math]: about_to_get_search_direction.\n"
    for (index i ← 1; i ≤ M.rows(); ++i) \{
      for (index j ← 1; j ≤ M.cols(); ++j) cout ≡ M(i, j) ≡ '\n';
    }\n    cout ≡ '\n';
    cout ≡ 'dest=\n';
    for (index i ← 1; i ≤ dest.rows(); ++i) cout ≡ dest-get(i, 1) ≡ '\n';
    cout ≡ '\n';
    \#endif
    lu::solve(M, *dest);
```
\begin{verbatim}
dest-resize(x.rows(),1);
#endif _MATH_DEBUG_
    cout << "dest=\n";
    for (index i ← 1; i ≤ dest-rows(); ++i) cout << dest-get(i,1) << ' ';
    cout << 'n';
#endif
}

catch(error::singular e)
{
    f-grad(x,*dest);
}

return (*dest) *= -1;
\end{verbatim}

See also section 409.

This code is used in section 405.

\textbf{409.} Now we provide a function to set the $A$ matrix.

\language{MATH} \begin{verbatim}
\textbf{Newton with equality functions} 408 \equiv
void set_A(const matrix<T,unstructured,storage> &a)
{
    has_t ← false;
    xrows ← a.cols();
    A ← a;
}
\end{verbatim}
410. Optimization algorithms. We have a generic matrix class, we are able to perform some useful decompositions, we have a functional class, a line search class and a descent direction class. We are in position to build up a generic optimization package. The ultimate goal would be to efficiently solve the problem

\[
\begin{align*}
\text{minimize} & \quad f_0(x) \\
\text{subject to} & \quad f_i(x) < 0, \quad i = 1, \ldots, p \\
& \quad g_i(x) = 0, \quad i = 1, \ldots, q
\end{align*}
\]

for any type of convex constraints. The best way to accomplish this goal is by using barrier-function based optimization algorithms. These are functions that are infinite outside the feasible set of the constraint and finite inside, that is, \( \phi_i(x) = \infty \) if and only if \( f_i(x) \geq 0 \). With this property, we see that the minimum value of \( f_0(x) + \sum \phi_i(x) \) is bounded above if and only if the optimization problem is feasible. In what follows we build the base that will enable us to come up with a generic sequential unconstrained optimization routine. The basic idea is as follows, we compute the minimum of \( tf_0(x) + \sum \phi_i(x) \) for increasing values of \( t \). As this value goes to infinity we will approach the optimal solution of the original problem. The equality constraints will be satisfied if we use a search direction class that ensures this property – so we are able to ensure \( Ax = b \), for example.

411. Functional minimization. The first thing we need is to be able to minimize a functional without any constraints. Strictly speaking the correct thing to do would be to define a base class for functional minimization and specialize it depending on the stop criteria, but in the real world we face only two of them:

\[
\begin{align*}
|x_i - x^*_i| & \leq |x^*_i| \text{reltol} + \text{abstol} \\
\|\nabla f(x)\| & \leq \text{abstol}.
\end{align*}
\]

Of course, the first condition only makes sense when \( \text{abstol} \cdot \text{reltol} = 0 \). From that observation we decided to define only one functional minimization function that can take into account all of the above stopping criteria.

The function we’ll define will take various arguments: the first are the functional to be minimized, the starting point, the line search algorithm and the search direction algorithm. The starting point will be overwritten with the optimal point approximation and returned on exit. The next arguments define the stopping criteria: \( \text{abstol} \) and \( \text{reltol} \) have the meaning of that first stopping condition described above. They can be both positive, in which case the optimization stops when \( \text{both} \) the absolute error and the relative error conditions are met. If you don’t want one of them to have an effect, simply assign it zero. The \( gtol \) argument specifies, if nonzero, the stop condition for the gradient norm. At last, the user can input the maximum allowed number of iterations and a pointer to a function that gets the current piont at each iteration (possibly for displaying interactively).

\/* Empty, waiting for export. */
412. \langle fmin.h 412 \rangle \equiv

```c
#ifndef __MATH_FMIN__
#define __MATH_FMIN__ 1.0

#include <math/functionalbase.h>
#include <math/linesearchbase.h>
#include <math/searchdirbase.h>

namespace math {

    template<class T>
    matrix<T, unstructured, dense>& fmin(functional::base<T>* f, matrix<T, unstructured, dense>& x, linesearch::base<T>* lsearch, searchdir::base<T>* sdir, T abstol ← 1·10^−4, T rettol ← 1·10^−3, T gtol ← 1·10^−3, unsigned long maxiter ← 1000, void(*disp)(const matrix<T, unstructured, dense>&)) ← 0) {
        (Functional minimization algorithm 413)
    }
}
#endif
```

413. The algorithm is pretty simple: we keep calling the line minimization function until the stop criteria is met or the maximum number of iterations is reached (in which case we return the last line minimizer). The only trick in this function is that we only use the gradient for line minimization if \( gtol \) > 0. In this case, our function keeps the last function gradient in \( \text{grad} \) (which is otherwise not used).

\langle Functional minimization algorithm 413 \rangle \equiv

```c
matrix(T, unstructured, dense) x0, grad, dir;
if (gtol) f-grad(x, grad);
bool stop ← false;
for (unsigned long iter ← 0; ¬stop ∧ iter ≠ maxiter; ++iter) {
    x0 ← x;
    if (gtol) x ← lsearch-minimize(f, x, grad, sdir-dir(f, x, &dir));
    else x ← lsearch-minimize(f, x, sdir-dir(f, x, &dir));
    (Update functional minimization stop criteria 414)
    if (disp) disp(x);
} return x;
```

This code is used in section 412.

414. We test if all stopping criteria are satisfied. For the relative tolerance we have to check if it makes sense: if \( x_i = 0 \) no relative tolerance can be reached (in practice).

\langle Update functional minimization stop criteria 414 \rangle \equiv

```c
stop ← true;
if (abstol) for (index i ← 1; i ≤ x.rows() ∧ stop; ++i) stop ← (fabs(x(i) - x0(i)) ≤ abstol);
if (reltol) for (index i ← 1; i ≤ x.rows() ∧ stop; ++i)
    if (x(i)) stop ← (fabs(x(i) - x0(i)) ≤ fabs(x(i)) * rettol);
if (gtol) stop &=(norm2(f-grad(x, grad)) ≤ gtol);
```

This code is used in section 413.
415. **Barrier functions.** The next step necessary in order to achieve our goal is to define barrier functions. Barrier functions, as aligned before, are functionals that are bounded above if and only if the constraint to which they relate is feasible. In our convention, a constraint is feasible when its defining function is negative.

/ * Empty, waiting for export */

416. \{\texttt{barrierbase.h} 416\} ≡

```c++
#ifndef __MATH_BARRIER__
#define __MATH_BARRIER__ 1.0
#include <math/functionalbase.h>
#include <math/functionbase.h>
namespace math {
    namespace barrier {
        template< class T >
        class base : public functional::base< T >
        {
            // Barrier function internal variables 417
            public:
                // Barrier function methods 418
        }
    }
}
#endif
```

417. In order to make things easier for the programmer, we allow a barrier function to be related to a vector-valued function. We will store pointers to the function or functional, and the convention is that only one of them can be nonzero at any time.

\{Barrier function internal variables 417\} ≡

```c++
protected: functional::base< T > *fctnal;
function::base< T > *fct;
```

See also section 420.

This code is used in section 416.
418. ⟨Barrier function methods 418⟩ \equiv

\begin{verbatim}
base(functional::base(T) *newf ← 0)
  : fctnal(newf), fct(0), has.t(false) { }
base(function::base(T) *newf)
  : fctnal(0), fct(newf), has.t(false) { }
void f(functional::base(T) *newf)
  {
    fctnal ← newf;
    fct ← 0;
  }
void f(function::base(T) *newf)
  {
    fctnal ← 0;
    fct ← newf;
  }
virtual ~base(void) { }
\end{verbatim}

See also sections 421, 422, 423, and 424.

This code is used in section 416.

419. We don’t need to define the gradient and Hessian methods because we’re already derived from a class that derive them. The only thing we have to define is the behavior when dealing with vector-valued functions: in this case, the gradient is the sum of the gradient of the vector elements, and similarly for the Hessian and the evaluation functions. There’s another important thing that must be taken into account by evaluation methods.

420. We use barrier function in constrained optimization problems. Some algorithms don’t require an initial feasible point, that is, we don’t need an \(x\) such that \(f(x) < 0\). One standard way to find a feasible point is to add a variable to the constraint so that \(f(x) < t\) is feasible, and then minimize \(t\). These methods will then call the \(addt\) function of the barrier functions to signal that we should behave as if this variable existed. From then on, until a call for \(delt\), the last component of the argument for \(eval\) and so on is considered to be \(t\), and the evaluation methods must deal with it accordingly.

⟨Barrier function internal variables 417⟩ +≡

\begin{verbatim}
bool has.t;
\end{verbatim}

421. The \(addt\) function returns a value of \(t\) that satisfies \(f(x) \leq t\).

⟨Barrier function methods 418⟩ +≡

\begin{verbatim}
T addt(const matrix(T, unstructured, dense) &x)
{
  T result ← T(0);
  if (fctnal) result ← fctnal-eval(x);
  else {
    matrix(T, unstructured, dense) aux;
    fct-eval(x, aux);
    for (index i ← 1; i ≤ aux.rows(); ++i) result += aux(i);
  }
  has.t ← true;
  return result;
}\end{verbatim}
422. Barrier function methods +

\[
\text{virtual } T \text{ dual} \text{(const matrix}(T, \text{unstructured}, \text{dense}) \text{ &)} \leftarrow 0;
\]

\[
\text{virtual } T \text{ dual times f} \text{(const matrix}(T, \text{unstructured}, \text{dense}) \text{ &)} \leftarrow 0;
\]

\[
\text{template<class T> classlog:public base}(T)
\]

\[
\text{classlog}
\]

\[
\text{public:}
\]

\[
\langle \text{Log barrier methods 427} \rangle
\]

423. Barrier functions are used in optimization problems in order to deal with constraints. To each constraint we have an associated dual variable. This is what happens: we will want to solve the problem

\[
\begin{align*}
\text{minimize} & \quad f_0(x) \\
\text{subject to} & \quad f_i(x) < 0
\end{align*}
\]

and we will sequentially solve

\[
\begin{align*}
\text{minimize} & \quad \mu f_0(x) + \sum \phi(f_i(x))
\end{align*}
\]

for various values of \( \mu \). For each of these values, the solution of the second problem will also satisfy \( \nabla f_0(x) + \sum \lambda_i \nabla f_i(x) = 0 \) for some \( \lambda_i \) that depends on the particular barrier function used. But if \( x^* \) satisfies that equality, it is a minimizer for the Lagrangian with the same \( \lambda_i \), that is, \( \lambda_i \) are feasible dual variables and can provide a lower bound on the optimal value. It is clear that the \( \lambda_i \) will be dependent on \( \mu \), but the barrier function doesn’t know its value. What we do then is to return \( \mu \lambda_i \), which is independent of \( \mu \).

424. On other ocasions it will be useful to get the value of the dual variable times the function itself, and it is sometimes much cheaper to compute this value (see, for example, the log barrier function). The same above remarks with respect to \( \mu \) are valid here, that is, the function should return \( \mu \lambda_i f(x) \).

425. The log barrier function. Perhaps the most common, its value is \( -\log(-f(x)) \).

\[
\text{#ifndef } \_\_MATH_LOG_BARRIER\_\_
\]

\[
\text{#define } \_\_MATH_LOG_BARRIER\_\_
\]

\[
\text{#include <math/barrierbase.h>}
\]

\[
\text{namespace math} \{
\]

\[
\text{template<class T> classlog:public base}(T)
\]

\[
\text{public:}
\]

\[
\langle \text{Log barrier methods 427} \rangle
\]

426. barrier/log.h 426

\[
\#ifndef \_\_MATH_LOG_BARRIER\_\_
\]

\[
\#define \_\_MATH_LOG_BARRIER\_\_
\]

\[
\#include <math/barrierbase.h>
\]

\[
\text{namespace math} \{
\]

\[
\text{template<class T> classlog:public base}(T)
\]

\[
\text{public:}
\]

\[
\langle \text{Log barrier methods 427} \rangle
\]

427. Barrier function methods +

\[
\text{virtual } T \text{ dual} \text{(const matrix}(T, \text{unstructured}, \text{dense}) \text{ &)} \leftarrow 0;
\]

\[
\text{virtual } T \text{ dual times f} \text{(const matrix}(T, \text{unstructured}, \text{dense}) \text{ &)} \leftarrow 0;
\]

\[
\langle \text{Log barrier methods 427} \rangle
\]

428. Barrier function methods +

\[
\text{virtual } T \text{ dual} \text{(const matrix}(T, \text{unstructured}, \text{dense}) \text{ &)} \leftarrow 0;
\]

\[
\text{virtual } T \text{ dual times f} \text{(const matrix}(T, \text{unstructured}, \text{dense}) \text{ &)} \leftarrow 0;
\]

\[
\langle \text{Log barrier methods 427} \rangle
\]

429. Barrier function methods +
427. ⟨Log barrier methods 427⟩ ≡
\[
\begin{align*}
\log\text{(functional :: base(T) *newf)} & \{ f(newf); \} \\
\log\text{(function :: base(T) *newf)} & \{ f(newf); \}
\end{align*}
\]
See also sections 428, 429, 430, and 431.
This code is used in section 426.

428. Remember that we have to take into account that an auxiliary variable may be present. In that case, we first get the value of \( t \), resize \( x \) and get the value and the gradient of the function. Resizing is not a big burden on performance because its optimized so that normally no memory allocation is necessary.

\[
\begin{align*}
\text{T eval(const matrix(T, unstructured, dense) \&x)} & \\
\{ & \\
\text{T result ← 0, t ← 0;} \\
\text{matrix(T, unstructured, dense) X ← x;} & \quad /* We may resize x. */ \\
\text{if (has_t) } & \\
\text{t ← X(X.rows());} \\
\text{X.resize(X.rows() - 1, 1);} & \\
\} \\
\text{if (fctnal) } & \\
\text{result ← fctnal-eval(X) - t;} \\
\text{if (result ≥ 0) throw error :: domain();} \\
\text{return −::log(−result);} & \\
\} \\
fct-eval(X, aux); \\
\text{for (index i ← 1; i ≤ aux.rows(); ++i) } & \\
\text{if (aux(i) - t ≥ 0) throw error :: domain();} \\
\text{result ← ::log(t - aux(i));} & \\
\} \\
\text{return result;} & \\
\}
\]

429. For the logarithm, the dual variable is \(-1/(\mu f(x))\), so we return \(-1/f(x)\).

\[
\begin{align*}
\text{T dual(const matrix(T, unstructured, dense) \&x)} & \{ \text{return } -1/\text{eval(x)}; \} \\
\text{T dual_times_f(const matrix(T, unstructured, dense) \&)} & \{ \text{return T(−1.0); } \}
\]
430. Now to the gradient: for a functional $f$ we have $\nabla - \log(-f) = -(\nabla f)/f$. By our definition for a function we have the sum $-\sum_i (\nabla f_i)/f_i$.

\begin{verbatim}
matrix(T, unstructured, dense) & grad(const matrix(T, unstructured, dense)
                           & x, matrix(T, unstructured, dense) & dest)
{
  T t ← 0;
  matrix(T, unstructured, dense) X ← x;  /* We may resize x. */
  if (has_t) {
    t ← X(X.rows());
    X.resize(X.rows() - 1, 1);
  }
  if (fctnal) {
    fctnal-grad(X, dest);
    if (has_t) {
      dest.resize(X.rows() + 1, 1);
      dest.entry(dest.rows()) ← T(-1);
    }
    dest / t = fctnal-eval(X);
    return dest;
  }
  fct-eval(X, aux);
  fct-jacobian(X, dest);
  if (has_t) {
    dest.resize(dest.rows(), dest.cols() + 1);
    dest.subm(1, dest.rows(), dest.cols()) ← matrix(T, unstructured, dense)(dest.rows(), 1, T(-1));
  }
  for (index i ← 1; i ≠ aux.rows(); ++i) {
    dest.subm(i, i, 1, dest.cols()) / t = aux(i);
    if (i ≠ 1) dest.subm(1, 1, dest.cols()) += dest.subm(i, i, 1, dest.cols());
  }
  dest ← transpose(dest.subm(1, 1, 1, dest.cols()));
  return dest;
}
\end{verbatim}

431. Finally the Hessian: for a functional we have $\nabla^2 - \log(-f) = (\nabla f \nabla^T f)/f^2 - (\nabla f)^2/f$.

\begin{verbatim}
matrix(T, symmetric, dense) & hess(const matrix(T, unstructured, dense)
                           & x, matrix(T, symmetric, dense) & dest)
{
  #define hess_aux dense_aux
  \{ Compute Hessian for log barrier 432; \}
  #undef hess_aux

  matrix(T, symmetric, sparse) & hess(const matrix(T, unstructured, dense)
                           & x, matrix(T, symmetric, sparse) & dest)
  {
    #define hess_aux sparse_aux
    \{ Compute Hessian for log barrier 432; \}
    #undef hess_aux
  }
\end{verbatim}
432. \{ Compute Hessian for log barrier \}

\textbf{Math Library (Version alpha 3-22-2002)}

\[ 432. \langle \text{Compute Hessian for log barrier } 432 \rangle \equiv \]

\begin{verbatim}
T t ← 0;
matrix(T, unstructured, dense) X ← x;  /\* We may resize x. */

if (has_t) {
  t ← X(X.rows());
  X.resize(X.rows() - 1, 1);
}
if (fctnal) {
  T fval ← fctnal-eval(X) - t;
  fctnal-grad(X, aux);
  fctnal-hess(X, dest);
  if (has_t) {
    aux.resize(aux.rows() + 1, 1);
    dest.resize(dest.rows() + 1, dest.rows() + 1);
    aux.entry(aux.rows()) ← T(-1);
    for (index i ← 1; i ≤ dest.cols(); ++i) {
      dest.entry(dest.rows(), i) ← 0;
      dest.entry(i, dest.cols()) ← 0;
    }
  }
  aux /= fval;
  dest /= -fval;
  dest += outerp(aux, &hess_aux);
  return dest;
}

dest.init(X.rows() + (has_t ? 1 : 0), X.rows() + (has_t ? 1 : 0));
dest.fillwith(0);

fct-eval(X, aux);

fct-jacobian(X, jac);
if (has_t) {
  jac.resize(jac.rows(), jac.cols() + 1);
  jac.subm(1, jac.rows(), jac.cols()) ← matrix(T, unstructured, dense)(jac.rows(), 1, T(-1));
}

for (index i ← 1; i ≤ aux.rows(); ++i) {
  jac.subm(i, i, 1, jac.cols()) /= aux(i) - t;
  fct-hess(X, hess_aux, i);
  if (has_t) dest.subm(1, dest.rows() - 1, 1, dest.cols() - 1) ← (hess_aux /= t - aux(i));
  else dest ← (hess_aux /= aux(i));
  dest += outerp(jac.subm(i, i, 1, jac.cols()), &hess_aux);
}
return dest;
\end{verbatim}

This code is used in section 431.
433. Sequential unconstrained minimization. function defined, we are ready to tackle our main problem, namely

\[
\begin{align*}
\text{minimize} & \quad f_0(x) \\
\text{subject to} & \quad f_i(x) < 0, \quad i = 1, \ldots, p \\
& \quad g_i(x) = 0, \quad i = 1, \ldots, q.
\end{align*}
\]

The method used to solve this problem is to minimize, for increasing values of \(t\), the functional

\[
\mu f_0(x) + \sum \phi_i(x),
\]

where \(\phi_i\) are barrier functions related to the inequality constraints. The equality constraints are taken care by the search direction function — the method doesn’t do anything to enforce them. The algorithms that work this way are called Sequential Unconstrained Minimization Techniques, or SUMT for brevity.

/* Empty, waiting for export */

434. \langle sumt.h 434\rangle ≡

```cpp
#ifndef __MATH_SUMT__
#define __MATH_SUMT__
#include <vector>
#include <math/fmin.h>
#include <math/functional/linear.h>
#include <math/functional/gaxpy.h>
#include <math/barrier/log.h>
#include <math/linesearchbase.h>
namespace math {
    ⟨SUMT functions 435⟩;
}
#endif
```
435. The arguments for the function are as follows: first we have the objective functional and a list of barrier functions. If the objective functional is zero, a feasibility problem will be solved. We have to enforce that the list is actually of barrier functions because we’ll need information about dual variables in order to determine that our solution is precise enough, to detect infeasibility and so on. Next, the user provides an initial point, and line minimization and search direction algorithms. The last parameters are the absolute and relative precisions, followed by a pointer to a function that gets as arguments the current phase, the value of the objective function (primal) and the dual slack.

\[ \text{SUMT functions} \quad 435 \equiv \]

```cpp
template<class T>
void sumt (functional::base<T> *obj, vector<barrier::base<T> *> &barrier_functions, matrix<T, unstructured, dense> &x, linesearch::base<T> *line, searchdir::base<T> *dir, T abstol, T reltol, void(*disp)(int, double, double) ← 0 )
{
    #ifdef __MATH_DEBUG__
        cout ≡ "[math]::sumt::begins.\n";
    #endif
    T mu;
    functional::linear<T> f;
    typedef typename vector<barrier::base<T> *>::iterator sumt_iterator;
    (Build unconstrained objective function 436);
    bool initial_point_is_feasible ← true;
    for (sumt_iterator i ← barrier_functions.begin(); i ≠ barrier_functions.end() ∧ initial_point_is_feasible; ++i)
    {
        try {
            (*i)-eval(x);
        }
        catch(error::domain)
        {
            initial_point_is_feasible ← false;
        }
        if (¬initial_point_is_feasible) {
            (SUMT Phase one 438);
        }
        (SUMT Phase two 441);
    #ifdef __MATH_DEBUG__
        cout ≡ "[math]::sumt::ends.\n";
    #endif
    }
}
```

This code is used in section 434.

436. Before doing anything we build the functional we’ll minimize. We use the linear functional, which has provision for weighting.

\[ \text{Build unconstrained objective function} \quad 436 \equiv \]

```cpp
f.add(obj);
for (sumt_iterator i ← barrier_functions.begin(); i ≠ barrier_functions.end(); ++i) f.add(*i);
```

This code is used in section 435.
437. At this point we know something is wrong with the initial point. The problem is obviously not feasible here, but there can be two reasons: one is that one of the constraints is positive, so the barrier function throws a domain error. But it can happen that the functional called by the barrier function throws a domain error too. Consider that the constraint is $\sqrt{x} > 1$ and that the initial point is $-1$. A log barrier function would call $\sqrt{-1}$, which would throw a domain error. The first type of infeasibility is easy to deal with through a Phase I approach. The other is hard in this phase, so we assume the user will provide good enough an initial point. If not, the next piece of code will end up throwing domain, which will flag the problem.

438. Ok. Now that we don’t have a feasible point, we modify the problem so it becomes feasible. The standard way to do that is to modify all constraints of the form $f(x) < 0$ to $f(x) < t$, where $t$ is an auxiliary variable. The addt function of barrier functions update the constraint and returns the minimum value of $t$ necessary in order for the constraint to be feasible with the $x$ provided. At the end of the loop we know that aux_var will be positive (the problem was infeasible), so we can multiply it by two in order to obtain an interior point.

\[ \langle \text{SUMT Phase one 438} \rangle \equiv \]
\[
T \ aux\_var \leftarrow 0; \\
\text{for (sumt\_iterator } i \leftarrow \text{barrier\_functions.begin}(); i \neq \text{barrier\_functions.end}(); ++i) \\
\quad aux\_var \leftarrow \max (aux\_var, (i)\text{-addt}(x)); \\
\text{aux\_var \leftarrow aux\_var + aux\_var};
\]

See also sections 439 and 440.

This code is used in section 435.

439. By now we have an interior feasible point for the augmented system. What we have to do is to build the initial point vector accordingly and to build the unconstrained function to be minimized.

\[ \langle \text{SUMT Phase one 438} \rangle \equiv \]
\[
x\text{.resize}(x\text{.rows()} + 1, 1); \\
x\text{.entry}(x\text{.rows}()) \leftarrow aux\_var; \\
\text{matrix}(T, \text{unstructured}, \text{dense}) \ phasei\_cost(x\text{.rows}(), 1); \\
\text{phasei\_cost}\text{.entry}(x\text{.rows}()) \leftarrow 1.0; \\
\text{functional}::\text{gaxpy}(T, \text{unstructured}, \text{dense}) \ phasei(phasei\_cost, 0.0); \\
f\text{.set\_term}(0, &phasei);
\]
440. We now try to find a feasible point for the original problem. We don’t have to go all the way, just enough so that the final value of $t$ is not too close to zero, otherwise the initial point for phase II will be barely feasible and numerical problems could arise. If we find the global optimum and $t \geq 0$, then the problem is infeasible. We also compute the Lagrange dual function in order to test for infeasibility. The value of the Lagrange dual function depends on the barrier functions being used.

\[
\langle \text{SUMT Phase one 438} \rangle \equiv
\]

\[
\begin{align*}
mu & \leftarrow 1; \\
T & (\text{cost} \leftarrow 2 * \text{abstol};
\text{T lagrange} \leftarrow 0; \text{while} \ (\text{cost} - \text{lagrange} \geq \text{abstol} \land x(x.\text{rows}()) > -0.1) \ {\{ f.\text{set_weight}(0, \text{mu}); \text{try} \ {\{ \\
x \leftarrow \text{fmin} (&f, x, \text{line} , \text{dir} , \text{abstol}, \text{reltol}, 0.0 ) ; \}} \text{catch} (\text{error}::\text{maxiterations} \ e) \}
\}}
\}
\text{lagrange} \leftarrow \text{cost}; \\
\text{for} (\text{int} \ i \leftarrow 1; i \neq f.\text{size}(); ++i)
\text{lagrange} +\equiv \text{static_cast} (\text{barrier} :: \text{base}(\text{T}) \ast)(f.\text{get_term}(i)) \ast \text{dual_times}_f(x) / \mu;
\text{if} (\text{disp}) \text{disp}(1, \text{cost}, \text{cost} - \text{lagrange}); \\
\text{if} (\text{lagrange} \geq 0) \text{throw} \text{error}::\text{infeasible}(); \\
\text{mu} \leftarrow \text{mu} \ast 50; \}
\}
\text{if} (x(x.\text{rows}()) \geq 0) \text{throw} \text{error}::\text{infeasible}();
\text{x.resize(x.\text{rows}())}; \\
\text{for} (\text{sumt_iterator} \ i \leftarrow \text{barrier}\_\text{functions}.\text{begin}(); i \neq \text{barrier}\_\text{functions}.\text{end}(); ++i) \ (i)\text{-}\text{delt}();
\]

441. Ready we are for phase II. We have to restore the original cost function that was overwritten in phase I. After that, we proceed almost exactly as in phase I.

\[
\langle \text{SUMT Phase two 441} \rangle \equiv
\]

\[
\begin{align*}
\text{if} (\neg \text{obj}) \text{return}; \quad /\ast \text{It was a feasibility problem.} \ast/ \\
\text{f.\text{set_term}(0, \text{obj});} \\
\text{bool} \text{stop} \leftarrow \text{false}; \\
\text{mu} \leftarrow 1; \text{while} (\neg \text{stop}) \ {\{ f.\text{set_weight}(0, \text{mu}); x \leftarrow \text{fmin} (&f, x, \text{line} , \text{dir} , \text{abstol}, \text{reltol}, 0.0 ) ; 
\}}
\}
\text{T cost} \leftarrow \text{obj}\_\text{eval}(x); \\
\text{T slack} \leftarrow 0; \\
\text{for} (\text{int} \ i \leftarrow 1; i \neq f.\text{size}(); ++i)
\text{slack} \equiv \text{static_cast} (\text{barrier} :: \text{base}(\text{T}) \ast)(f.\text{get_term}(i)) \ast \text{dual_times}_f(x) / \mu;
\text{if} (\text{disp}) \text{disp}(2, \text{cost}, \text{slack}); \\
\text{mu} \leftarrow \text{mu} \ast 50; \\
\text{stop} \leftarrow \text{true}; \\
\text{#ifdef} \ _\text{MATH}\_\text{DEBUG} \\
\text{cout} \equiv "\text{[math]}\:\"; \\
\text{cout} \equiv "\text{slack}" \equiv \text{slack} \equiv ",\text{cost}" \equiv \text{cost} \equiv ",\text{abstol}" \equiv \text{abstol} \equiv ",\text{reltol}" \equiv \text{reltol} \equiv 
\text{\"\n\"}; \\
\text{#endif} \\
\text{if} (\text{abstol}) \text{stop} \leftarrow (\text{slack} \leq \text{abstol}); \\
\text{if} (\text{reltol} \land \text{cost} \land \text{stop}) \text{stop} \leftarrow (\text{fabs} (\text{slack}/\text{cost}) \leq \text{reltol});
\}
\]

This code is used in section 435.
442. Index. Here is a cross-reference table for MATH. Underlined entries correspond to where the identifier was declared.

- _MATH_...: 3
- _MATH_ALGEBRA_...: 118
- _MATH_BACKTRACKING_LINESearch_...: 381
- _MATH_BARRIER_...: 416
- _MATH_BISECTION_...: 374
- _MATH_CHOLESKY_...: 211
- _MATH_DEBUG...: 441
- _MATH_DEBUG_...: 384, 385, 401, 403, 408, 435
- _MATH_DET_...: 243
- _MATH_ENTRopY_FUNCTIONAL_...: 324
- _MATH_ERF_FUNCTIONAL_...: 343
- _MATH_EXP_FUNCTIONAL_...: 353
- _MATH_EYE_...: 238
- _MATH_FM...: 412
- _MATH_FN_...: 258
- _MATH_FN...: 154
- _MATH_FUNCTION_...: 245
- _MATH_GAXPY_FUNCTION_...: 251
- _MATH_GAXPY_FUNCTIONAL_...: 264
- _MATH_INSANE_DEBUG_...: 24, 26, 265, 36, 53
- _MATH_LINEAR_FUNCTIONAL_...: 271
- _MATH_LI...: 371
- _MATH_LOG_BARRIER_...: 426
- _MATH_LOG_FUNCTIONAL_...: 362
- _MATH_LU_...: 197
- _MATH_NORM2ERR_FUNCTIONAL_...: 290
- _MATH_ONES_...: 240
- _MATH_POWER_FUNCTIONAL_...: 315
- _MATH_PRD_FUNCTIONAL_...: 298
- _MATH_QR_...: 225
- _MATH_QUADRATIC_FUNCTIONAL_...: 283
- _MATH_RATIO_FUNCTIONAL_...: 301
- _MATH_RELATIVE_ENTRopY_FUNCTIONAL_...: 307
- _MATH_SEARCHDIR_EQUALITY_...: 333
- _MATH_SEARCHDIR_EQUALITY_NEWTON_...: 405
- _MATH_SEARCHDIR_GRADIENT_...: 390
- _MATH_SEARCHDIR_NEWTON_...: 393
- _MATH_SEARCHDIRBASE_...: 387
- _MATH_SPARSE_...: 136
- _MATH_SUM_...: 434
- _MATH_SYMMETRIC_...: 147
- a: 122, 253, 266, 382, 409
- Ablock: 228
- abs: 197, 200
- abstol: 412, 414, 435, 440, 441
- add: 273, 436
- addt: 420, 421, 438
- alpha: 381, 382, 384
- assign: 165
- At: 233, 234, 235
- atzmul: 128, 228, 231, 235, 294
- aux_var: 438, 439
- auxdense: 339, 340
- auxgrad: 339, 341
- auxhess: 340, 341
- auxsparse: 339, 340
- auxvec: 337, 338, 341
- axmul: 127, 286
- ay: 252, 253, 254, 255, 265, 266, 267, 268
- B: 202, 203, 215, 216, 229, 253, 266
- b: 253, 266, 382
- backtracking: 381, 382
- backtracking::minimize: 383, 384
- backtracking::infty::minimize: 381, 385
- backup: 80
- barrier: 416, 426, 435, 440, 441
- barrier::log::grad: 430
- barrier::log::hess: 431
- barrier::addt: 438
- barrier::delt: 440
- barrier_functions: 435, 436, 438, 440
- Bblock: 231, 235
- bee: 252, 253, 265, 266, 267
- begin: 140, 205, 209, 276, 277, 279, 281, 435, 436, 438, 440
- beta: 134, 226, 227, 228, 381, 382, 384, 385
- binary: 158, 159, 183, 184
- bisection: 374, 375, 378
- bisection::minimize: 376, 377
- bisection::infty::minimize: 374, 379
- c_str: 180, 191
- Char: 161, 168, 188, 192
- character: 219
- cholesky: 211, 395
- col: 58, 59, 61, 62, 63, 64, 69, 71, 91, 142, 143, 150
exponent: 316, 317, 318, 320, 322.
extra_component: 403.
eye: 237, 248.
F: 291.
fabs: 378, 414, 441.
false: 41, 59, 64, 112, 150, 408, 409, 413, 418, 422, 435, 441.
fast_assignment: 40, 41, 42, 43.
fct: 417, 418, 421, 428, 430, 432.
fctnml: 417, 418, 421, 428, 430, 432.
fd: 157, 182.
file_flag: 165.
filerr: 9, 167, 170.
fillwith: 34, 35, 36, 256, 269, 279, 281, 322, 401, 403, 432.
find: 142, 143.
first: 112, 275, 277, 279, 281.
fmin: 412, 440, 441.
fortran: 205, 206, 209, 218, 219, 221, 222, 223.
ftreshold: 384, 385.
function::base::grad: 248.
function::base::hess: 249.
function::gaxpy::A: 253.
function::gaxpy::b: 253.
function::gaxpy::eval: 254.
function::gaxpy::hess: 256.
function::gaxpy::jacobian: 255.
function::eval: 293, 294, 295, 296, 428, 430, 432.
function::hess: 295, 296, 432.
function::jacobian: 294, 295, 296, 430, 432.
functional::base::grad: 261.
functional::base::hess: 262.
functional::gaxpy::A: 266.
functional::gaxpy::b: 266.
functional::gaxpy::grad: 268.
functional::gaxpy::hess: 269.
functional::norm2err::eval: 293.
functional::norm2err::grad: 294.
functional::norm2err::hess: 295, 296.
linesearch: 371, 374, 381, 412, 435.
linesearch::bisection::minimize: 376.
linesearch::minimize: 413.
Long: 161, 188.
lssearch: 412, 413.
lu: 197, 243, 395, 401, 403, 408.
lu::decompose: 401.
lu::finish: 403.
M: 44, 81, 406.
m: 127, 128, 134, 206, 219, 229, 382.
_\_PI: 348, 351.
map: 110, 138.
mat: 69, 85, 86, 89, 90.
matrix::(): 62.
matrix::--: 42, 44.
matrix::*=: 279, 281, 385.
matrix::*+: 385.
matrix::/=: 430.
matrix::=: 385.
matrix::copyfrom: 46.
matrix::detach: 57, 58, 73, 74, 76.
matrix::entry: 71, 255, 401, 430.
matrix::fillwith: 34, 36, 256, 279, 281, 401, 403, 432.
matrix::init: 38, 44, 76, 89, 256, 432.
norm2: 126.
nonsquare: 9.
nonpositive_def: 9.
newy: 285.
newP: 292.
norm2: 126, 378, 414.
norm2err: 290, 292.
notimplemented: 9.
num_cols: 17, 18, 26, 28, 29, 30, 33, 38, 39, 47, 48, 52, 77, 80.
num_instances: 21, 22, 26, 47, 54.
num_matrices: 172, 173.
num_rows: 17, 18, 26, 28, 29, 30, 33, 38, 39, 47, 48, 52, 77, 80.
number: 167, 168, 170, 190, 192, 194.
nunnonzeros: 115.
obj: 435, 436, 441.
ofstream: 181, 182, 183, 184, 185, 186, 187, 188, 189, 193.
ofstream::open: 184.
one: 239, 240.
open: 158, 159, 183, 184.
out: 183.
outer_iter: 378.
outerp: 131, 313, 322, 331, 341, 351, 360, 369, 432.
outerp_update: 134, 228, 231, 235.
P: 285.
permutations: 199, 200.
phase: 439.
phase_cost: 439.
pi: 285.
P: 284, 285, 286.
pivot: 199, 200.
pop_back: 274.
position: 176, 177, 179, 180.
pow: 318, 320, 322.
power: 315, 317.
preprocess: 59, 60, 63, 64, 150.
prod: 298, 300.
prot: 158, 159, 183, 184.
push_back: 273.
pub: 188, 191, 192.
qr: 225.
quadratic: 283, 285.
r: 375.
rank_deficient: 9, 232, 234.
real: 194, 209, 221, 223.
size: 191, 273, 275, 279, 281, 440, 441.
skip: 172, 173, 177.
skip_data: 172, 173, 174, 175, 180.
skipto: 176, 177, 178, 179, 185.
slack: 441.
solve: 202, 216, 229, 395, 408.
sparse: 143.
sparse: 144.
sparse: 142.
sparse: 143.
sparse: 144.
sparse: 142.
step: 373.
STD: 44.
STOA: 44, 45.
stop: 413, 414, 441.
storage: 59, 63.
storage: 77.
storage: 16, 73.
storage: 72, 73, 116, 205, 209, 218, 221, 222, 223.
STR: 44.
STRA: 44, 45.
strcmp: 180.
string: 4, 5, 8, 162, 181.
structure: 59, 63.
structure: 77.
structure: 215, 216.
structure: 215, 216.
structure: 16.
〈Algebraic operations 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134〉 Used in section 118.
〈Apply Cholesky transformation 214〉 Used in section 212.
〈Apply Gauss transformation 201〉 Used in section 199.
〈Apply Householder transformation 228〉 Used in section 226.
〈Backtracking line search methods 382, 383, 384〉 Used in section 381.
〈Barrier function internal variables 417, 420〉 Used in section 416.
〈Barrier function methods 418, 421, 422, 423, 424〉 Used in section 416.
〈Basic algebraic operations 105〉 Used in section 3.
〈Basic definitions 4, 5, 40, 82〉 Used in section 3.
〈Big definitions 112〉 Used in section 2.
〈Bisection line search methods 375, 376, 377〉 Used in section 374.
〈Build L^T row 213〉 Used in section 212.
〈Build unconstrained objective function 436〉 Used in section 345.
〈Check for lapack LU errors 207〉 Used in sections 205 and 209.
〈Check lapack Cholesky errors 220〉 Used in sections 218, 221, 222, and 223.
〈Cholesky lapack interface 218, 221, 222, 223〉 Used in section 217.
〈Cholesky prototypes 212, 215, 216, 217〉 Used in section 211.
〈Compute Hessian for log barrier 432〉 Used in section 431.
〈Compute Hessian for erf functional 351〉 Used in section 350.
〈Compute Householder vector for column j 227〉 Used in section 226.
〈Compute next function value for backtracking 385〉 Used in section 384.
〈Compute next function value for bisection 379〉 Used in section 378.
〈Compute number of permutations and store it in status 208〉 Used in sections 205 and 209.
〈Dense storage internal variables 28〉 Used in section 10.
〈Dense storage methods 29, 30, 32, 48, 52, 55, 56, 61, 64, 75, 79〉 Used in section 10.
〈Element definition 65〉 Used in section 3.
〈Entropy of a functional class methods 326, 327, 329, 331〉 Used in section 324.
〈Entropy of a functional internal variables 325, 330〉 Used in section 324.
〈Equality search direction internal variables 398, 400〉 Used in section 397.
〈Equality search direction methods 399, 401, 403〉 Used in section 397.
〈Error function of a functional class methods 345, 346, 348, 350〉 Used in section 343.
〈Error function of a functional internal variables 344, 349〉 Used in section 343.
〈Exponential of a functional class methods 355, 356, 358, 360〉 Used in section 353.
〈Exponential of a functional internal variables 354, 359〉 Used in section 353.
〈Function base class methods 246, 247, 248, 249〉 Used in section 245.
〈Functional base class methods 259, 260, 261, 262〉 Used in section 258.
〈Functional minimization algorithm 413〉 Used in section 412.
〈Gaxpy function class methods 253, 254, 255, 256〉 Used in section 251.
〈Gaxpy function internal variables 252〉 Used in section 251.
〈Gaxpy functional class methods 266, 267, 268, 269〉 Used in section 264.
〈Gaxpy functional internal variables 265〉 Used in section 264.
〈Generic error class methods 7, 8〉 Used in section 5.
〈Get number from file, using the same tricky method as for longs 168〉 Used in sections 167 and 170.
〈Gradient search direction methods 391〉 Used in section 390.
〈Include files fstream 156, 163, 171〉 Used in section 154.
〈Include files math 6, 31, 113, 114〉 Used in section 3.
〈Include files sparse 139〉 Used in section 136.
〈LU lapack interface 205, 209〉 Used in section 204.
〈LU prototypes 199, 202, 203, 204〉 Used in section 197.
〈Line search base class methods 372〉 Used in section 371.
〈Linear combination of functionals class methods 273, 274, 275, 276, 277, 279, 281〉 Used in section 271.
〈Linear combination of functionals internal variables 272, 278, 280〉 Used in section 271.
Log barrier methods 427, 428, 429, 430, 431  Used in section 426.
Logarithm of a functional internal variables 363, 368  Used in section 362.
Matrix definition 12  Used in section 3.
Matrix element internal variables 66, 67  Used in section 65.
Matrix element methods 68, 69, 70, 95, 97, 100, 103, 107  Used in section 65.
Matrix internal types 14, 16, 88  Used in section 12.
Matrix internal variables 15  Used in section 12.
Matrix representation definition 13  Used in section 12.
Matrix representation internal variables 17, 20, 21, 50  Used in section 13.
Matrix representation methods 18, 22, 25, 26, 47, 51, 54, 59, 63, 72, 77  Used in section 13.
Newton search direction internal variables 394  Used in section 393.
Newton search direction methods 395  Used in section 393.
Newton with equality functions 408, 409  Used in section 405.
Newton with equality internal variables 406, 407  Used in section 405.
Norm-2 error functional class methods 292, 293, 294, 295, 296  Used in section 290.
Norm-2 error functional internal variables 291  Used in section 290.
Power of a functional class methods 317, 318, 320, 322  Used in section 315.
Power of a functional internal variables 316, 321  Used in section 315.
Predefined error types 9  Used in section 5.
Prepare for lapack Cholesky 219  Used in sections 218, 221, 222, and 223.
Prepare for lapack LU 206  Used in sections 205 and 209.
Product of functionals class methods 300, 301, 303, 305  Used in section 298.
Product of functionals internal variables 299, 302, 304  Used in section 298.
QR prototypes 226, 229  Used in section 225.
Quadratic functional internal variables 284  Used in section 283.
Quadratic functional methods 285, 286, 287, 288  Used in section 283.
Ratio of functionals class methods 309, 311, 313  Used in section 307.
Ratio of functionals internal variables 308, 310, 312  Used in section 307.
Relative entropy functional class methods 335, 336, 338, 340  Used in section 333.
Relative entropy functional internal variables 334, 337, 339  Used in section 333.
SUMT Phase one 438, 439, 440  Used in section 435.
SUMT Phase two 441  Used in section 435.
SUMT functions 435  Used in section 434.
Search direction base class methods 388  Used in section 387.
Search for correct matrix name 180  Used in section 179.
Search for pivot and swap rows if necessary 200  Used in section 199.
Solve $Q^T x = y$ 235  Used in section 233.
Solve $Q y = b$ 231  Used in section 230.
Solve $R^T y = b$ 234  Used in section 233.
Solve $R x = y$ 232  Used in section 230.
Solve least squares problem 230  Used in section 229.
Solve minimum norm problem 233  Used in section 229.
Sparse storage definition 137  Used in section 136.
Sparse storage internal variables 138  Used in section 137.
Sparse storage methods 140, 141, 142, 143, 144  Used in section 137.
Specializations 106, 109, 110, 111, 116  Used in section 3.
Storage definition 10  Used in section 3.
Structure definition 11  Used in section 3.
Submatrix definition 81  Used in section 3.
\{Submatrix internal variables 83, 84\} Used in section 81.
\{Submatrix methods 85, 86, 87, 90, 91, 93, 98, 101, 104\} Used in section 81.
\{Symmetric structure definition 148\} Used in section 147.
\{Symmetric structure methods 149, 150, 151\} Used in section 148.
\{Unstructured structure methods 27, 49, 60, 78\} Used in section 11.
\{Update functional minimization stop criteria 414\} Used in section 413.
\{Write matrix header 191\} Used in sections 190 and 194.
\{Write number to file, using the same tricky method as for longs 192\} Used in sections 190 and 194.
\{algebra.h 118\}
\{barrier/log.h 426\}
\{barrierbase.h 416\}
\{cholesky.h 211\}
\{det.h 243\}
\{eye.h 238\}
\{fmin.h 412\}
\{fstream.h 154\}
\{function/gaxpy.h 251\}
\{functional/entr.h 324\}
\{functional/erf.h 343\}
\{functional/exp.h 353\}
\{functional/gaxpy.h 264\}
\{functional/linear.h 271\}
\{functional/log.h 362\}
\{functional/norm2err.h 290\}
\{functional/power.h 315\}
\{functional/prod.h 298\}
\{functional/quadratic.h 283\}
\{functional/ratio.h 307\}
\{functional/relext.h 333\}
\{functionalbase.h 258\}
\{functionbase.h 245\}
\{linesearch/backtracking.h 381\}
\{linesearch/bisection.h 374\}
\{linesearchbase.h 371\}
\{lu.h 197\}
\{math.h 3\}
\{ones.h 240\}
\{qr.h 225\}
\{searchdir/equality.h 397\}
\{searchdir/equality/newton.h 405\}
\{searchdir/gradient.h 390\}
\{searchdir/newton.h 393\}
\{searchdirbase.h 387\}
\{sparse.h 136\}
\{sumt.h 434\}
\{symmetric.h 147\}
\{bisection big definitions 378\} Used in section 374.
\{export-wait big definitions 33, 35, 39, 43, 80\} Used in section 3.
\{fstream declarations 155, 181\} Used in section 154.
\{fstream structures 162\} Used in section 154.
\{ofstream methods 182, 183, 184, 185, 187, 189, 190, 193, 194\} Used in section 181.
⟨relentr Hessian 341⟩ Used in section 340.
# TABLE OF CONTENTS

**Functionals**
- The gaxpy functional .............................................. 257 68
- The linear combination of functionals .......................... 263 69
- The quadratic functional ........................................... 270 70
- The norm-2 error ................................................... 278 73
- The product of two functionals .................................... 289 75
- The ratio of two functionals ........................................ 306 79
- The power functional ............................................... 314 81
- The entropy of a functional ........................................ 323 83
- Relative entropy ................................................... 332 85
- The Error Function .................................................. 342 88
- The exponential of a functional ................................... 352 90
- The logarithm of a functional ..................................... 361 91

**Line Searching**
- The bisection algorithm .............................................. 370 94
- Backtracking .......................................................... 380 97

**Computing a search direction**
- The gradient direction .............................................. 386 100
- The Newton direction ............................................... 392 101
- Enforcing equality constraints ..................................... 396 103
- Newton direction with equality constraints .................... 404 105

**Optimization algorithms**
- Functional minimization .......................................... 410 109
- Barrier functions ................................................... 411 109
- The log barrier function ........................................... 425 113
- Sequential unconstrained minimization .......................... 433 117

**Index** ...................................................................... 442 121

---

Copyright © 1998–2000 César A. R. Crusius

Permission is granted to make and distribute verbatim copies of this document provided that the copyright
notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this document under the conditions for
verbatim copying, provided that the entire resulting derived work is given a different name and distributed
under the terms of a permission notice identical to this one.