The MATH Library

(Version alpha 3-22-2002)

César A. R. Crusius

March 22, 2002 at 23:26

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 templatized, 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

$$A(i, j) = 2 * A(i, j).$$

The first assignment works fine, but when you try $A(j, i) = 2 * A(j, i)$ 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.

```
#include "math.h"
1   ⟨ Big definitions 112 ⟩
```

3. ⟨ `math.h` 3 ⟩ ≡

```
2  #ifndef __MATH__
3  #define __MATH__ 1.0
4    ⟨ Include files math 6 ⟩
5    ⟨ Preprocessor definitions ⟩
6    namespace math {
7      ⟨ Basic definitions 4 ⟩
8      ⟨ Element definition 65 ⟩
9      ⟨ Structure definition 11 ⟩
10     ⟨ Storage definition 10 ⟩
11     ⟨ Matrix definition 12 ⟩
12     ⟨ Submatrix definition 81 ⟩
13     ⟨ Basic algebraic operations 105 ⟩
14     ⟨ Specializations 106 ⟩
15     ⟨ export-waiting big definitions 33 ⟩
16   }
17 #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**.

⟨ Basic definitions 4 ⟩ ≡

```
18   typedef unsigned int index;
```

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 a 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.

⟨ Basic definitions 4 ⟩ +≡

```
19  namespace error {
20    class generic {
21      string theMessage;
22    public:
23      ⟨ Generic error class methods 7 ⟩
24      virtual ~generic( ) { }      /* A base class it is. */
25    };
26    ⟨ Predefined error types 9 ⟩
27  }
```

6. ⟨ Include files **math** 6 ⟩ ≡

```
28  #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.

⟨ Generic error class methods 7 ⟩ ≡

```
29  generic(void):theMessage("unknown") { }
30  generic(const char *msg):theMessage(msg) { }
```

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 retreive the friendly error message. Depending on the error type we could even not consider it an error (see, for example, the *det* function).

⟨ Generic error class methods 7 ⟩ +≡

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

9. Next we deal with predefined errors. What we do is to derive some classes that implement common errors.

⟨ Predefined error types 9 ⟩ ≡

```
32    class singular : public generic {
33    public: singular( )
34      : generic("Matrix␣is␣singular␣to␣working␣precision.") { } };
35    class filerr : public generic {
36    public: filerr( )
37      : generic("Generic␣file␣error.") { } };
38    class infeasible : public generic {
39    public: infeasible( )
40      : generic("Problem␣is␣infeasible.") { } };
41    class nonsquare : public generic {
42    public: nonsquare( )
43      : generic("Matrix␣should␣be␣square") { } };
44    class nonpositivedef : public generic {
45    public: nonpositivedef( )
46      : generic("Matrix␣should␣be␣positive␣definite.") { } };
47    class dimension : public generic {
48    public: dimension( )
49      : generic("Wrong␣matrix␣dimensions.") { } };
50    class notimplemented : public generic {
51    public: notimplemented( )
52      : generic("Feature␣not␣yet␣implemented.") { } };
53    class maxiterations : public generic {
54    public: maxiterations( )
55      : generic("Maximum␣number␣of␣iterations␣reached.") { } };
56    class unboundedbelow : public generic {
57    public: unboundedbelow( )
58      : generic("Problem␣is␣unbounded␣below.") { } };
59    class domain : public generic {
60    public: domain( )
61      : generic("Domain␣violation.") { } };
62    class rankdefficient : public generic {
63    public: rankdefficient( )
64      : generic("Matrix␣is␣rank␣defficient.") { } };
```

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 ⟩ ≡
65    template⟨class T⟩
66    class dense {
67      typedef T element_type;
68      ⟨ Dense storage internal variables 28 ⟩
69    public:
70      ⟨ Dense storage methods 29 ⟩
71    };
```

This code is used in section 3.

11.

```
⟨ Structure definition 11 ⟩ ≡
72    template⟨class T⟩
73    class unstructured {
74      typedef T element_type;
75    public:
76      ⟨ Unstructured structure methods 27 ⟩
77    };
```

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 assigment 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.

```
#define matrix_simple_template
78          class T ← double ,
79          template⟨class⟩ class structure ← unstructured ,
80          template⟨class⟩ class storage ← dense
```

```
⟨ Matrix definition 12 ⟩ ≡
81    ⟨ Matrix representation definition 13 ⟩
82    template⟨matrix_simple_template⟩
83    class matrix {
84    public:
85      ⟨ Matrix internal types 14 ⟩
86    private:
87      ⟨ Matrix internal variables 15 ⟩
88    public:
89      ⟨ Matrix methods 19 ⟩
90    };
```

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.

⟨ Matrix representation definition 13 ⟩ ≡

```
91  template⟨matrix_simple_template⟩
92  class representation {
93    typedef T element_type;
94    ⟨ Matrix representation internal variables 17 ⟩
95  public:
96    ⟨ Matrix representation methods 18 ⟩
97  };
```

This code is used in section 12.

14. A matrix knows its representation through a pointer to a 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.

⟨ Matrix internal types 14 ⟩ ≡

```
98  typedef representation⟨T, structure, storage⟩ rep_type;
```

See also sections 16 and 88.

This code is used in section 12.

15. ⟨ Matrix internal variables 15 ⟩ ≡

```
99  rep_type *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.

⟨ Matrix internal types 14 ⟩ +≡

```
100 typedef T element_type;
101 typedef storage⟨T⟩ storage_type;
102 typedef structure⟨T⟩ structure_type;
103 typedef matrix⟨T, structure, storage⟩ 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.

⟨ Matrix representation internal variables 17 ⟩ ≡

```
104 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).

⟨ Matrix representation methods 18 ⟩ ≡

```
105  index &rows(void) { return num_rows; }
106  index &cols(void) { return num_cols; }
107  index rows(void) const { return num_rows; }
108  index cols(void) const { return num_cols; }
```

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

This code is used in section 13.

19. ⟨ Matrix methods 19 ⟩ ≡

```
109  index rows(void) const { return (theRepresentation ? theRepresentation→rows( ) : 0); }
110  index cols(void) const { return (theRepresentation ? theRepresentation→cols( ) : 0); }
```

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.

⟨ Matrix representation internal variables 17 ⟩ +≡

```
111  storage⟨T⟩ *theStorage;
112  structure⟨T⟩ *theStructure;
```

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 retreive this information.

⟨ Matrix representation internal variables 17 ⟩ +≡

```
113  int num_instances;
```

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.

⟨ Matrix representation methods 18 ⟩ +≡

```
114  inline int &instances(void) { return num_instances; }
```

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.

⟨ Matrix methods 19 ⟩ +≡

```
115  matrix(void):theRepresentation(0) { }
```

24. ⟨ Matrix methods 19 ⟩ +≡

```
116  matrix(const index rows, const index cols) #ifdef __MATH_INSANE_DEBUG__
117    cout ≪ "[math]: matrix(" ≪ rows ≪ ',' ≪ cols ≪ ") oldrep=" ≪ theRepresentation;
118  #endif
119    theRepresentation ← new rep_type(rows, cols); #ifdef __MATH_INSANE_DEBUG__
120    cout ≪ " newrep=" ≪ theRepresentation ≪ '\n';
121  #endif
122  }
```

25. 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 trhows an error.

⟨ Matrix representation methods 18 ⟩ +≡

```
123  representation(void)
124  {
125    throw error::generic("Cannot instantiate empty matrix representation!");
126  }
```

26. 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.

⟨ Matrix representation methods 18 ⟩ +≡

```
127  representation(index rows, index cols):num_rows(rows), num_cols(cols), num_instances(1)
128  {
129    theStructure ← new structure⟨element_type⟩(&rows, &cols);
130    theStorage ← new storage⟨element_type⟩(rows, cols);
131  }
```

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 *rows* ∗ *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.

⟨ Unstructured structure methods 27 ⟩ ≡

```
132   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 *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 *data*. Later on this can appear to be unintuitive, but this scheme saves memory for vectors, when there is only one element in *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.

⟨ Dense storage internal variables 28 ⟩ ≡

```
133   element_type *elements;
134   element_type **data;
135   index num_rows, num_cols;
136   index max_rows, max_cols;
```

This code is used in section 10.

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

⟨ Dense storage methods 29 ⟩ ≡

```
137   dense( ):elements(0), 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 *memset* for doing this. One could argue that we should do a loop and assign **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 *init* method to be used for initialization and call it from the constructor.

⟨ Dense storage methods 29 ⟩ +≡

```
138   dense(const index &rows, const index &cols)
139   : elements(0), data(0) {
140     init(rows, cols);
141     memset(elements, 0, num_rows * num_cols * sizeof(element_type));
142   }
```

31. ⟨ Include files **math** 6 ⟩ +≡

```
143  #include <string.h>     /* For memset (and memcpy). */
```

32. The *init* function allocates space and initialize 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, *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 *data*[*j*][*i*].

⟨ Dense storage methods 29 ⟩ +≡

```
144   void init(const index &rows, const index &cols);
```

33. ⟨ **export**-waiting big definitions 33 ⟩ ≡

```
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.

⟨ Matrix methods 19 ⟩ +≡

```
void fillwith(const element_type &value);
```

35. ⟨ **export**-waiting big definitions 33 ⟩ +≡

```
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 ⟩ +≡

```
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.

⟨ Matrix methods 19 ⟩ +≡

```
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 19 ⟩ +≡

```
177    void init(const index num_rows, const index num_cols);
```

39. ⟨ **export**-waiting big definitions 33 ⟩ +≡

```
178    template⟨matrix_simple_template⟩ void matrix⟨T, structure, storage⟩::init(const index
           num_rows, const index num_cols)
179    {
180      if (theRepresentation ∧ cols( ) ≡ num_cols ∧ rows( ) ≡ num_rows ∧ theRepresentation→instances( ) ≡ 1)
181        return;
182      if (theRepresentation ∧ −−theRepresentation→instances( ) ≡ 0) delete theRepresentation;
183      theRepresentation ← new rep_type(num_rows, num_cols);
184    }
```

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 4 ⟩ +≡

```
185    extern bool fast_assignment;
```

41.

```
bool math::fast_assignment ← 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 19 ⟩ +≡

```
186    matrix_type &operator←(const matrix_type &source);
```

43. ⟨ **export**-waiting big definitions 33 ⟩ +≡

```
187    template⟨matrix_simple_template⟩ matrix⟨T, structure, storage⟩::matrix_type
           &matrix⟨T, structure, storage⟩::operator←(const matrix_type &source)
188    {
189      if (fast_assignment) return copyfrom(source);
190      if (source.theRepresentation) source.theRepresentation→instances( )++;
191      if (theRepresentation ∧ −−theRepresentation→instances( ) ≡ 0) delete theRepresentation;
192      theRepresentation ← source.theRepresentation;
193      return ∗this;
194    }
```

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 ( ), which retreives an element from a matrix, and the methods *set*, which assigns a value to a particular element. The details are irrelevant for now.

```
#define matrix_template(M)
          class T ##M ,
          template⟨class⟩ class STR##M ,
          template⟨class⟩ class STO##M
```

⟨ Matrix methods 19 ⟩ +≡

```
  template⟨matrix_template(A)⟩
  matrix_type &operator←(const matrix⟨TA, STRA, STOA⟩ &source)
  {
    init(source.rows( ), source.cols( ));
    index i, j;
    for (i ← 1; i ≤ rows( ); ++i)
      for (j ← 1; j ≤ cols( ); ++j) set(i, j, source(i, j));
    return ∗this;
  }
```

45. ⟨ Matrix methods 19 ⟩ +≡

```
  template⟨matrix_template(A)⟩
  matrix(const matrix⟨TA, STRA, STOA⟩ &source):theRepresentation(0)
  { ∗this ← source; }
```

46. On ocasion 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.

⟨ Matrix methods 19 ⟩ +≡

```
  matrix_type &copyfrom(const matrix_type &source)
  {
    if (theRepresentation ≡ source.theRepresentation) return ∗this;
    if (theRepresentation ∧ theRepresentation¬instances( ) ≡ 1 ∧ rows( ) ≡ source.rows( ) ∧ cols( ) ≡
          source.cols( )) {
      ∗theRepresentation ← ∗(source.theRepresentation);
      return ∗this;
    }
    if (theRepresentation ∧ −−theRepresentation¬instances( ) ≡ 0) delete theRepresentation;
    theRepresentation ← new rep_type(∗(source.theRepresentation));
    return ∗this;
  }
```

47. In order to define the *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.

⟨ Matrix representation methods 18 ⟩ +≡

```
221  representation(const representation &source)
222  {
223    theStructure ← new structure⟨T⟩(*(source.theStructure));
224    theStorage ← new storage⟨T⟩(*(source.theStorage));
225    num_instances ← 1;
226    num_rows ← source.rows( );
227    num_cols ← source.cols( );
228  }
```

48. ⟨ Dense storage methods 29 ⟩ +≡

```
229  dense(const dense &source)
230  : elements(0), data(0) {
231    init(source.num_rows, source.num_cols);
232    memcpy(elements, source.elements, num_rows * num_cols * sizeof(element_type));
233  }
```

49. ⟨ Unstructured structure methods 27 ⟩ +≡

```
234  unstructured(const unstructured &) { }
```

50. We also used assignment operators in *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).

⟨ Matrix representation internal variables 17 ⟩ +≡

```
235  typedef representation⟨T, structure, storage⟩ rep_type;
```

51. ⟨ Matrix representation methods 18 ⟩ +≡

```
236  rep_type &operator←(const rep_type &source)
237  {
238    if (source.rows( ) ≠ rows( ) ∨ source.cols( ) ≠ cols( ))
         throw error::generic("Can only assign representation to same dimension.");
239    *theStorage ← *(source.theStorage);
240    *theStructure ← *(source.theStructure);
241    return *this;
242  }
```

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

⟨ Dense storage methods 29 ⟩ +≡

```
243  dense⟨T⟩ &operator←(const dense⟨T⟩ &source)
244  {
245    if (source.num_rows < num_rows ∨ source.num_cols < num_cols)
         throw error::generic("Incompatible dimension in dense assignment operator.");
246    for (index i ← 1; i ≤ num_cols; ++i)
         memcpy(data[i] + 1, source.data[i] + 1, num_rows * sizeof(element_type));
247    return *this;
248  }
```

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.

```
⟨ Matrix methods 19 ⟩ +≡
249   ~matrix(void)
250   {
251 #ifdef __MATH_INSANE_DEBUG__
252     cout ≪ "[math]: ~matrix, this=" ≪ this ≪ ", theRepresentation=" ≪ theRepresentation ≪
          '\n';
253 #endif
254     if (theRepresentation ∧ −−theRepresentation→instances( ) ≡ 0) delete theRepresentation;
255   }
```

54. Deleting a representation requires a representation destructor. If you try to delete a representation that is still shared MATH gives you a little piece of its mind.

```
⟨ Matrix representation methods 18 ⟩ +≡
256   ~representation(void)
257   {
258     if (num_instances) throw error::generic("Deleting a referenced representation!");
259     delete theStorage;
260     delete theStructure;
261   }
```

55. Of course, deleting a representation requires structure and storage destructors. For the unstructured class we rely on the default destructor, since the class is empty anyway. For the dense storage it is convenient, like with the constructing part, to define an auxiliary method and to call it within the destructor. In deleting the *data* vector we only have to keep in mind that it was adjusted during creation to obey the $(1, 1)$ origin default.

```
⟨ Dense storage methods 29 ⟩ +≡
262   ~dense( ) { destroy( ); }
```

56.

```
⟨ Dense storage methods 29 ⟩ +≡
263   void destroy(void)
264   {
265     if (elements) delete[ ] elements;
266     if (data) delete[ ] ++data;
267     elements ← 0;
268     data ← 0;
269   }
```

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.

```
⟨ Matrix methods 19 ⟩ +≡
270   void detach(void)
271   {
272     if (theRepresentation ∧ theRepresentation→instances( ) > 1) {
273       −− theRepresentation→instances( );
274       theRepresentation ← new rep_type(∗theRepresentation);
275     }
276   }
```

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

```
⟨ Matrix methods 19 ⟩ +≡
277   element_type set(index row, index col, element_type value)
278   {
279     detach( );
280     return theRepresentation→set(row, col, value);
281   }
```

59. The represenation 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.

```
⟨ Matrix representation methods 18 ⟩ +≡
282   element_type set(index row, index col, element_type value)
283   {
284     if (theStructure→preprocess(&row, &col)) return theStorage→set(row, col, value);
285     return theStorage→get(row, col);
286   }
```

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

```
⟨ Unstructured structure methods 27 ⟩ +≡
287   bool preprocess(index ∗, index ∗) { return true; }
```

61.

```
⟨ Dense storage methods 29 ⟩ +≡
288   element_type set(const index &row, const index &col, const element_type &value)
289   { return data[col][row] ← value; }
```

62. Wow! We are now able to assign elements to matrices! Next I guess you will want to retreive 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 retreive the value, since the way to retreive 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).

⟨ Matrix methods 19 ⟩ +≡

```
290   const element_type operator( )(const index row, const index col ← 1) const
291   { return theRepresentation→get(row, col); }
292   const element_type get(const index row, const index col) const
293   { return theRepresentation→get(row, col); }
```

63. The representation needs now to perform the communication again. First it calls the structure *preprocess* to get the right index, and then retreive 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.

⟨ Matrix representation methods 18 ⟩ +≡

```
294   const element_type get(index row, index col) const
295   {
296     theStructure→preprocess(&row, &col);
297     return theStorage→get(row, col);
298   }
```

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*).

⟨ Dense storage methods 29 ⟩ +≡

```
299   const element_type get(const index &row, const index &col) const
300   {
301     if (¬row ∨ ¬col) return element_type(0);
302     return data[col][row];
303   }
```

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.

⟨ Element definition 65 ⟩ ≡

```
304   template⟨class matrix_type⟩
305   class element {
306     ⟨ Matrix element internal variables 66 ⟩
307   public:
308     ⟨ Matrix element methods 68 ⟩
309   };
```

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 ⟩ ≡

```
310   matrix_type *theMatrix;
311   index i, j;
```

See also section 67.

This code is used in section 65.

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

⟨ Matrix element internal variables 66 ⟩ +≡

```
312   typedef typename matrix_type::element_type element_type;
```

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 ⟩ ≡

```
313   element(void):theMatrix(0), i(0), j(0)
314   { throw error::generic("Default constructor of element should not be used!"); }
```

See also sections 69, 70, 95, 97, 100, 103, and 107.

This code is used in section 65.

69. ⟨ Matrix element methods 68 ⟩ +≡

```
315   element(matrix_type *mat, const index row, const index col):theMatrix(mat), i(row), j(col) { }
```

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 ⟩ +≡

```
316   inline element_type operator←(const element_type &value) const {
        return theMatrix→set(i, j, value); }
```

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 $A.entry(1,1) \leftarrow 0$ 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 ⟩ +≡

```
317   element⟨matrix_type⟩ entry(const index row, const index col ← 1)
318   {
319     return element⟨matrix_type⟩(this, row, col);
320   }
```

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 ⟩ +≡

```
321   storage⟨T⟩ *storg(void)
322   { return theStorage; }
```

73. ⟨ Matrix methods 19 ⟩ +≡

```
323   storage_type *storg(void)
324   {
325     detach( );
326     return (theRepresentation ? theRepresentation→storg( ) : 0);
327   }
```

74. And sometimes we may want the representation too.

⟨ Matrix methods 19 ⟩ +≡

```
328   rep_type *rep(void)
329   {
330     detach( );
331     return theRepresentation;
332   }
```

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.

⟨ Dense storage methods 29 ⟩ +≡

```
333   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.

⟨ Matrix methods 19 ⟩ +≡

```
334   void resize(const index rows, const index cols)
335   {
336     if (¬theRepresentation) {
337       init(rows, cols);
338       return;
339     }
340     detach( );
341     theRepresentation→resize(rows, cols);
342   }
```

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 **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.

⟨ Matrix representation methods 18 ⟩ +≡

```
343   void resize(index rows, index cols)
344   {
345     num_rows ← rows;
346     num_cols ← cols;
347     theStructure→resize(&rows, &cols);
348     theStorage→resize(rows, cols);
349   }
```

78. ⟨ Unstructured structure methods 27 ⟩ +≡

```
350   void resize(const index *, const index *) const { }
```

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 *init* won't erase the old data. We do that by copying it to another **dense** instance and zeroing out our data pointers.

⟨ Dense storage methods 29 ⟩ +≡

```
351   void resize(const index rows, const index cols);
```

80. ⟨ **export**-waiting big definitions 33 ⟩ +≡

```
352  template⟨class T⟩ void dense⟨T⟩::resize(const index rows, const index cols)
353  {
354    if (cols ≤ max_cols ∧ rows ≤ max_rows) {
355      num_rows ← rows;
356      num_cols ← cols;
357      return;
358    }
359    dense backup(∗this);
360    elements ← 0;
361    data ← 0;
362    init(rows, cols);
363    for (index j ← 1; j ≤ cols ∧ j ≤ backup.num_cols; ++j)
364      for (index i ← 1; i ≤ rows ∧ i ≤ backup.num_rows; ++i) data[j][i] ← backup.data[j][i];
365  }
```

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.

#define *submatrix_template*(M) **class SUBM** $\#\#M$

⟨ Submatrix definition 81 ⟩ ≡

```
366  template⟨class T⟩
367  class submatrix {
368    ⟨ Submatrix internal variables 83 ⟩
369  public:
370    ⟨ Submatrix methods 85 ⟩
371  };
```

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.

⟨ Basic definitions 4 ⟩ +≡

```
372  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.

⟨ Submatrix internal variables 83 ⟩ ≡

```
373  typedef T internal_matrix_type;
374  typedef typename T::element_type element_type;
375  typedef submatrix⟨T⟩ submatrix_type;
376  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.

⟨ Submatrix internal variables 83 ⟩ +≡

```
377  internal_matrix_type *theMatrix;
378  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 intented" 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 funcion you'll need this unless you want your screen filled with "discards const" warnings).

⟨ Submatrix methods 85 ⟩ ≡

```
379  submatrix(void):theMatrix(0)
380  { throw error::generic("Default␣constructor␣of␣submatrix␣sould␣not␣be␣used!"); }
381  submatrix(const internal_matrix_type *mat, const index row1, const index row2, const index
         col1 ← 1, const index col2 ← 1)
382  {
383    theMatrix ← const_cast⟨internal_matrix_type *⟩(mat);
384    i1 ← row1 − 1;
385    i2 ← row2 − 1;
386    j1 ← col1 − 1;
387    j2 ← max(col2, col1) − 1;
388  }
```

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.

⟨ Submatrix methods 85 ⟩ +≡

```
389  submatrix(const internal_matrix_type &mat) { *this ← mat; }
390  submatrix_type &operator←(const internal_matrix_type &mat)
391  {
392    if (mat.rows( ) ≠ i2 − i1 + 1 ∨ mat.cols( ) ≠ j2 − j1 + 1) throw error::dimension( );
393    for (index i ← 1; i ≤ mat.rows( ); ++i)
394      for (index j ← 1; j ≤ mat.cols( ); ++j) theMatrix→set(i1 + i, j1 + j, mat.get(i, j));
395    return *this;
396  }
```

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.

⟨ Submatrix methods 85 ⟩ +≡

```
397  index rows(void) const { return i2 − i1 + 1; }
398  index cols(void) const { return j2 − j1 + 1; }
399  element_type operator( )(const index i, const index j ← 1) const
400  { return theMatrix→get(i1 + i, j1 + j); }
```

88. ⟨ Matrix internal types 14 ⟩ +≡

```
401  typedef submatrix⟨matrix_type⟩ submatrix_type;
```

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

⟨ Matrix methods 19 ⟩ +≡

```
402  template⟨class SUBM⟩ matrix(const submatrix⟨SUBM⟩ &mat)
403  : theRepresentation(0) { *this ← mat; }
404  template⟨class SUBM⟩ matrix_type &operator←(const submatrix⟨SUBM⟩ &mat)
405  {
406    init(mat.rows( ), mat.cols( ));
407    for (index i ← 1; i ≤ rows( ); ++i)
408      for (index j ← 1; j ≤ cols( ); ++j) set(i, j, mat(i, j));
409    return *this;
410  }
```

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.

⟨ Submatrix methods 85 ⟩ +≡

```
411  submatrix(const submatrix_type &mat) { *this ← mat; }
412  submatrix_type &operator←(const submatrix_type &mat)
413  {
414    if (theMatrix ≡ mat.theMatrix ∧ j1 ≤ mat.j2 ∧ j2 ≥ mat.j1 ∧ i1 ≤ mat.i2 ∧ i2 ≥ mat.i1)
415      *this ← internal_matrix_type(mat);
416    else
417      for (index i ← 1; i ≤ rows( ); ++i)
418        for (index j ← 1; j ≤ cols( ); ++j) theMatrix→set(i1 + i, j1 + j, mat(i, j));
419    return *this;
420  }
```

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

⟨ Submatrix methods 85 ⟩ +≡

```
421  void set(index row, index col, element_type value) { theMatrix→set(row + i1, col + j1, value); }
422  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.

⟨ Matrix methods 19 ⟩ +≡

```
423  submatrix_type subm(const index row1, const index row2, const index col1 ← 1, index
         col2 ← 1) const
424  {
425    col2 ← max(col2, col1);
426    return submatrix_type(this, row1, row2, col1, col2);
427  }
```

93. Again we provide a *subm* method for submatrices in order to make them resemble an actual matrix.

⟨ Submatrix methods 85 ⟩ +≡

```
428   submatrix_type subm(const index row1, const index row2, const index col1 ← 1, index
          col2 ← 1) const
429   {
430     col2 ← max(col2, col1);
431     return theMatrix→subm(i1 + row1, i1 + row2, j1 + col1, j1 + col2);
432   }
```

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:

⟨ Matrix methods 19 ⟩ +≡

```
433  matrix_type &operator *= (const T &value)
434  {
435    for (index j ← 1; j ≤ cols( ); ++j)
436      for (index i ← 1; i ≤ rows( ); ++i) set(i, j, value * get(i, j));
437    return *this;
438  }
```

95. ⟨ Matrix element methods 68 ⟩ +≡

```
439  element_type operator *= (const element_type &val)
440  {
441    *this ← theMatrix→get(i, j) * val;
442    return value( );
443  }
```

96. ⟨ Matrix methods 19 ⟩ +≡

```
444  matrix_type &operator /= (const T &value)
445  {
446    for (index j ← 1; j ≤ cols( ); ++j)
447      for (index i ← 1; i ≤ rows( ); ++i) set(i, j, get(i, j)/value);
448    return *this;
449  }
```

97. ⟨ Matrix element methods 68 ⟩ +≡

```
450  element_type operator/=(const element_type &val)
451  {
452    *this ← theMatrix→get(i, j)/val;
453    return value( );
454  }
```

98. ⟨ Submatrix methods 85 ⟩ +≡

```
455  submatrix_type &operator/=(const element_type &value) {
456    for (index j ← 1; j ≤ cols( ); ++j)
457      for (index i ← 1; i ≤ rows( ); ++i) theMatrix→entry(i1 + i, j1 + j) /= value;
458    return *this;
459  }
```

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.

⟨ Matrix methods 19 ⟩ +≡

```
460  template⟨submatrix_template(A)⟩
461  matrix_type &operator+=(const SUBMA &value)
462  {
463    if (rows() ≠ value.rows() ∨ cols() ≠ value.cols()) throw error::dimension();
464    for (index j ← 1; j ≤ cols(); ++j)
465      for (index i ← 1; i ≤ rows(); ++i) set(i, j, get(i, j) + value(i, j));
466    return *this;
467  }
```

100. ⟨ Matrix element methods 68 ⟩ +≡

```
468  element_type operator+=(const element_type &val)
469  {
470    *this ← theMatrix→get(i, j) + val;
471    return value();
472  }
```

101. ⟨ Submatrix methods 85 ⟩ +≡

```
473  submatrix_type &operator+=(const matrix_type &value)
474  {
475    if (value.rows() ≠ rows() ∨ value.cols() ≠ cols()) throw error::dimension();
476    for (index j ← 1; j ≤ cols(); ++j)
477      for (index i ← 1; i ≤ rows(); ++i) theMatrix→entry(i1 + i, j1 + j) += value.get(i, j);
478    return *this;
479  }
```

102. ⟨ Matrix methods 19 ⟩ +≡

```
480  template⟨submatrix_template(A)⟩
481  matrix_type &operator−=(const SUBMA &value)
482  {
483    if (rows() ≠ value.rows() ∨ cols() ≠ value.cols()) throw error::dimension();
484    for (index j ← 1; j ≤ cols(); ++j)
485      for (index i ← 1; i ≤ rows(); ++i) set(i, j, get(i, j) − value(i, j));
486    return *this;
487  }
```

103. ⟨ Matrix element methods 68 ⟩ +≡

```
488  element_type operator−=(const element_type &val)
489  {
490    *this ← theMatrix→get(i, j) − val;
491    return value();
492  }
```

104. ⟨ Submatrix methods 85 ⟩ +≡

```
493   submatrix_type &operator−=(const matrix_type &value)
494   {
495     if (value.rows( ) ≠ rows( ) ∨ value.cols( ) ≠ cols( )) throw error::dimension( );
496     for (index j ← 1; j ≤ cols( ); ++j)
497       for (index i ← 1; i ≤ rows( ); ++i) theMatrix→entry(i1 + i, j1 + j) −= value.get(i, j);
498     return ∗this;
499   }
```

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 ⟩ ≡

```
500   template⟨matrix_simple_template⟩
501   matrix⟨T, structure, storage⟩ transpose(const matrix⟨T, structure, storage⟩ &x)
502   {
503     matrix⟨T, structure, storage⟩ result(x.cols( ), x.rows( ));
504     for (index j ← 1; j ≤ x.cols( ); ++j)
505       for (index i ← 1; i ≤ x.rows( ); ++i) result.set(j, i, x(i, j));
506     return result;
507   }
508   template⟨matrix_simple_template⟩ matrix⟨T, structure, storage⟩ transpose(const
          submatrix⟨matrix⟨T, structure, storage⟩⟩ &x)
509   {
510     matrix⟨T, structure, storage⟩ result(x.cols( ), x.rows( ));
511     for (index j ← 1; j ≤ x.cols( ); ++j)
512       for (index i ← 1; i ≤ x.rows( ); ++i) result.set(j, i, x(i, j));
513     return result;
514   }
```

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 **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.

⟨ Specializations 106 ⟩ ≡

```
515  template⟨class T⟩ inline void swap(element⟨T⟩ x, element⟨T⟩ y)
516  {
517    typename T::element_type aux ← x.value( );
518    x ← y.value( );
519    y ← aux;
520  }
```

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

This code is used in section 3.

107. ⟨ Matrix element methods 68 ⟩ +≡

```
521  inline element_type value(void) const { return theMatrix→get(i, j); }
```

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.

⟨ Matrix methods 19 ⟩ +≡

```
522  void swaprows(const index i, const index k)
523  {
524    for (index j ← 1; j ≤ cols( ); ++j) swap(entry(i, j), entry(k, j));
525  }
```

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).

⟨ Specializations 106 ⟩ +≡

```
526  template⟨matrix_simple_template⟩
527  bool same(const matrix⟨T, structure, storage⟩ &x, const matrix⟨T, structure, storage⟩ &y)
528  {
529    return (x.theRepresentation ≡ y.theRepresentation);
530  }
```

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.

⟨ Specializations 106 ⟩ +≡

```
531  typedef pair⟨math::index, math::index⟩ pair;
```

111. ⟨ Specializations 106 ⟩ +≡

```
532  bool operator < (const math::pair &x, const math::pair &y);
```

112. ⟨ Big definitions 112 ⟩ ≡

```
533   bool operator < (const math::pair &x, const math::pair &y)
534   {
535     if (x.first > y.first) return false;
536     if (x.first < y.first) return true;
537     return (x.second < y.second);
538   }
```

This code is used in section 2.

113. ⟨ Include files **math** 6 ⟩ +≡

```
539  #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 `LAPACK` definition built-in.

⟨ Include files **math** 6 ⟩ +≡

```
540  #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.

⟨ Matrix methods 19 ⟩ +≡

```
541   index numnonzeros(void) const
542   {
543     index result ← 0;
544     for (index j ← 1; j ≤ cols(); ++j)
545       for (index i ← 1; i ≤ rows(); ++i)
546         if (get(i, j)) result++;
547     return result;
548   }
```

116. The *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 **unstructured** and **dense** matrices for now. We use our knowledge about the **dense** storage to make this operation very fast. The source matrix is overwritten with the new one.

⟨ Specializations 106 ⟩ +≡

```
549   template⟨class T⟩
550   matrix⟨T, unstructured, dense⟩ &reshape(matrix⟨T, unstructured, dense⟩ *x, index rows, index
          cols)
551   {
552     if (x→rows() * x→cols() ≠ rows * cols) throw error::dimension();
553     x→storg()→init(rows, cols);
554     x→rep()→rows() ← rows;
555     x→rep()→cols() ← cols;
556     return *x;
557   }
```

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.

```
/* 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.

⟨ 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, 123, 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.

⟨ Algebraic operations 119 ⟩ +≡

```
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;
}
```

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

⟨ Algebraic operations 119 ⟩ +≡

```
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;
}
```

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.

⟨ Algebraic operations 119 ⟩ +≡

```
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;
}
```

125. 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.

```
⟨ Algebraic operations 119 ⟩ +≡
606   template⟨submatrix_template(X), submatrix_template(Y)⟩
607   typename SUBMX::element_type ttdot(const SUBMX &x, const SUBMY &y)
608   {
609     if (x.rows( ) ≠ 1 ∨ y.rows( ) ≠ 1 ∨ x.cols( ) ≠ y.cols( )) throw error::dimension( );
610     typename SUBMX::element_type result ← 0;
611     for (index i ← 1; i ≤ x.cols( ); ++i) result += x(i) ∗ y(i);
612     return result;
613   }
```

126. 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.

```
⟨ Algebraic operations 119 ⟩ +≡
614   template⟨submatrix_template(X)⟩
615   typename SUBMX::element_type norm2(const SUBMX &x)
616   {
617     return (sqrt(dot(x, x)));
618   }
```

127. 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.

```
⟨ Algebraic operations 119 ⟩ +≡
619   template⟨submatrix_template(A), submatrix_template(X), matrix_simple_template⟩
620   matrix⟨T, structure, storage⟩ &axmul(const SUBMA &A, const SUBMX
          &x, matrix⟨T, structure, storage⟩ ∗dest)
621   {
622     index n ← A.rows( );
623     index m ← x.cols( );
624     index p ← x.rows( );
625     if (A.cols( ) ≠ p) throw error::dimension( );
626     dest→resize(n, m);
627     for (index i ← 1; i ≤ n; ++i)
628       for (index j ← 1; j ≤ m; ++j) {
629         T aux ← 0.0;
630         for (index k ← 1; k ≤ p; ++k) aux += A(i, k) ∗ x(k, j);
631         dest→entry(i, j) ← aux;
632       }
633     return ∗dest;
634   }
```

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 ⟩ +≡

```
635   template⟨submatrix_template(A), submatrix_template(X), matrix_simple_template⟩
636   matrix⟨T, structure, storage⟩ &atxmul(const SUBMA &A, const SUBMX
          &x, matrix⟨T, structure, storage⟩ *dest)
637   {
638     index n ← A.cols( );
639     index m ← x.cols( );
640     index p ← x.rows( );
641     if (A.rows( ) ≠ p) throw error::dimension( );
642     dest→resize(n, m);
643     for (index i ← 1; i ≤ n; ++i)
644       for (index j ← 1; j ≤ m; ++j) {
645         T aux ← 0.0;
646         for (index k ← 1; k ≤ p; ++k) aux += A(k, i) * x(k, j);
647         dest→entry(i, j) ← aux;
648       }
649     return *dest;
650   }
```

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 ⟩ +≡

```
651   template⟨submatrix_template(A), submatrix_template(X), submatrix_template(Y)⟩
652   SUBMY &gaxpy(const SUBMA &A, const SUBMX &x, SUBMY *y)
653   {
654     if (y→rows( ) ≠ A.rows( ) ∨ y→cols( ) ≠ x.cols( ) ∨ x.rows( ) ≠ A.cols( )) throw error::dimension( );
655     for (index j ← 1; j ≤ y→cols( ); ++j)
656       for (index i ← 1; i ≤ y→rows( ); ++i)
657         y→set(i, j, y→get(i, j) + tdot(A.subm(i, i, 1, A.cols( )), x.subm(1, x.rows( ), j)));
658     return *y;
659   }
```

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

⟨ Algebraic operations 119 ⟩ +≡

```
660   template⟨submatrix_template(A), submatrix_template(X), submatrix_template(Y)⟩
661   SUBMY &gatxpy(const SUBMA &A, const SUBMX &x, SUBMY *y)
662   {
663     if (y→rows( ) ≠ A.cols( ) ∨ y→cols( ) ≠ x.cols( ) ∨ x.rows( ) ≠ A.rows( )) throw error::dimension( );
664     for (index j ← 1; j ≤ y→cols( ); ++j)
665       for (index i ← 1; i ≤ y→rows( ); ++i)
666         y→set(i, j, y→get(i, j) + dot(A.subm(1, A.rows( ), i), x.subm(1, A.rows( ), j)));
667     return *y;
668   }
```

131. The outer product operation (returns $X \cdot X^T$) is sometimes useful.

⟨ Algebraic operations 119 ⟩ +≡

```
template⟨matrix_simple_template, submatrix_template(X)⟩
matrix⟨T, structure, storage⟩ &outerp(const SUBMX &x, matrix⟨T, structure, storage⟩ *dest)
{
  dest→resize(x.rows( ), x.rows( ));
  T aux;
  for (index i ← 1; i ≤ x.rows( ); ++i)
    for (index j ← i; j ≤ x.rows( ); ++j) {
      aux ← ttdot(x.subm(i, i, 1, x.cols( )), x.subm(j, j, 1, x.cols( )));
      dest→entry(i, j) ← aux;
      dest→entry(j, i) ← aux;
    }
  return *dest;
}
```

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

⟨ Algebraic operations 119 ⟩ +≡

```
template⟨matrix_simple_template, submatrix_template(X)⟩
matrix⟨T, structure, storage⟩ &outterp(const SUBMX &x, matrix⟨T, structure, storage⟩ *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;
      dest→entry(j, i) ← aux;
    }
  return *dest;
}
```

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

⟨ Algebraic operations 119 ⟩ +≡

```
template⟨submatrix_template(Y), submatrix_template(X), matrix_simple_template⟩
matrix⟨T, structure, storage⟩ &xyyx(const SUBMX &x, const SUBMY &y, matrix⟨T,
        structure, storage⟩ *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 $A + \beta x y^T$.

```
⟨ Algebraic operations 119 ⟩ +≡
705   template⟨submatrix_template(A), submatrix_template(X), submatrix_template(Y)⟩ SUBMA
          &outerp_update(SUBMA *A, typename SUBMA::element_type beta, const SUBMX
          &x, const SUBMY &y)
706   {
707     if (y.cols( ) ≠ 1 ∨ x.cols( ) ≠ 1 ∨ A→rows( ) ≠ x.rows( ) ∨ A→cols( ) ≠ y.rows( ))
        throw error::dimension( );
708     index n ← A→rows( );
709     index m ← A→cols( );
710     for (index j ← 1; j ≤ m; ++j)
711       for (index i ← 1; i ≤ n; ++i) A→set(i, j, A→get(i, j) + beta * x(i) * y(j));
712     return *A;
713   }
```

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. ⟨ sparse.h 136 ⟩ ≡

```
#ifndef __MATH_SPARSE__
#define __MATH_SPARSE__ 1.0
  ⟨ Include files sparse 139 ⟩
  namespace math {
    ⟨ Sparse storage definition 137 ⟩
  }
#endif
```

137. ⟨ Sparse storage definition 137 ⟩ ≡

```
  template⟨class T⟩
  class sparse {
    ⟨ Sparse storage internal variables 138 ⟩
  public:
    ⟨ Sparse storage methods 140 ⟩
  };
```

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.

⟨ Sparse storage internal variables 138 ⟩ ≡

```
  typedef map⟨math::pair, T, less⟨math::pair⟩⟩ storage;
  storage *elements;
```

This code is used in section 137.

139. ⟨ Include files **sparse** 139 ⟩ ≡

```
#include <math/math.h>    /* For math::pair – otherwise unecessary. */
#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.

⟨ Sparse storage methods 140 ⟩ ≡

```
  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. ⟨ Sparse storage methods 140 ⟩ +≡

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

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.

⟨ Sparse storage methods 140 ⟩ +≡

```
739    void set(const index &row, const index &col, const T &value)
740    {
741      math::pair i(row, col);
742      if (value ≠ T(0)) (*elements)[i] ← value;
743      else if (elements¬find(i) ≠ elements¬end( )) elements¬erase(i);
744    }
```

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 *row* or *col* are zero we are supposed to return 0 – which will happen because there is no element with index $(0, 0)$ stored in a sparse matrix.

⟨ Sparse storage methods 140 ⟩ +≡

```
745    const T get(const index &row, const index &col) const
746    {
747      math::pair i(row, col);
748      if (elements¬find(i) ≠ elements¬end( )) return (*elements)[i];
749      return T(0);
750    }
```

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.

⟨ Sparse storage methods 140 ⟩ +≡

```
751    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).

146. 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 */
```

147. ⟨ symmetric.h 147 ⟩ ≡

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

148. ⟨ Symmetric structure definition 148 ⟩ ≡

```
  template⟨class T⟩
  class symmetric {
  public:
    ⟨ Symmetric structure methods 149 ⟩
  };
```

This code is used in section 147.

149. 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

$$\begin{bmatrix} a_{11} & a_{12} & a_{22} & a_{13} & a_{23} & a_{33} \end{bmatrix}.$$

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); }
```

See also sections 150 and 151.

This code is used in section 148.

150. 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.

⟨ Symmetric structure methods 149 ⟩ +≡

```
771   bool preprocess(math::index *row, math::index *col) const
772   {
773     if (*col ≥ *row) {
774       *row ← (*col − 1) * (*col)/2 + (*row);
775       *col ← 1;
776       return true;      /* Assignable. */
777     }
778     *row ← (*row − 1) * (*row)/2 + (*col);
779     *col ← 1;
780     return false;      /* Not assignable. */
781   }
```

151. We need a copy constructor, which is empty.

⟨ Symmetric structure methods 149 ⟩ +≡

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

152. 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.

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

154. ⟨ `fstream.h` 154 ⟩ ≡

```
#ifndef __MATH_FSTREAM__
#define __MATH_FSTREAM__ 1.0
  ⟨ Include files fstream 156 ⟩
#include <math/math.h>
  namespace math {
    ⟨ fstream structures 162 ⟩
    ⟨ 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.

⟨ *fstream* declarations 155 ⟩ ≡

```
  class ifstream : public std::ifstream {
  public:
    ⟨ ifstream methods 157 ⟩
  };
```

See also section 181.

This code is used in section 154.

156. ⟨ Include files *fstream* 156 ⟩ ≡

```
#include <fstream>
```

See also sections 163 and 171.

This code is used in section 154.

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

⟨ **ifstream** methods 157 ⟩ ≡

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

See also sections 158, 159, 160, 164, 166, 167, 169, 170, 172, 174, 176, and 178.

This code is used in section 155.

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.

⟨ **ifstream** methods 157 ⟩ +≡

```
  ifstream(const char *name, int mode ← ios::in | ios::binary, int prot ← °664)
  : std::ifstream(name, mode, prot) { }
```

159. ⟨ **ifstream** methods 157 ⟩ +≡

```
803   void open(const char *name, int mode ← ios::in | ios::binary, int prot ← °664 )
804   {
805     std::ifstream::open(name, mode, prot);
806   }
```

160. Unfortunately, the binary format of my lybrary does not work for **long** or **double**, that is, I cannot use the operator ≫ 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 **unsigned char** and **long** (or **double**) and read **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 ≫ operator for the **long** case, because it will be convenient when reading matrices headers. For the **double** case it is simpler to do the trick when reading the matrix data itself.

⟨ **ifstream** methods 157 ⟩ +≡

```
807   ifstream &operator≫(long &);
```

161.

```
808   math::ifstream &math::ifstream::operator≫(long &dest)
809   {
810     union {
811       long Long;
812       unsigned char Char[sizeof(long)];
813     } tricky;
814     for (int i ← 0; i ≠ sizeof(long); this→get(tricky.Char[i++])) ;
815     dest ← tricky.Long;
816     return *this;
817   }
```

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.

⟨ *fstream* structures 162 ⟩ ≡

```
818   struct fheader {
819     long type;      /* Type of the object. */
820     long rows;      /* Number of rows. */
821     long cols;      /* Number of cols. */
822     bool iflag;     /* true if matrix has imaginary part. */
823     string name;    /* Matrix name. */
824   };
```

This code is used in section 154.

163. ⟨ Include files *fstream* 156 ⟩ +≡

```
825   #include <string>
```

164. The actual data on a MATLAB file is somewhat different. The three first fields from **fheader** are the same. The *iflag* field is a **long** on the original file, and after it there is a **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.

⟨ **ifstream** methods 157 ⟩ +≡

```
826   void parse_header(fheader &header);
```

165. **void math::ifstream::***parse_header*(**math::fheader** &*header*)

```
{
  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 ≠ int(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.

⟨ **ifstream** methods 157 ⟩ +≡

```
template⟨matrix_simple_template⟩
ifstream &operator≫(matrix⟨T, structure, storage⟩ &dest)
```

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

⟨ **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. ⟨ Get *number* from file, using the same tricky method as for **long**s 168 ⟩ ≡

```
union {
  double Double;
  unsigned char Char[sizeof(double)];
} tricky;
for (int k ← 0; k ≠ sizeof(double); this¬get(tricky.Char[k++])) ;
number ← tricky.Double;
```

This code is used in sections 167 and 170.

169. 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.

```
⟨ ifstream methods 157 ⟩ +≡
857   template⟨matrix_simple_template⟩
858   ifstream &operator≫(matrix⟨complex⟨T⟩, structure, storage⟩ &dest)
```

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

```
⟨ ifstream methods 157 ⟩ +≡
859   {
860     math::fheader header;
861     parse_header(header);
862     if (rdstate() ≠ goodbit) throw math::error::filerr();
863     dest.init(header.rows, header.cols);
864     for (int ipart ← 0; ipart ≤ header.iflag; ipart++)
865       for (int j ← 1; j ≤ header.cols; j++)
866         for (int i ← 1; i ≤ header.rows; i++) {
867           double number;
868           ⟨ Get number from file, using the same tricky method as for longs 168 ⟩
869           if (¬ipart) dest.set(i, j, number);
870           else dest.set(i, j, dest(i, j) + number ∗ std::complex⟨double⟩(0, 1));
871         }
872     return ∗this;
873   }
```

171. ⟨ Include files *fstream* 156 ⟩ +≡

```
874  #include <complex>
```

172. 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.

```
⟨ ifstream methods 157 ⟩ +≡
875   void skip(int num_matrices);
```

173.

```
876   void math::ifstream::skip(int num_matrices)
877   {
878     math::fheader header;
879     while (num_matrices−−) {
880       parse_header(header);
881       if (rdstate() ≠ goodbit)
882         throw math::error::generic("Too␣many␣matrices␣to␣skip.");
883       skip_data(header);
884     }
   }
```

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

```
⟨ ifstream methods 157 ⟩ +≡
885   void skip_data(const fheader &header);
```

175. **void math::ifstream::***skip_data*(**const math::fheader** &*header*)

```
886  {
887    long data_size ← header.cols * header.rows * sizeof(double);
888    if (header.iflag) data_size *= 2;
889    for (unsigned char garbage; data_size; data_size--) this→get(garbage);
890  }
```

176. With the above definitions we can jump to an arbitrary matrix on the file. The first matrix has position 1.

⟨ **ifstream** methods 157 ⟩ +≡

```
891  void skipto(const int position);
```

177. **void math::ifstream::***skipto*(**int** *position*)

```
892  {
893    seekg(0);
894    if (--position > 0) skip(position);
895  }
```

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 ⟩ +≡

```
896  void skipto(const char *matrix_name);
```

179. **void math::ifstream::***skipto*(**const char** **matrix_name*)

```
897  {
898    int position ← 0;
899    seekg(0);
900    ⟨ Search for correct matrix name 180 ⟩
901    skipto(position);
902  }
```

180. ⟨ Search for correct matrix name 180 ⟩ ≡

```
903  math::fheader header;
904  header.name ← "";
905  while (strcmp(header.name.c_str(), matrix_name)) {
906    if (position) skip_data(header);      /* Skip data of previous matrix */
907    parse_header(header);
908    if (rdstate() ≠ goodbit)
909      throw math::error::generic("No␣matrix␣with␣supplied␣name␣in␣the␣file.");
910    position++;
911  }
```

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.

⟨ *fstream* declarations 155 ⟩ +≡

```
912  class ofstream : public std::ofstream {
913    string matrix_name;
914  public:
915    ⟨ ofstream methods 182 ⟩
916  };
```

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

⟨ **ofstream** methods 182 ⟩ ≡

```
917  ofstream( )
918  : std::ofstream( ) { }
919  ofstream(int fd)
920  : std::ofstream(fd) { }
```

See also sections 183, 184, 185, 187, 189, 190, 193, and 194.

This code is used in section 181.

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.

⟨ **ofstream** methods 182 ⟩ +≡

```
921  ofstream(const char *name, int mode ← ios::out | ios::binary, int prot ← °664 )
922  : std::ofstream(name, mode, prot) { }
```

184. ⟨ **ofstream** methods 182 ⟩ +≡

```
923  void open(const char *name, int mode ← ios::in | ios::binary, int prot ← °664 )
924  {
925    std::ofstream::open(name, mode, prot);
926  }
```

185. 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.

⟨ **ofstream** methods 182 ⟩ +≡

```
927  ofstream &operator≪(const char *name);
```

186.

```
928  math::ofstream &math::ofstream::operator≪(const char *name)
929  {
930    matrix_name ← name;
931    return *this;
932  }
```

187. Like with the input stream case, we define a method to output **long**s. The trick used is exactly the same.

⟨ **ofstream** methods 182 ⟩ +≡

```
933  ofstream &operator≪(const long &);
```

188. **math**::**ofstream** &**math**::**ofstream**::**operator**≪(**const long** &*source*)

```
{
  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.

⟨ **ofstream** methods 182 ⟩ +≡

```
template⟨matrix_simple_template⟩
ofstream &operator≪(const matrix⟨T, structure, storage⟩ &source)
```

190. Waiting for **export**...

⟨ **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 mehtod as for longs 192 ⟩
    }
  return ∗this;
}
```

191. ⟨ Write matrix header 191 ⟩ ≡

```
∗this ≪ long(0) ≪ long(source.rows()) ≪ long(source.cols()) ≪ iflag ≪
    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 mehtod as for **long**s 192 ⟩ ≡

```
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.

⟨ **ofstream** methods 182 ⟩ +≡

```
template⟨matrix_simple_template⟩
ofstream &operator≪(const matrix⟨complex⟨T⟩, structure, storage⟩ &source)
```

194. Waiting for **export**...

```
⟨ 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 mehtod 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>      /* For templatized abs. */
  namespace math {
  namespace lu {
    ⟨ LU prototypes 199 ⟩
  } }
#endif
```

198. We can understand the way the decomposition works by considering the first step in a 3×3 matrix. We build a matrix M_1 such that the first column of $M_1 A$ 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 ith 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 3×3 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.

199. 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.

```
⟨ LU prototypes 199 ⟩ ≡
990   template⟨class T, template⟨class⟩ class storage⟩
991   int decompose(matrix⟨T, unstructured, storage⟩ *A, vector⟨index⟩ *pivots)
992   {
993     int permutations ← 1;      /* No permutations yet. */
994     index n ← A→rows( );
995     if (n ≠ A→cols( )) throw error::nonsquare( );
996     pivots→resize(n − 1, 0);      /* Reserve space for permutation data. */
997     for (index k ← 1; k ≠ n; ++k) {
998       T pivot;
999       ⟨ Search for pivot and swap rows if necessary 200 ⟩
1000      if (pivot ≠ 0) ⟨ Apply Gauss transformation 201 ⟩
1001    }
1002    return permutations;
1003  }
```

See also sections 202, 203, and 204.

This code is used in section 197.

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

```
1004  pivot ← A→get(k, k);
1005  (*pivots)[k − 1] ← k;      /* STL vector has base index 0. */
1006  for (index i ← k + 1; i ≤ n; ++i)
1007    if (abs(A→get(i, k)) > abs(pivot)) {
1008      pivot ← A→get(i, k);
1009      (*pivots)[k − 1] ← i;
1010    }
1011  if ((*pivots)[k − 1] ≠ k)      /* If need to swap. */
1012  {
1013    permutations *= −1;
1014    A→swaprows(k, (*pivots)[k − 1]);
1015  }
```

This code is used in section 199.

201. ⟨ Apply Gauss transformation 201 ⟩ ≡

```
1016  for (index i ← k + 1; i ≤ n; ++i) {
1017    if (¬finite(A→entry(i, k) ← A→get(i, k)/A→get(k, k))) throw error::singular( );
1018    for (index j ← k + 1; j ≤ n; ++j) A→entry(i, j) ← A→get(i, j) − A→get(i, k) * A→get(k, j);
1019  }
```

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 $P_{n-1} \dots P_1 A = LU$, where P_i are permutation matrices defined by the *pivots* vector. Having the LU decomposition, we solve the system $Ax = b$ by first solving $Ly = P_{n-1} \dots P_1 b$ 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 $A^k X = B$, you need to perform only one decomposition and call this function k times.

```
⟨ LU prototypes 199 ⟩ +≡
  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} ... 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 (¬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 $B = I_n$.

```
⟨ LU prototypes 199 ⟩ +≡
  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.

```
⟨ LU prototypes 199 ⟩ +≡
#ifdef LAPACK
  ⟨ LU lapack interface 205 ⟩
#endif
```

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.

```
⟨ LU lapack interface 205 ⟩ ≡
  template⟨⟩
  int decompose(matrix⟨fortran::double_precision, unstructured, dense⟩ *A, vector⟨index⟩
      *pivots)
  {
    ⟨ Prepare for lapack LU 206 ⟩
    dgetrf(&m, &n, A→storg()→memory(), &m, (int *) pivots→begin(), &status);
    ⟨ Check for lapack LU errors 207 ⟩
    ⟨ Compute number of permutations and store it in status 208 ⟩
    return status;
  }
```

See also section 209.

This code is used in section 204.

206. ⟨ Prepare for lapack LU 206 ⟩ ≡

```
  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.

207. ⟨ Check for lapack LU errors 207 ⟩ ≡

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

This code is used in sections 205 and 209.

208. 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).

```
⟨ Compute number of permutations and store it in status 208 ⟩ ≡
  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. ⟨ LU lapack interface 205 ⟩ +≡

```
template⟨⟩
int decompose(matrix⟨fortran::real, unstructured, dense⟩ *A, vector⟨index⟩ *pivots)
{
  ⟨ Prepare for lapack LU 206 ⟩
  sgetrf(&m, &n, A→storg()→memory(), &m, (int *) pivots→begin(), &status);
  ⟨ Check for lapack LU errors 207 ⟩
  ⟨ Compute number of permutations and store it in status 208 ⟩
  return status;
}
```

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."

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

211. ⟨ cholesky.h 211 ⟩ ≡

```
#ifndef __MATH_CHOLESKY__
#define __MATH_CHOLESKY__ 1.0
#include <math/math.h>
#include <math/symmetric.h>
#include <math.h>      /* for sqrt. */
  namespace math {
  namespace cholesky {
    ⟨ Cholesky prototypes 212 ⟩
  } }
#endif
```

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

⟨ Cholesky prototypes 212 ⟩ ≡

```
  template⟨matrix_simple_template⟩
  void decompose(matrix⟨T, structure, storage⟩ *A)
  {
    index n ← A→rows( );
    if (n ≠ A→cols( )) throw error::nonsquare( );
    for (index k ← 1; k ≤ n; ++k) {
      if (A→get(k, k) ≤ 0) throw error::nonpositivedef( );
      ⟨ Build L^T row 213 ⟩;
      ⟨ Apply Cholesky transformation 214 ⟩;
    }
  }
```

See also sections 215, 216, and 217.

This code is used in section 211.

213. ⟨ Build L^T row 213 ⟩ ≡

```
  A→entry(k, k) ← sqrt(A→get(k, k));
  for (index j ← k + 1; j ≤ n; ++j)
    if (¬finite(A→entry(k, j) ← A→get(k, j)/A→get(k, k))) throw error::singular( );
```

This code is used in section 212.

214. ⟨ Apply Cholesky transformation 214 ⟩ ≡

```
1103    for (index i ← k + 1; i ≤ n; ++i)
1104      for (index j ← i; j ≤ n; ++j) A→entry(i, j) ← A→get(i, j) − A→get(k, i) ∗ A→get(k, j);
```

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.

⟨ Cholesky prototypes 212 ⟩ +≡

```
1105    template⟨class T, template⟨class⟩ class structure_A, template⟨class⟩ class
          structure_B, template⟨class⟩ class storage⟩
1106    void finish(const matrix⟨T, structure_A, storage⟩ &A, matrix⟨T, structure_B, storage⟩ ∗B)
1107    {
1108      index n ← A.rows( );
1109      for (index k ← B→cols( ); k ≠ 0; −−k) {
1110        for (index i ← 1; i ≤ n; ++i)      /∗ Solve Ly = B(:, k). ∗/
1111        {
1112          T inner ← 0;
1113          for (index j ← 1; j < i; ++j)
                /∗ Here we have to remember that we overwrote only the upper triangular part of A, so that
                   now we have to get the element A(j, i) instead of A(i, j). ∗/
1114            inner += A(j, i) ∗ B→get(j, k);
1115          if (¬finite(B→entry(i, k) ← (B→get(i, k) − inner)/A(i, i))) throw error::singular( );
1116        }
1117        for (index i ← n; i ≥ 1; −−i)      /∗ Solve L^T x = y. ∗/
1118        {
1119          T inner ← 0;
1120          for (index j ← i + 1; j ≤ n; ++j) inner += A(i, j) ∗ B→get(j, k);
1121          if (¬finite(B→entry(i, k) ← (B→get(i, k) − inner)/A(i, i))) throw error::singular( );
1122        }
1123      }
1124    }
```

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!!!

⟨ Cholesky prototypes 212 ⟩ +≡

```
1125    template⟨class T, template⟨class⟩ class structure_A, template⟨class⟩ class
          structure_B, template⟨class⟩ class storage⟩
1126    matrix⟨T, structure_B, storage⟩ &solve(matrix⟨T, structure_A, storage⟩ ∗A, matrix⟨T,
            structure_B, storage⟩ ∗B)
1127    {
1128      decompose(A);
1129      finish(∗A, B);
1130      return ∗B;
1131    }
```

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?

```
⟨ Cholesky prototypes 212 ⟩ +≡
#include <math/private/fortran.h>
#ifdef HAVE_LIBLAPACK
  ⟨ Cholesky lapack interface 218 ⟩
#endif
```

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

```
⟨ Cholesky lapack interface 218 ⟩ ≡
  template⟨⟩
  void decompose(matrix⟨fortran::double_precision, symmetric, dense⟩ *A)
  {
    ⟨ Prepare for lapack Cholesky 219 ⟩
    dpptrf(&mode, &n, A→storg()→memory(), &status);
    ⟨ Check lapack Cholesky errors 220 ⟩
  }
```

See also sections 221, 222, and 223.

This code is used in section 217.

219. ⟨ Prepare for lapack Cholesky 219 ⟩ ≡

```
  fortran::integer n ← A→rows();
  fortran::integer m ← A→cols();
  fortran::integer status ← 0;
  fortran::character mode ← 'U';
  if (n ≠ m) throw error::nonsquare();
```

This code is used in sections 218, 221, 222, and 223.

220. ⟨ Check lapack Cholesky errors 220 ⟩ ≡

```
  if (status > 0) throw error::nonpositivedef();
  if (status < 0) throw error::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.

```
⟨ Cholesky lapack interface 218 ⟩ +≡
  template⟨⟩
  void decompose(matrix⟨fortran::real, symmetric, dense⟩ *A)
  {
    ⟨ Prepare for lapack Cholesky 219 ⟩
    spptrf(&mode, &n, A→storg()→memory(), &status);
    ⟨ Check lapack Cholesky errors 220 ⟩
  }
```

222. We begin with the symmetric matrix cases.

```
⟨ Cholesky lapack interface 218 ⟩ +≡
1157    template⟨ ⟩
1158    void decompose(matrix⟨fortran::double_precision, unstructured, dense⟩ *A)
1159    {
1160      ⟨ Prepare for lapack Cholesky 219 ⟩
1161      dpotrf(&mode, &n, A→storg( )→memory( ), &m, &status);
1162      ⟨ Check lapack Cholesky errors 220 ⟩
1163    }
```

223. ⟨ Cholesky lapack interface 218 ⟩ +≡

```
1164    template⟨ ⟩
1165    void decompose(matrix⟨fortran::real, unstructured, dense⟩ *A)
1166    {
1167      ⟨ Prepare for lapack Cholesky 219 ⟩
1168      spotrf(&mode, &n, A→storg( )→memory( ), &m, &status);
1169      ⟨ Check lapack Cholesky errors 220 ⟩
1170    }
```

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.

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

225. ⟨ qr.h 225 ⟩ ≡

```
#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.

⟨ QR prototypes 226 ⟩ ≡

```
  template⟨class T, template⟨class⟩ class storage⟩
  matrix⟨T, unstructured, storage⟩ &decompose(matrix⟨T, unstructured, storage⟩ *A)
  {
    matrix⟨T, unstructured, dense⟩ v(A→rows( ), 1);
    matrix⟨T, unstructured, dense⟩ w(A→cols( ), 1);
    T beta;
    for (index j ← 1; j ≤ A→cols( ); ++j) {
      ⟨ Compute Householder vector for column j 227 ⟩;
      ⟨ Apply Householder transformation 228 ⟩;
    }
    return *A;
  }
```

See also section 229.

This code is used in section 225.

227. A Householder reflection can in fact be represented by a single vector v and a scalar β. In fact, the definition of a Householder matrix is any $n-$ by $-n$ 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 Px 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 $Px = P\beta x^0{}_2 e_1$. The normalization of the first entry of the Householder vector is desirable becaise 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 $x_1 > 0$.

⟨ Compute Householder vector for column j 227 ⟩ ≡

```
1192    v.resize(A→rows( ) − j + 1, 1);
1193    v.entry(1) ← T(0.0);
1194    v.subm(2, v.rows( )) ← A→subm(j + 1, A.rows( ), j, j);
1195    T sigma ← dot(v, v);
1196    v.entry(1) ← T(1.0);
1197    if (sigma ≡ 0) beta ← 0;
1198    else {
1199      T x ← A→get(j, j);
1200      T mu ← sqrt(sigma + x ∗ x);
1201      v.entry(1) ← (x ≤ 0 ? x − mu : −sigma/(x + mu));
1202      T v1 ← v(1);
1203      beta ← 2 ∗ v1 ∗ v1/(sigma + v1 ∗ v1);
1204      v /= v1;
1205    }
```

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.

```
⟨ Apply Householder transformation 228 ⟩ ≡
1206    submatrix⟨matrix⟨T, unstructured, storage⟩⟩ Ablock(A, j, A→rows( ), j, A→cols( ));
1207    w ← atxmul(Ablock, v, &w);
1208    (void) outerp_update(&Ablock, −beta, v, w);
1209    if (j < A→rows( )) A→subm(j + 1, A→rows( ), j, j) ← v.subm(2, v.rows( ));
```

This code is used in section 226.

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 $A \in \mathbf{R}^{m\times n}$ 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 $A \in \mathbf{R}^{m\times n}$ and $n > m$. The system $Ax = b$ either has an infinite number of solutios or none. If it does have one, we compute the *minimum norm* solution.

```
⟨ QR prototypes 226 ⟩ +≡
1210    template⟨class T, template⟨class⟩ class storage⟩
1211    matrix⟨T, unstructured, storage⟩ &solve(matrix⟨T, unstructured, storage⟩
            ∗A, matrix⟨T, unstructured, dense⟩ ∗B)
1212    {
1213      index l ← B→cols( );
1214      index m ← A→rows( );
1215      index n ← A→cols( );
1216      matrix⟨T, unstructured, dense⟩ w(l, 1);
1217      if (m ≥ n) {
1218        ⟨ Solve least squares problem 230 ⟩;
1219      }
1220      else {
1221        ⟨ Solve minimum norm problem 233 ⟩;
1222      }
1223      return ∗B;
1224    }
```

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.

```
⟨ Solve least squares problem 230 ⟩ ≡
1225    (void) decompose(A);
1226    ⟨ Solve Qy = b 231 ⟩;
1227    ⟨ Solve Rx = y 232 ⟩;
1228    B→resize(n, l);
```

This code is used in section 229.

231. To solve $Qy = b$ we have to remember that Q is orthogonal, so all we have to do is to compute $y = Q^T b$.

```
⟨ Solve Qy = b 231 ⟩ ≡
  matrix⟨T, unstructured, storage⟩ v(m, 1);
  v.entry(1) ← T(1);
  for (index j ← 1; j ≤ n; ++j) {
    v.resize(m − j + 1, 1);
    v.subm(2, v.rows( )) ← A→subm(j + 1, m, j, j);
    submatrix⟨matrix⟨T, unstructured, dense⟩⟩ Bblock(B, j, m, 1, l);
    w ← atxmul(Bblock, v, &w);
    outerp_update(&Bblock, −2/dot(v, v), v, w);
  }
```

This code is used in section 230.

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

```
⟨ Solve Rx = y 232 ⟩ ≡
  for (index k ← 1; k ≤ l; ++k)
    for (index i ← n; i ≥ 1; −−i) {
      T inner ← 0;
      for (index j ← i + 1; j ≤ n; ++j) inner += A→get(i, j) ∗ B→get(j, k);
      if (¬finite(B→entry(i, k) ← (B→get(i, k) − inner)/A→get(i, i))) throw error::rankdefficient( );
    }
```

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 $Q \in \mathbf{R}^{n\times n}$ and $R \in \mathbf{R}^{n\times m}$. The matrix we'll work with is At, which is in $\mathbf{R}^{n\times m}$.

```
⟨ Solve minimum norm problem 233 ⟩ ≡
  matrix⟨T, unstructured, storage⟩ At ← transpose(∗A);
  (void) decompose(&At);
  ⟨ Solve R^T y = b 234 ⟩;
  ⟨ Solve Q^T x = y 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 $[\,R_1 \quad 0\,]\,[\,y_1\,;\,y_2\,] = b$, 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.

```
⟨ Solve R^T y = b 234 ⟩ ≡
  for (index k ← 1; k ≤ l; ++k)
    for (index i ← 1; i ≤ m; ++i) {
      T inner ← 0;
      for (index j ← 1; j < i; ++j) inner += At(j, i) ∗ B→get(j, k);
      if (¬finite(B→entry(i, k) ← (B→get(i, k) − inner)/At.(i, i))) throw error::rankdefficient( );
    }
```

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.

⟨ Solve $Q^T x = y$ 235 ⟩ ≡

```
1254    B→resize(n, l);
1255    B→subm(m + 1, n, 1, l) ← matrix⟨T, unstructured, storage⟩(n − m, l);
1256    matrix⟨T, unstructured, storage⟩ v(n, 1);
1257    v.entry(1) ← T(1);
1258    for (index k ← m; k ≥ 1; −−k) {
1259      index nrows ← n − k + 1;
1260      v.resize(nrows, 1);
1261      v.subm(2, nrows) ← At.subm(k + 1, n, k, k);
1262      submatrix⟨matrix⟨T, unstructured, dense⟩⟩ Bblock(B, k, n, 1, l);
1263      w ← atxmul(Bblock, v, &w);
1264      outerp_update(&Bblock, −2/dot(v, v), v, w);
1265    }
```

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.

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

238. ⟨ eye.h 238 ⟩ ≡

```
#ifndef __MATH_EYE__
#define __MATH_EYE__ 1.0
#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, whe 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 240 ⟩ ≡

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

241. Matrix functions.

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

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

where A_{1j} is an $(n-1)$-by-$(n-1)$ matrix obtained by deleting the first row and jth comlumn 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 *permutted* 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 ⟩ ≡

```
#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.

/* Not until **export** */

245. ⟨ functionbase.h 245 ⟩ ≡

```
#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.

⟨ Function base class methods 246 ⟩ ≡

```
  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.

⟨ Function base class methods 246 ⟩ +≡

```
  matrix⟨T, unstructured, dense⟩ &operator( )(const matrix⟨T, unstructured, dense⟩
        &x, matrix⟨T, unstructured, dense⟩ *dest)
  { return eval(x, dest); }
  virtual matrix⟨T, unstructured, dense⟩ &eval(const matrix⟨T, unstructured, dense⟩
    &x, matrix⟨T, unstructured, dense⟩ *dest) ← 0;
```

248. If possible, a function should compute the Jacobian. If $f : \mathbf{R}^n \to \mathbf{R}^m$, then the Jacobian is a matrix $J \in \mathbf{R}^{m \times n}$ defined by $J_{ij} = df_i/dx_j$. 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, *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**::*resize* method. Since this method checks for same dimensions, the performance does not suffer too much.

⟨ Function base class methods 246 ⟩ +≡

```
1332  virtual matrix⟨T, unstructured, dense⟩ &jacobian(const matrix⟨T, unstructured, dense⟩
          &x, matrix⟨T, unstructured, dense⟩ *dest) ← 0;
```

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 *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).

⟨ Function base class methods 246 ⟩ +≡

```
1333  virtual matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
          &x, matrix⟨T, symmetric, dense⟩ *dest, const index i, const index j ← 1) ← 0;
1334  virtual matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
          &x, matrix⟨T, symmetric, sparse⟩ *dest, const index i, const index j ← 1) ← 0;
```

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 */
```

251. ⟨ function/gaxpy.h 251 ⟩ ≡

```
1335  #ifndef __MATH_GAXPY_FUNCTION__
1336  #define __MATH_GAXPY_FUNCTION__ 1.00
1337  #include <math/math.h>
1338  #include <math/algebra.h>
1339  #include <math/functionbase.h>
1340    namespace math {
1341      namespace function
1342      {
1343        template⟨matrix_simple_template⟩
1344        class gaxpy:public base⟨T⟩ {
1345          ⟨ Gaxpy function internal variables 252 ⟩
1346        public:
1347          ⟨ Gaxpy function class methods 253 ⟩
1348        };
1349      }
1350    }
1351  #endif
```

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

⟨ Gaxpy function internal variables 252 ⟩ ≡

```
1352    matrix⟨T, structure, storage⟩ ay;
1353    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.

⟨ Gaxpy function class methods 253 ⟩ ≡

```
1354    matrix⟨T, structure, storage⟩ &A(void) { return ay; }
1355    matrix⟨T, unstructured, dense⟩ &b(void) { return bee; }
1356    gaxpy(const matrix⟨T, structure, storage⟩ &a, const T &B)
1357    : 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.

⟨ Gaxpy function class methods 253 ⟩ +≡

```
1358    virtual matrix⟨T, unstructured, dense⟩ &eval(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, unstructured, dense⟩ *dest)
1359    { return (gaxpy(ay, x, &(dest→copyfrom(b)))); }
```

255. The Jacobian is simply A.

⟨ Gaxpy function class methods 253 ⟩ +≡

```
1360    virtual matrix⟨T, unstructured, dense⟩ &jacobian(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, unstructured, dense⟩ *dest)
1361    {
1362      return (*dest) ← ay;
1363    }
```

256. The hessian of an affine function is zero.

⟨ Gaxpy function class methods 253 ⟩ +≡

```
1364    virtual matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, dense⟩ *dest, const index, const index)
1365    {
1366      dest→resize(x.rows( ), x.rows( ));
1367      dest→fillwith(0);
1368      return *dest;
1369    }
1370    virtual matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, sparse⟩ *dest, const index, const index)
1371    {
1372      *dest ← matrix⟨T, symmetric, sparse⟩(x.rows( ), x.rows( ));
1373      return *dest;
1374    }
```

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.

/* Not until **export** */

258. ⟨ functionalbase.h 258 ⟩ ≡

```
#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.

⟨ Functional base class methods 259 ⟩ ≡

```
  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.

⟨ Functional base class methods 259 ⟩ +≡

```
  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.

⟨ Functional base class methods 259 ⟩ +≡

```
1394    virtual matrix⟨T, unstructured, dense⟩ &grad(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, unstructured, dense⟩ &dest) ← 0;
```

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).

⟨ Functional base class methods 259 ⟩ +≡

```
1395    virtual matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, dense⟩ &dest) ← 0;
1396    virtual matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, sparse⟩ &dest) ← 0;
```

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 ⟩ ≡

```
1397 #ifndef __MATH_GAXPY_FUNCTIONAL__
1398 #define __MATH_GAXPY_FUNCTIONAL__ 1.00
1399 #include <math/algebra.h>
1400 #include <math/functionalbase.h>
1401   namespace math {
1402     namespace functional
1403     {
1404       template⟨matrix_simple_template⟩
1405       class gaxpy :public base⟨T⟩ {
1406         ⟨ Gaxpy functional internal variables 265 ⟩
1407       public:
1408         ⟨ Gaxpy functional class methods 266 ⟩
1409       };
1410     }
1411   }
1412 #endif
```

265. To compute a **gaxpy** operation we need two parameters, the a vector and the scalar b.

⟨ Gaxpy functional internal variables 265 ⟩ ≡

```
1413    matrix⟨T, structure, storage⟩ ay;
1414    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.

```
⟨ Gaxpy functional class methods 266 ⟩ ≡
1415    matrix⟨T, structure, storage⟩ &a(void) { return ay; }
1416    T &b(void) { return bee; }
1417    gaxpy(const matrix⟨T, structure, storage⟩ &A, const T &B)
1418    : ay(A), bee(B) { }
```

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.

```
⟨ Gaxpy functional class methods 266 ⟩ +≡
1419    virtual T eval(const matrix⟨T, unstructured, dense⟩ &x)
1420    { return dot(ay, x) + bee; }
```

268. The gradient is simply a.

```
⟨ Gaxpy functional class methods 266 ⟩ +≡
1421    virtual matrix⟨T, unstructured, dense⟩ &grad(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, unstructured, dense⟩ &dest)
1422    { return (dest ← ay); }
```

269. The hessian of an affine function is zero.

```
⟨ Gaxpy functional class methods 266 ⟩ +≡
1423    virtual matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, dense⟩ &dest)
1424    {
1425      dest.init(x.rows( ), x.rows( ));
1426      dest.fillwith(0);
1427      return dest;
1428    }
1429    virtual matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, sparse⟩ &dest)
1430    {
1431      dest ← matrix⟨T, symmetric, sparse⟩(x.rows( ), x.rows( ));
1432      return dest;
1433    }
```

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).

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

271. ⟨ functional/linear.h 271 ⟩ ≡

```
#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 272 ⟩
      public:
        ⟨ Linear combination of functionals class methods 273 ⟩
      };
    }
  }
#endif
```

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

⟨ Linear combination of functionals internal variables 272 ⟩ ≡

```
  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 273 ⟩ ≡

```
  linear(void) { }
  linear(base⟨T⟩ *elment, T weight ← 1) { add(element, 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⟩ *element, T weight ← 1)
  {
    for (int i ← 0; i ≠ element→size( ); ++i) add(element→get_term(i), weight * element→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 273 ⟩ +≡

```
  void pop_back(void)
  {
    elements.pop_back( );
  }
```

275. We will need to modify weights

```
⟨ Linear combination of functionals class methods 273 ⟩ +≡
1469  void set_weight(int i, T value) { elements[i].first ← value; }
1470  T get_weight(int i) const { return elements[i].first; }
1471  void set_term(int i, base⟨T⟩ *value) { elements[i].second ← value; }
1472  base⟨T⟩ *get_term(int i) const { return elements[i].second; }
1473  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.

```
⟨ Linear combination of functionals class methods 273 ⟩ +≡
1474  ~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.

```
⟨ Linear combination of functionals class methods 273 ⟩ +≡
1475  T eval(const matrix⟨T, unstructured, dense⟩ &x)
1476  {
1477    T result ← 0;
1478    for (iterator i ← elements.begin( ); i ≠ elements.end( ); ++i) result += i→first * i→second→eval(x);
1479    return result;
1480  }
```

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.

```
⟨ Linear combination of functionals internal variables 272 ⟩ +≡
1481  matrix⟨T, unstructured, dense⟩ aux;
```

279.

```
⟨ Linear combination of functionals class methods 273 ⟩ +≡
1482  matrix⟨T, unstructured, dense⟩ &grad(const matrix⟨T, unstructured, dense⟩
          &x, matrix⟨T, unstructured, dense⟩ &dest)
1483  {
1484    if (elements.size( ) ≡ 0) {
1485      dest.resize(x.rows( ), 1);
1486      dest.fillwith(0.0);
1487      return dest;
1488    }
1489    iterator i ← elements.begin( );
1490    i→second→grad(x, dest);
1491    if (i→first ≠ 1) dest *= i→first;
1492    for (i++; i ≠ elements.end( ); ++i) {
1493      i→second→grad(x, aux);
1494      if (i→first ≠ 1) aux *= i→first;
1495      dest += aux;
1496    }
1497    return dest;
1498  }
```

280. We follow the same algorithm for the Hessian, again testing for unity weights in order to save computation.

```
⟨ Linear combination of functionals internal variables 272 ⟩ +≡
1499    matrix⟨T, symmetric, dense⟩ Hdense;
1500    matrix⟨T, symmetric, sparse⟩ Hsparse;
```

281. ⟨ Linear combination of functionals class methods 273 ⟩ +≡

```
1501    matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, dense⟩ &dest)
1502    {
1503      if (elements.size( ) ≡ 0) {
1504        dest.resize(x.rows( ), x.rows( ));
1505        dest.fillwith(0.0);
1506        return dest;
1507      }
1508      iterator i ← elements.begin( );
1509      i→second→hess(x, dest);
1510      if (i→first ≠ 1) dest ∗= i→first;
1511      for (i++; i ≠ elements.end( ); ++i) {
1512        i→second→hess(x, Hdense);
1513        if (i→first ≠ 1) Hdense ∗= i→first;
1514        dest += Hdense;
1515      }
1516      return dest;
1517    }
1518    matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, sparse⟩ &dest)
1519    {
1520      if (elements.size( ) ≡ 0) {
1521        dest ← matrix⟨T, symmetric, sparse⟩(x.rows( ), x.rows( ));
1522        return dest;
1523      }
1524      iterator i ← elements.begin( );
1525      i→second→hess(x, dest);
1526      if (i→first ≠ 1) dest ∗= i→first;
1527      for (i++; i ≠ elements.end( ); ++i) {
1528        i→second→hess(x, Hsparse);
1529        if (i→first ≠ 1) Hsparse ∗= i→first;
1530        dest += Hsparse;
1531      }
1532      return dest;
1533    }
```

282. The quadratic functional. The next functional we implement is the quadratic. It computes the value of $f(x) = x^T Px + p^T x + \pi$. 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 export ∗/
```

283. ⟨ functional/quadratic.h 283 ⟩ ≡

```
#ifndef __MATH_QUADRATIC_FUNCTIONAL__
#define __MATH_QUADRATIC_FUNCTIONAL__ 1.00
#include <math/algebra.h>
#include <math/functionalbase.h>
  namespace math {
    namespace functional
    {
      template⟨class T, template⟨class⟩ class storage ← dense⟩
      class quadratic:public base⟨T⟩ {
        ⟨ Quadratic functional internal variables 284 ⟩
      public:
        ⟨ Quadratic functional methods 285 ⟩
      };
    }
  }
#endif
```

284. We first provide storage for the functional parameters. As a design decision, the vector p will be dense and unstructured.

⟨ Quadratic functional internal variables 284 ⟩ ≡

```
  matrix⟨T, symmetric, storage⟩ Pee;
  matrix⟨T, unstructured, dense⟩ pee;
  T Pi;
```

This code is used in section 283.

285. We provide means to change the parameters in the same way we did with the **gaxpy** functional.

⟨ Quadratic functional methods 285 ⟩ ≡

```
  matrix⟨T, symmetric, storage⟩ &P(void) { return Pee; }
  matrix⟨T, unstructured, dense⟩ &p(void) { return pee; }
  T &pi(void) { return Pi; }
  quadratic(const matrix⟨T, symmetric, storage⟩ &newP, const matrix⟨T, unstructured, dense⟩
        &newp, const T &newpi)
  : Pee(newP), pee(newp), Pi(newpi) { }
```

See also sections 286, 287, and 288.

This code is used in section 283.

286. Evaluating: we use the *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.

⟨ Quadratic functional methods 285 ⟩ +≡

```
  virtual T eval(const matrix⟨T, unstructured, dense⟩ &x)
  {
    T result ← dot(pee, x) + Pi;
    matrix⟨T, unstructured, dense⟩ y(Pee.rows( ), 1);
    y ← axmul(Pee, x, &y);
    result += dot(x, y);
    return result;
  }
```

287. The gradient is given by $2Px + p$.

```
⟨ Quadratic functional methods 285 ⟩ +≡
1566    virtual matrix⟨T, unstructured, dense⟩ &grad(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, unstructured, dense⟩ &dest)
1567    {
1568      return gaxpy(Pee, x * 2, &(dest.copyfrom(pee)));
1569    }
```

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

```
⟨ Quadratic functional methods 285 ⟩ +≡
1570    virtual matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, dense⟩ &dest)
1571    {
1572      dest ← Pee;
1573      return dest *= 2;
1574    }
1575    virtual matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, sparse⟩ &dest)
1576    {
1577      dest ← Pee;
1578      return dest *= 2;
1579    }
```

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.

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

290. ⟨ functional/norm2err.h 290 ⟩ ≡

```
1580  #ifndef __MATH_NORM2ERR_FUNCTIONAL__
1581  #define __MATH_NORM2ERR_FUNCTIONAL__
1582  #include <math/algebra.h>
1583  #include <math/function.h>
1584  #include <math/functionalbase.h>
1585    namespace math {
1586      namespace functional
1587      {
1588        template⟨class T, template⟨class⟩ class storage⟩
1589        class norm2err:public base⟨T⟩ {
1590          ⟨ Norm-2 error functional internal variables 291 ⟩
1591        public:
1592          ⟨ Norm-2 error functional class methods 292 ⟩
1593        };
1594      }
1595    }
1596  #endif
```

291. The internal matrix variables are the vector y, two auxiliar vectors for computing the value and gradient of the function, and an auxiliar matrix for computing the Hessian. The other internal variable is the approximating function.

⟨ Norm-2 error functional internal variables 291 ⟩ ≡

```
1597    matrix⟨T, unstructured, dense⟩ Y, aux, fval;
1598    matrix⟨T, symmetric, storage⟩ H;
1599    function::base⟨T⟩ *F;
```

This code is used in section 290.

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

⟨ Norm-2 error functional class methods 292 ⟩ ≡

```
1600    matrix⟨T, unstructured, dense⟩ &y(void) { return Y; }
1601    function::base⟨T⟩ *&f(void) { return F; }
1602    norm2err(const matrix⟨T, unstructured, dense⟩ &newy, function::base⟨T⟩ *newf ← 0)
1603    : Y(newy), F(newf) { }
1604    norm2err(function::base⟨T⟩ *newf ← 0)
1605    : F(newf) { }
```

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

This code is used in section 290.

293. Now to the eval function.

⟨ Norm-2 error functional class methods 292 ⟩ +≡

```
1606    virtual T eval(const matrix⟨T, unstructured, dense⟩ &x)
1607    {
1608      fval ← F→eval(x, &fval);
1609      fval −= Y;
1610      return dot(fval, fval);
1611    }
```

294. The gradient is given by

$$2Df(x)^T(f-y)$$

⟨ Norm-2 error functional class methods 292 ⟩ +≡

```
1612    virtual matrix⟨T, unstructured, dense⟩ &grad(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, unstructured, dense⟩ &dest)
1613    {
1614      fval ← F→eval(x, &fval);
1615      fval −= Y;
1616      F→jacobian(x, &aux);
1617      atxmul(aux, fval, &dest);
1618      dest *= 2;
1619      return dest;
1620    }
```

295. The Hessian is a little bit more complicated. It is given by

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

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

⟨ Norm-2 error functional class methods 292 ⟩ +≡

```
1621  virtual matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
          &x, matrix⟨T, symmetric, dense⟩ &dest)
1622  {
1623    outterp(F→jacobian(x, &aux), &dest);
1624    fval ← F→eval(x, &fval);
1625    fval −= Y;
1626    for (index i ← 1; i ≤ Y.rows(); ++i) {
1627      H ← F→hess(x, &H, i);
1628      H ∗= fval(i);
1629      dest += H;
1630    }
1631    dest ∗= 2;
1632    return dest;
1633  }
```

296. ⟨ Norm-2 error functional class methods 292 ⟩ +≡

```
1634  virtual matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
          &x, matrix⟨T, symmetric, sparse⟩ &dest)
1635  {
1636    outterp(F→jacobian(x, aux), &dest);
1637    fval ← F→eval(x, &fval);
1638    fval −= Y;
1639    for (index i ← 1; i ≤ Y.rows(); ++i) {
1640      H ← F→hess(x, &H, i);
1641      H ∗= fval(i);
1642      dest += H;
1643    }
1644    dest ∗= 2;
1645    return dest;
1646  }
```

297. 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;
  element_type f, g;
```

See also sections 302 and 304.

This code is used in section 298.

300. ⟨ Product of functionals class methods 300 ⟩ ≡

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

See also sections 301, 303, and 305.

This code is used in section 298.

301. ⟨ Product of functionals class methods 300 ⟩ +≡

```
  T eval(const matrix⟨T, unstructured, dense⟩ &x) { return f→eval(x) * g→eval(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→eval(x);
    g→grad(x, dest);
    dest *= f→eval(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 ⟩ +≡

```
matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
        &x, matrix⟨T, symmetric, dense⟩ &dest)
{
  f→hess(x, dest);
  dest *= g→eval(x);
  g→hess(x, Hdense);
  Hdense *= f→eval(x);
  dest += Hdense;
  f→grad(x, aux);
  g→grad(x, xua);
  dest += xyyx(aux, xua, &Hdense);
  return dest;
}
matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
        &x, matrix⟨T, symmetric, sparse⟩ &dest)
{
  f→hess(x, dest);
  dest *= g→eval(x);
  g→hess(x, Hsparse);
  Hsparse *= f→eval(x);
  dest += Hsparse;
  f→grad(x, aux);
  g→grad(x, xua);
  dest += xyyx(aux, xua, &Hsparse);
  return dest;
}
```

306. The ratio of two functionals.

/* Empty, waiting for **export** */

307. ⟨ functional/ratio.h 307 ⟩ ≡

```
#ifndef __MATH_RATIO_FUNCTIONAL__
#define __MATH_RATIO_FUNCTIONAL__ 1.00
#include <math/functionalbase.h>
  namespace math {
    namespace functional
    {
      template⟨class T⟩
      class ratio:public base⟨T⟩ {
        ⟨ Ratio of functionals internal variables 308 ⟩
      public:
        ⟨ Ratio of functionals class methods 309 ⟩
      };
    }
  }
#endif
```

308. ⟨ Ratio of functionals internal variables 308 ⟩ ≡

```
1717    typedef base⟨T⟩ *element_type;
1718    element_type f, g;
```

See also sections 310 and 312.

This code is used in section 307.

309. ⟨ Ratio of functionals class methods 309 ⟩ ≡

```
1719    ratio(element_type new_f, element_type new_g):f(new_f), g(new_g) { }
1720    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 ⟩ +≡

```
1721    matrix⟨T, unstructured, dense⟩ aux;
```

311. ⟨ Ratio of functionals class methods 309 ⟩ +≡

```
1722    matrix⟨T, unstructured, dense⟩ &grad(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, unstructured, dense⟩ &dest)
1723    {
1724      T result ← g¬eval(x);
1725      g¬grad(x, aux);
1726      aux *= −f¬eval(x)/result;
1727      f¬grad(x, dest);
1728      dest += aux;
1729      return dest /= result;
1730    }
```

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

⟨ Ratio of functionals internal variables 308 ⟩ +≡

```
1731    matrix⟨T, unstructured, dense⟩ xua;
1732    matrix⟨T, symmetric, dense⟩ Hdense;
1733    matrix⟨T, symmetric, sparse⟩ Hsparse;
```

313. ⟨ Ratio of functionals class methods 309 ⟩ +≡

```
1734  matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
          &x, matrix⟨T, symmetric, dense⟩ &dest)
1735  {
1736    T result ← g¬eval(x);
1737    T tluser ← f¬eval(x);
1738    f¬grad(x, aux);
1739    g¬grad(x, xua);
1740    outerp(xua, &dest);
1741    dest *= 2 * tluser/result;
1742    dest -= xyyx(aux, xua, &Hdense);
1743    g¬hess(x, Hdense);
1744    Hdense *= tluser;
1745    dest -= Hdense;
1746    dest /= result;
1747    dest += f¬hess(x, Hdense);
1748    return dest /= result;
1749  }
1750  matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
          &x, matrix⟨T, symmetric, sparse⟩ &dest)
1751  {
1752    T result ← g¬eval(x);
1753    T tluser ← f¬eval(x);
1754    f¬grad(x, aux);
1755    g¬grad(x, xua);
1756    outerp(xua, &dest);
1757    dest *= 2 * tluser/result;
1758    dest -= xyyx(aux, xua, &Hsparse);
1759    g¬hess(x, Hsparse);
1760    Hsparse *= tluser;
1761    dest -= Hsparse;
1762    dest /= result;
1763    dest += f¬hess(x, Hsparse);
1764    return dest /= result;
1765  }
```

314. The power functional.

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

315. ⟨ functional/power.h 315 ⟩ ≡

```
#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.

⟨ Power of a functional internal variables 316 ⟩ ≡

```
  typedef base⟨T⟩ *element_type;
  element_type f;
  double exponent;
```

See also section 321.

This code is used in section 315.

317.

⟨ Power of a functional class methods 317 ⟩ ≡

```
  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.

⟨ Power of a functional class methods 317 ⟩ +≡

```
  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. ⟨ Power of a functional class methods 317 ⟩ +≡

```
  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 $nf^{n-1}\nabla^2 f + n(n-1)f^{n-2}\nabla f \nabla f^T$.

```
⟨ Power of a functional internal variables 316 ⟩ +≡
1798    matrix⟨T, symmetric, dense⟩ Hdense;
1799    matrix⟨T, symmetric, sparse⟩ Hsparse;
1800    matrix⟨T, unstructured, dense⟩ aux;
```

322.
```
⟨ Power of a functional class methods 317 ⟩ +≡
1801    matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, dense⟩ &dest)
1802    {
1803      T result ← f→eval(x);
1804      if (result ≡ 0 ∧ exponent ≥ 2) {
1805        dest.resize(x.rows( ), x.rows( ));
1806        dest.fillwith(0.0);
1807        return dest;
1808      }
1809      if (result ≡ 0 ∧ exponent < 2) throw error::domain( );
1810      f→grad(x, aux);
1811      outerp(aux, &Hdense);
1812      Hdense *= (exponent − 1)/result;
1813      f→hess(x, dest);
1814      dest += Hdense;
1815      return dest *= pow(result, exponent − 1) * exponent;
1816    }
1817    matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, sparse⟩ &dest)
1818    {
1819      T result ← f→eval(x);
1820      if (result ≡ 0 ∧ exponent ≥ 2) {
1821        dest.resize(x.rows( ), x.rows( ));
1822        dest.fillwith(0.0);
1823        return dest;
1824      }
1825      if (result ≡ 0 ∧ exponent < 2) throw error::domain( );
1826      f→grad(x, aux);
1827      outerp(aux, &Hsparse);
1828      Hsparse *= (exponent − 1)/result;
1829      f→hess(x, dest);
1830      dest += Hsparse;
1831      return dest *= pow(result, exponent − 1) * exponent;
1832    }
```

323. The entropy of a functional.

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

324. ⟨ functional/entr.h 324 ⟩ ≡

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

325. ⟨ Entropy of a functional internal variables 325 ⟩ ≡

```
  typedef base⟨T⟩ *element_type;
  element_type f;
```

See also section 330.

This code is used in section 324.

326. ⟨ Entropy of a functional class methods 326 ⟩ ≡

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

See also sections 327, 329, and 331.

This code is used in section 324.

327. ⟨ Entropy of a functional class methods 326 ⟩ +≡

```
  T eval(const matrix⟨T, unstructured, dense⟩ &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. ⟨ Entropy of a functional class methods 326 ⟩ +≡

```
  matrix⟨T, unstructured, dense⟩ &grad(const matrix⟨T, unstructured, dense⟩
      &x, matrix⟨T, unstructured, dense⟩ &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 $-(1+\log f)\nabla^2 f-(1/f)\nabla f\nabla^T f$.

```
⟨ Entropy of a functional internal variables 325 ⟩ +≡
1865    matrix⟨T, symmetric, dense⟩ Hdense;
1866    matrix⟨T, symmetric, sparse⟩ Hsparse;
1867    matrix⟨T, unstructured, dense⟩ aux;
```

331.
```
⟨ Entropy of a functional class methods 326 ⟩ +≡
1868    matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, dense⟩ &dest)
1869    {
1870      T result ← f→eval(x);
1871      if (result ≤ 0) throw error::domain( );
1872      f→grad(x, aux);
1873      outerp(aux, &Hdense);
1874      Hdense /= (−result);
1875      f→hess(x, dest);
1876      dest *= −(::log(result) + 1);
1877      return dest += Hdense;
1878    }
1879    matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, sparse⟩ &dest)
1880    {
1881      T result ← f→eval(x);
1882      if (result ≤ 0) throw error::domain( );
1883      f→grad(x, aux);
1884      outerp(aux, &Hsparse);
1885      Hsparse /= (−result);
1886      f→hess(x, dest);
1887      dest *= −(::log(result) + 1);
1888      return dest += Hsparse;
1889    }
```

332. Relative entropy. The relative entropy (also known as Kullback Leibler distance) function is given by $x\log x/y$. 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 lenght $\mathbf{entr}(x)$. If we use y, however, we would need $\mathbf{entr}(x)+\mathbf{relentr}(x,y)$ to describe the variable.

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

333. ⟨ functional/relentr.h 333 ⟩ ≡

```
#ifndef __MATH_RELATIVE_ENTROPY_FUNCTIONAL__
#define __MATH_RELATIVE_ENTROPY_FUNCTIONAL__ 1.00
#include <math.h>
#include <math/functionalbase.h>
  namespace math { namespace functional {
    template⟨class T⟩
    class relentr : public base⟨T⟩ {
      ⟨ Relative entropy functional internal variables 334 ⟩
    public:
      ⟨ Relative entropy functional class methods 335 ⟩
    };
  } }
#endif
```

334. ⟨ Relative entropy functional internal variables 334 ⟩ ≡

```
  typedef base⟨T⟩ *element_type;
  element_type f, g;
```

See also sections 337 and 339.

This code is used in section 333.

335. ⟨ Relative entropy functional class methods 335 ⟩ ≡

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

See also sections 336, 338, and 340.

This code is used in section 333.

336. ⟨ Relative entropy functional class methods 335 ⟩ +≡

```
  T eval(const matrix⟨T, unstructured, dense⟩ &x)
  {
    T fval ← f→eval(x);
    T gval ← g→eval(x);
    if (gval ≤ 0 ∨ fval ≤ 0) throw error::domain( );
    return fval * ::log(fval/gval);
  }
```

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

⟨ Relative entropy functional internal variables 334 ⟩ +≡

```
  matrix⟨T, unstructured, dense⟩ auxvec;
```

338. ⟨ Relative entropy functional class methods 335 ⟩ +≡

```
matrix⟨T, unstructured, dense⟩ &grad(const matrix⟨T, unstructured, dense⟩
        &x, matrix⟨T, unstructured, dense⟩ &dest)
{
  T fval ← f¬eval(x);
  T gval ← g¬eval(x);
  if (gval ≤ 0 ∨ fval ≤ 0) throw error::domain( );
  g¬grad(x, auxvec);
  auxvec *= fval/gval;
  f¬grad(x, dest);
  dest *= (::log(fval/gval) + 1);
  return dest −= auxvec;
}
```

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

⟨ Relative entropy functional internal variables 334 ⟩ +≡

```
matrix⟨T, unstructured, dense⟩ auxgrad;
matrix⟨T, symmetric, dense⟩ auxdense;
matrix⟨T, symmetric, sparse⟩ auxsparse;
```

340. ⟨ Relative entropy functional class methods 335 ⟩ +≡

```
matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
        &x, matrix⟨T, symmetric, dense⟩ &dest)
{
#define auxhess auxdense
  ⟨ relentr Hessian 341 ⟩;
#undef auxhess
}
matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
        &x, matrix⟨T, symmetric, sparse⟩ &dest)
{
#define auxhess auxsparse
  ⟨ relentr Hessian 341 ⟩;
#undef auxhess
}
```

341. ⟨ **relentr** Hessian 341 ⟩ ≡

```
T fval ← f→eval(x);
T gval ← g→eval(x);
if (gval ≤ 0 ∨ fval ≤ 0) throw error::domain( );
g→grad(x, auxvec);
f→grad(x, auxgrad);
auxvec /= gval;
auxgrad /= fval;
outerp(auxvec -= auxgrad, &dest);
dest *= fval;
g→hess(x, auxhess);
auxhess *= fval/gval;
dest -= auxhess;
f→hess(x, auxhess);
auxhess *= (::log(fval/gval) + 1);
return dest += auxhess;
```

This code is used in section 340.

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

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-u^2} du$$

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

343. ⟨ `functional/erf.h` 343 ⟩ ≡

```
#ifndef __MATH_ERF_FUNCTIONAL__
#define __MATH_ERF_FUNCTIONAL__ 1.00
#include <math.h>
#include <math/functionalbase.h>
  namespace math { namespace functional {
    template⟨class T⟩
    class erf:public base⟨T⟩ {
      ⟨ Error function of a functional internal variables 344 ⟩
    public:
      ⟨ Error function of a functional class methods 345 ⟩
    };
  } }
#endif
```

344. ⟨ Error function of a functional internal variables 344 ⟩ ≡

```
typedef base⟨T⟩ *element_type;
element_type f;
```

See also section 349.

This code is used in section 343.

345. ⟨ Error function of a functional class methods 345 ⟩ ≡

```
erf(element_type new_f):f(new_f) { }
```

See also sections 346, 348, and 350.

This code is used in section 343.

346. ⟨ Error function of a functional class methods 345 ⟩ +≡

```
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. ⟨ Error function of a functional class methods 345 ⟩ +≡

```
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 *= 2/::sqrt(M_PI) * ::exp(−result * result);
}
```

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

⟨ Error function of a functional internal variables 344 ⟩ +≡

```
matrix⟨T, symmetric, dense⟩ dense_aux;
matrix⟨T, symmetric, sparse⟩ sparse_aux;
matrix⟨T, unstructured, dense⟩ aux;
```

350. ⟨ Error function of a functional class methods 345 ⟩ +≡

```
matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
    &x, matrix⟨T, symmetric, dense⟩ &dest)
{
#define hess_aux dense_aux
  ⟨ Compute Hessian for erf functional 351 ⟩;
#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 erf functional 351 ⟩;
#undef hess_aux
}
```

351. ⟨ Compute Hessian for **erf** functional 351 ⟩ ≡

```
T result ← f→eval(x);
f→grad(x, aux);
f→hess(x, dest);
outerp(aux, &hess_aux);
hess_aux *= −2 * result;
dest += hess_aux;
return dest *= 2 * ::exp(−result * result)/::sqrt(M_PI);
```

This code is used in section 350.

352. The exponential of a functional.

/* Empty, waiting for **export** */

353. ⟨ functional/exp.h 353 ⟩ ≡

```
#ifndef __MATH_EXP_FUNCTIONAL__
#define __MATH_EXP_FUNCTIONAL__ 1.00
#include <math.h>
#include <math/functionalbase.h>
  namespace math {
    namespace functional
    {
      template⟨class T⟩
      class exp:public base⟨T⟩ {
        ⟨ Exponential of a functional internal variables 354 ⟩
      public:
        ⟨ Exponential of a functional class methods 355 ⟩
      };
    }
  }
#endif
```

354. ⟨ Exponential of a functional internal variables 354 ⟩ ≡

```
  typedef base⟨T⟩ *element_type;
  element_type f;
```

See also section 359.

This code is used in section 353.

355. ⟨ Exponential of a functional class methods 355 ⟩ ≡

```
  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 ⟩ +≡

```
  T eval(const matrix⟨T, unstructured, dense⟩ &x)
  {
    return ::exp(f¬eval(x));
  }
```

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

358. ⟨ Exponential of a functional class methods 355 ⟩ +≡

```
  matrix⟨T, unstructured, dense⟩ &grad(const matrix⟨T, unstructured, dense⟩
        &x, matrix⟨T, unstructured, dense⟩ &dest)
  {
    f¬grad(x, dest);
    return dest *= eval(x);
  }
```

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

⟨ Exponential of a functional internal variables 354 ⟩ +≡

```
matrix⟨T, symmetric, dense⟩ Hdense;
matrix⟨T, symmetric, sparse⟩ Hsparse;
matrix⟨T, unstructured, dense⟩ aux;
```

360. ⟨ Exponential of a functional class methods 355 ⟩ +≡

```
matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
        &x, matrix⟨T, symmetric, dense⟩ &dest)
{
  f¬grad(x, aux);
  outerp(aux, &Hdense);
  f¬hess(x, dest);
  dest += Hdense;
  return dest *= eval(x);
}
matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
        &x, matrix⟨T, symmetric, sparse⟩ &dest)
{
  f¬grad(x, aux);
  outerp(aux, &Hsparse);
  f¬hess(x, dest);
  dest += Hsparse;
  return dest *= eval(x);
}
```

361. The logarithm of a functional.

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

362. ⟨ functional/log.h 362 ⟩ ≡

```
#ifndef __MATH_LOG_FUNCTIONAL__
#define __MATH_LOG_FUNCTIONAL__ 1.00
#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 363 ⟩
      public:
        ⟨ Logarithm of a functional class methods 364 ⟩
      };
    }
  }
#endif
```

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

⟨ Logarithm of a functional internal variables 363 ⟩ ≡

```
2067    typedef base⟨T⟩ *element_type;
2068    element_type f;
```

See also section 368.

This code is used in section 362.

364.

⟨ Logarithm of a functional class methods 364 ⟩ ≡

```
2069    log(element_type new_f):f(new_f) { }
```

See also sections 365, 367, and 369.

This code is used in section 362.

365.

⟨ Logarithm of a functional class methods 364 ⟩ +≡

```
2070    T eval(const matrix⟨T, unstructured, dense⟩ &x)
2071    {
2072      T result ← ::log(f→eval(x));
2073      if (¬finite(result)) throw error::domain( );
2074      return result;
2075    }
```

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

367. ⟨ Logarithm of a functional class methods 364 ⟩ +≡

```
2076    matrix⟨T, unstructured, dense⟩ &grad(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, unstructured, dense⟩ &dest)
2077    {
2078      T result ← f→eval(x);
2079      f→grad(x, dest);
2080      return dest /= result;
2081    }
```

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

⟨ Logarithm of a functional internal variables 363 ⟩ +≡

```
2082    matrix⟨T, symmetric, dense⟩ Hdense;
2083    matrix⟨T, symmetric, sparse⟩ Hsparse;
2084    matrix⟨T, unstructured, dense⟩ aux;
```

369. ⟨ Logarithm of a functional class methods 364 ⟩ +≡

```
2085    matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, dense⟩ &dest)
2086    {
2087      T result ← f→eval(x);
2088      f→grad(x, aux);
2089      outerp(aux, &Hdense);
2090      Hdense /= (−result);
2091      f→hess(x, dest);
2092      dest += Hdense;
2093      return dest /= result;
2094    }
2095    matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
            &x, matrix⟨T, symmetric, sparse⟩ &dest)
2096    {
2097      T result ← f→eval(x);
2098      f→grad(x, aux);
2099      outerp(aux, &Hsparse);
2100      Hdense /= (−result);
2101      f→hess(x, dest);
2102      dest += Hsparse;
2103      return dest /= result;
2104    }
```

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).

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

371. ⟨ linesearchbase.h 371 ⟩ ≡

```
#ifndef __MATH_LINESEARCH__
#define __MATH_LINESEARCH__ 1.0
#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).

⟨ Line search base class methods 372 ⟩ ≡

```
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⟩ &, 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 $\|x - x^*\| \leq tol$, 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.

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

374. ⟨ linesearch/bisection.h 374 ⟩ ≡

```
#ifndef __MATH_BISECTION__
#define __MATH_BISECTION__ 1.0
#include <math.h>
  const double bisection_infty ← HUGE_VAL;
    /* Some compilers have problems inlining this constant. */
#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 ⟩;
    }
  }
#endif
```

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.

⟨ Bisection line search methods 375 ⟩ ≡

```
  bisection(double t ← 1 · 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.

⟨ Bisection line search methods 375 ⟩ +≡

```
  matrix⟨T, unstructured, dense⟩ minimize(functional::base⟨T⟩ *f,
          const matrix⟨T, unstructured, dense⟩ &x0, const matrix⟨T, unstructured, dense⟩
          &g0, const matrix⟨T, unstructured, dense⟩ &dir)
  {
    return minimize(f, x0, dir);
  }
```

377. The algorithm works as follows: we make $x \leftarrow x0$. 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^* and x since

$$\|x - x^*\| \leq \|x - (x + \delta x)\| = \|\delta x\|,$$

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:

```
⟨ Bisection line search methods 375 ⟩ +≡
2150  matrix⟨T, unstructured, dense⟩ minimize(functional::base⟨T⟩ *func, const matrix⟨T,
        unstructured, dense⟩ &x0, const matrix⟨T, unstructured, dense⟩ &dir);
```

378. ⟨ **bisection** big definitions 378 ⟩ ≡

```
2151  template⟨class T⟩ matrix⟨T, unstructured, dense⟩ bisection⟨T⟩::minimize(functional::base⟨T⟩
          *func, const matrix⟨T, unstructured, dense⟩ &x0, const matrix⟨T, unstructured, dense⟩
          &dir){ double stepsize ← 1, dirnorm ← norm2(dir);
2152    matrix⟨T, unstructured, dense⟩ x(x0);
2153    unsigned long outer_iter ← 0; while ( fabs ( stepsize / ratio )
          *dirnorm > tol ∧ ++outer_iter ≤ maxiter ) { double f ← func→eval(x);
2154    double newf;
2155    ⟨ Compute next function value for bisection 379 ⟩;
2156    unsigned long inner_iter ← 0;
2157    while (newf < f) {
2158      f ← newf;
2159      ⟨ Compute next function value for bisection 379 ⟩;
2160      if (++inner_iter > maxiter) throw error::unboundedbelow( );
2161    }
2162    stepsize *= −ratio; } return x; }
```

This code is used in section 374.

379. ⟨ Compute next function value for **bisection** 379 ⟩ ≡

```
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 λ, 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 α and β, where $0 < \alpha < 0.5$ and $0 < \beta < 1$.

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

381. ⟨ linesearch/backtracking.h 381 ⟩ ≡

```
#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.

```
⟨ Backtracking line search methods 382 ⟩ ≡
2201    backtracking(double a ← 0.3, double b ← 0.8, unsigned long m ← 1000)
2202    : alpha(a), beta(b), maxiter(m) { }
```

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.

```
⟨ Backtracking line search methods 382 ⟩ +≡
2203    matrix⟨T, unstructured, dense⟩ minimize(functional::base⟨T⟩ ∗f, const matrix⟨T,
            unstructured, dense⟩ &x0, const matrix⟨T, unstructured, dense⟩ &dir)
2204    {
2205      matrix⟨T, unstructured, dense⟩ g0;
2206      g0 ← f¬grad(x0, g0);
2207      return minimize(f, x0, g0, dir);
2208    }
```

384.

```
⟨ Backtracking line search methods 382 ⟩ +≡
2209    matrix⟨T, unstructured, dense⟩ minimize(functional::base⟨T⟩ ∗func,
            const matrix⟨T, unstructured, dense⟩ &x0, const matrix⟨T, unstructured, dense⟩
            &g0, const matrix⟨T, unstructured, dense⟩ &dir)
2210    {
2211      matrix⟨T, unstructured, dense⟩ x;
2212      double ftreshold ← func¬eval(x0);
2213      double gtreshold ← alpha ∗ dot(g0, dir);
2214      double stepsize ← 1/beta;
2215      unsigned long iter ← 0;
2216      double fval;
2217  #ifdef __MATH_DEBUG__
2218      cout ≪ "[math]: backtracking line search begin.\n";
2219      cout ≪ "[math]: x0 = ";
2220      for (index i ← 1; i ≤ x0.rows(); ++i) cout ≪ x0(i) ≪ ' ';
2221      cout ≪ '\n';
2222      cout ≪ "[math]: dir = ";
2223      for (index i ← 1; i ≤ dir.rows(); ++i) cout ≪ dir(i) ≪ ' ';
2224      cout ≪ '\n';
2225  #endif
2226      do {
2227        ⟨ Compute next function value for backtracking 385 ⟩;
2228        if (++iter ≡ maxiter) throw error::maxiterations();
2229      } while (fval > ftreshold + stepsize ∗ gtreshold);
2230  #ifdef __MATH_DEBUG__
2231      cout ≪ "[math]: backtracking line search end.\n";
2232  #endif
2233      return x;
2234    }
```

385. ⟨ Compute next function value for **backtracking** 385 ⟩ ≡

```
2235    fval ← backtracking_infty;
2236    while (¬finite(fval)) {
2237      x ← dir;
2238      x *= (stepsize *= beta);
2239      x += x0;
2240  #ifdef __MATH_DEBUG__
2241      cout ≪ "[math]:␣x␣=␣";
2242      for (index i ← 1; i ≤ x.rows(); ++i) cout ≪ x(i) ≪ '␣';
2243      cout ≪ '\n';
2244  #endif
2245      try {
2246        fval ← func→eval(x);
2247      }
2248      catch(error::domain e)
2249      {
2250        fval ← backtracking_infty;
2251      }
2252  #ifdef __MATH_DEBUG__
2253      cout ≪ "[math]:␣f(x)=" ≪ fval ≪ ",␣stepsize=" ≪ stepsize ≪ ",␣treshold=" ≪
            ftreshold + stepsize * gtreshold ≪ '\n';
2254  #endif
2255      if (stepsize ≡ 0.0) fval ← ftreshold + stepsize * gtreshold;
2256    }
```

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.

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

387. ⟨ searchdirbase.h 387 ⟩ ≡

```
#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.

⟨ Search direction base class methods 388 ⟩ ≡

```
  virtual matrix⟨T, unstructured, dense⟩ &dir(functional::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 costy for you, then the gradient search direction can be a good choice.

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

390. ⟨ searchdir/gradient.h 390 ⟩ ≡

```
#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 ⟩
      };
    }
  }
#endif
```

391. The search direction is basically $-\nabla f$.

⟨ 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;
  }
```

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. ⟨ searchdir/newton.h 393 ⟩ ≡

```
#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.

⟨ Newton search direction internal variables 394 ⟩ ≡

```
  matrix⟨T, symmetric, storage⟩ hess;
```

This code is used in section 393.

395. The search direction is now $-\left(\nabla^2 f\right)^{-1}\nabla f$. 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).

⟨ Newton search direction methods 395 ⟩ ≡

```
  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) { f→grad(x, *dest); }
    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 ⟩ ≡

```
#ifndef __MATH_SEARCHDIR_EQUALITY__
#define __MATH_SEARCHDIR_EQUALITY__ 1.0
#include <math/searchdirbase.h>
#include <math/symmetric.h>
#include <math/lu.h>
  namespace math {
    namespace searchdir
    {
      template⟨matrix_simple_template⟩
      class equality:public base⟨T⟩ {
        ⟨ Equality search direction internal variables 398 ⟩
      public:
        ⟨ Equality search direction methods 399 ⟩
      };
    }
  }
#endif
```

398. First we need a way to let the user define which basic search direction he wants to use.

⟨ Equality search direction internal variables 398 ⟩ ≡

```
  base⟨T⟩ *dirf;
```

See also section 400.

This code is used in section 397.

399. ⟨ Equality search direction methods 399 ⟩ ≡

```
  equality(void):dirf(0) { }
  equality(base⟨T⟩ *newf):dirf(newf) { }
  base⟨T⟩ *&f(void) { return dirf; }
```

See also sections 401 and 403.

This code is used in section 397.

400. 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 acomplished 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

$$d = (I - A^T \big(AA^T\big)^{-1} A) d_0$$

. 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.

⟨ Equality search direction internal variables 398 ⟩ +≡

```
2353    matrix⟨T, unstructured, storage⟩ decomp;
2354    vector⟨index⟩ pivots;
2355    index xrows;
```

401. ⟨ Equality search direction methods 399 ⟩ +≡

```
2356    equality(const matrix⟨T, structure, storage⟩ &A):dirf(0) { set_A(newa); }
2357    void set_A(const matrix⟨T, structure, storage⟩ &A)
2358    {
2359  #ifdef __MATH_DEBUG__
2360      cout ≪ "[math]: equality search constructor.\n";
2361      for (index i ← 1; i ≤ A.rows(); ++i) {
2362        for (index j ← 1; j ≤ A.cols(); ++j) cout ≪ A(i, j) ≪ ' ';
2363        cout ≪ '\n';
2364      }
2365  #endif
2366      xrows ← A.cols();
2367      decomp.resize(A.cols() + A.rows(), A.cols() + A.rows());
2368      decomp.fillwith(0.0);
2369      decomp.subm(1, A.cols(), 1 + A.cols(), decomp.cols()) ← transpose(A);
2370      decomp.subm(1 + A.cols(), decomp.rows(), 1, A.cols()) ← A;
2371      for (index i ← 1; i ≤ A.cols(); ++i) decomp.entry(i, i) ← 1;
2372      lu :: decompose(&decomp, &pivots);
2373    }
```

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 ⟩ +≡

```
2374  matrix⟨T, unstructured, dense⟩ &dir(functional::base⟨T⟩ *f, const matrix⟨T, unstructured,
          dense⟩ &x, matrix⟨T, unstructured, dense⟩ *dest)
2375  {
2376    double extra_component ← 0;
2377    dest→resize(decomp.rows( ), 1);
2378    dest→fillwith(0.0);
2379    dirf→dir(f, x, dest);
2380  #ifdef __MATH_DEBUG__
2381    cout ≪ "[math]: equality search dir begin.\n[math]: orig dir = ";
2382    for (index i ← 1; i ≤ x.rows( ); ++i) cout ≪ (*dest)(i) ≪ ' ';
2383    cout ≪ '\n';
2384  #endif
2385    if (x.rows( ) ≠ xrows) extra_component ← (*dest)(xrows + 1);
2386    dest→entry(xrows + 1) ← 0.0;
2387    lu::finish(decomp, pivots, dest);
2388    dest→resize(x.rows( ), 1);
2389    if (x.rows( ) ≠ xrows) dest→entry(xrows + 1) ← extra_component;
2390  #ifdef __MATH_DEBUG__
2391    cout ≪ "[math]: new dir = ";
2392    for (index i ← 1; i ≤ result.rows( ); ++i) cout ≪ (*dest)(i) ≪ ' ';
2393    cout ≪ "\n[math]: equality search dir end.\n";
2394  #endif
2395    return *dest;
2396  }
```

404. Newton direction with equality constraints.

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

405. ⟨ searchdir/equality/newton.h 405 ⟩ ≡

```
2397  #ifndef __MATH_SEARCHDIR_EQUALITY_NEWTON__
2398  #define __MATH_SEARCHDIR_EQUALITY_NEWTON__
2399  #include <math/math.h>
2400  #include <math/searchdirbase.h>
2401  #include <math/lu.h>
2402    namespace math { namespace searchdir { namespace equalty { template ⟨class T, template
          ⟨ class > class storage > class newton : public base⟨T⟩ {
2403    ⟨ Newton with equality internal variables 406 ⟩;
2404    public: ⟨ Newton with equality functions 408 ⟩;
2405    }; } } }
2406  #endif
```

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.

⟨ Newton with equality internal variables 406 ⟩ ≡

```
2407  matrix⟨T, unstructured, storage⟩ A;
2408  matrix⟨T, symmetric, storage⟩ H;
2409  matrix⟨T, unstructured, storage⟩ M;
```

See also section 407.

This code is used in section 405.

407. Of course, we also face the same auxiliary variable problem.

⟨ Newton with equality internal variables 406 ⟩ +≡

```
2410    bool has_t;
2411    index xrows;      /* Original number of variables. */
```

408. ⟨ Newton with equality functions 408 ⟩ ≡

```
matrix⟨T, unstructured, dense⟩ &dir(functional::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]: got Hessian.\n";
    cout ≪ "H=\n";
    for (index i ← 1; i ≤ H.rows(); ++i) {
      cout ≪ "row " ≪ i ≪ ": ";
      for (index j ← 1; j ≤ H.cols(); ++j) cout ≪ H(i, j) ≪ ' ';
      cout ≪ '\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);
      }
    }
    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);
      }
#ifdef __MATH_DEBUG__
    cout ≪ "[math]: about to get search direction.\n";
    cout ≪ "M=\n";
    for (index i ← 1; i ≤ M.rows(); ++i) {
      for (index j ← 1; j ≤ M.cols(); ++j) cout ≪ M(i, j) ≪ ' ';
      cout ≪ '\n';
    }
    cout ≪ "dest=\n";
    for (index i ← 1; i ≤ dest→rows(); ++i) cout ≪ dest→get(i, 1) ≪ ' ';
    cout ≪ '\n';
#endif
    lu::solve(M, *dest);
```

```
        dest→resize(x.rows( ), 1);
#ifdef __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;
    }
```

See also section 409.

This code is used in section 405.

409. Now we provide a function to set the A matrix.

```
⟨ Newton with equality functions 408 ⟩ +≡
    void set_A(const matrix⟨T, unstructured, storage⟩ &a)
    {
      has_t ← false;
      xrows ← a.cols( );
      A ← a;
    }
```

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

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \prec 0, \qquad i = 1, \ldots, p \\ & g_i(x) = 0, \qquad i = 1, \ldots, q \end{array}$$

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 $f_i(x) \succeq 0$. With this property, we see that the minimum value of $f_0(x) + \Sigma\phi_i(x)$ 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 $tf_0(x) + \Sigma\phi_i(x)$ 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:

$$|x_i - x_i^*| \leq |x_i^*| reltol + abstol$$
$$\|\nabla f(x)\| \leq abstol.$$

Of course, the first condition only makes sense when $abstol \cdot reltol = 0$. 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: *abstol* and *reltol* have the meaning of that first stopping condition described above. They can be both positive, in which case the optimization stops when *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. ⟨ fmin.h 412 ⟩ ≡

```
#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
          reltol ← 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 *grad* (which is otherwise not used).

⟨ Functional minimization algorithm 413 ⟩ ≡

```
  {
    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).

⟨ Update functional minimization stop criteria 414 ⟩ ≡

```
  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)) ∗ reltol);
  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. ⟨ barrierbase.h 416 ⟩ ≡

```
#ifndef __MATH_BARRIER__
#define __MATH_BARRIER__ 1.0
#include <math/functionalbase.h>
#include <math/functionbase.h>
  namespace math {
    namespace barrier
    {
      template⟨class T⟩ classbase: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 ⟩ ≡

```
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 ⟩ ≡

```
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) { }
```

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 ⟩ +≡

```
bool has_t;
```

421. The *addt* function returns a value of t that satisfies $f(x) \le t$.

⟨ Barrier function methods 418 ⟩ +≡

```
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;
}
```

422. ⟨ Barrier function methods 418 ⟩ +≡

```
2559   void delt(void) { has_t ← false; }
```

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

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) < 0 \end{array}$$

and we will sequentially solve

$$\text{minimize} \quad \mu f_0(x) + \sum \phi(f_i(x))$$

for various values of μ. 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 λ_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 λ_i, that is, λ_i are feasible dual variables and can provide a lower bound on the optimal value. It is clear that the λ_i will be dependent on μ, but the barrier function doesn't know its value. What we do then is to return $\mu\lambda_i$, which is independent of μ.

⟨ Barrier function methods 418 ⟩ +≡

```
2560   virtual T dual(const matrix⟨T, unstructured, dense⟩ &) ← 0;
```

424. On other ocasious it will be useful to get the value of the dual variable times the function itself, and it is sometimes much cheaper to compue this value (see, for example, the log barrier function). The same above remarks with respect to μ are valid here, that is, the function should return $\mu\lambda_i f(x)$.

⟨ Barrier function methods 418 ⟩ +≡

```
2561   virtual T dual_times_f(const matrix⟨T, unstructured, dense⟩ &) ← 0;
```

425. The log barrier function. Perhaps the most common, its value is $-\log(-f(x))$.

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

426. ⟨ barrier/log.h 426 ⟩ ≡

```
2562 #ifndef __MATH_LOG_BARRIER__
2563 #define __MATH_LOG_BARRIER__
2564 #include <math/barrierbase.h>
2565   namespace math {
2566     namespace barrier
2567     {
2568       template⟨class T⟩ classlog:public base⟨T⟩
2569       {
2570         matrix⟨T, unstructured, dense⟩ aux, jac;
2571         matrix⟨T, symmetric, dense⟩ dense_aux;
2572         matrix⟨T, symmetric, sparse⟩ sparse_aux;
2573       public:
2574         ⟨ Log barrier methods 427 ⟩
2575       }
2576       ;
2577     }
2578   }
2579 #endif
```

427. ⟨ Log barrier methods 427 ⟩ ≡

```
log(functional::base⟨T⟩ *newf) { f(newf); }
log(function::base⟨T⟩ *newf) { f(newf); }
```

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.

⟨ Log barrier methods 427 ⟩ +≡

```
T eval(const matrix⟨T, unstructured, dense⟩ &x)
{
  T result ← 0, 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) {
    result ← fctnal→eval(X) − t;
    if (result ≥ 0) throw error::domain( );
    return −::log(−result);
  }
  fct→eval(X, aux);
  for (index i ← 1; i ≤ aux.rows( ); ++i) {
    if (aux(i) − t ≥ 0) throw error::domain( );
    result −= ::log(t − aux(i));
  }
  return result;
}
```

429. For the logarithm, the dual variable is $-1/(\mu f(x))$, so we return $-1/f(x)$.

⟨ Log barrier methods 427 ⟩ +≡

```
T dual(const matrix⟨T, unstructured, dense⟩ &x) { return −1/eval(x); }
T dual_times_f(const matrix⟨T, unstructured, dense⟩ &) { 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 $-\sum_i(\nabla f_i)/f_i$.

```
⟨ Log barrier methods 427 ⟩ +≡
2604  matrix⟨T, unstructured, dense⟩ &grad(const matrix⟨T, unstructured, dense⟩
          &x, matrix⟨T, unstructured, dense⟩ &dest)
2605  {
2606    T t ← 0;
2607    matrix⟨T, unstructured, dense⟩ X ← x;      /* We may resize x. */
2608    if (has_t) {
2609      t ← X(X.rows( ));
2610      X.resize(X.rows( ) − 1, 1);
2611    }
2612    if (fctnal) {
2613      fctnal→grad(X, dest);
2614      if (has_t) {
2615        dest.resize(X.rows( ) + 1, 1);
2616        dest.entry(dest.rows( )) ← T(−1);
2617      }
2618      dest /= t − fctnal→eval(X);
2619      return dest;
2620    }
2621    fct→eval(X, aux);
2622    fct→jacobian(X, dest);
2623    if (has_t) {
2624      dest.resize(dest.rows( ), dest.cols( ) + 1);
2625      dest.subm(1, dest.rows( ), dest.cols( )) ← matrix⟨T, unstructured, dense⟩(dest.rows( ), 1, T(−1));
2626    }
2627    for (index i ← 1; i ≠ aux.rows( ); ++i) {
2628      dest.subm(i, i, 1, dest.cols( )) /= t − aux(i);
2629      if (i ≠ 1) dest.subm(1, 1, 1, dest.cols( )) += dest.subm(i, i, 1, dest.cols( ));
2630    }
2631    dest ← transpose(dest.subm(1, 1, 1, dest.cols( )));
2632    return dest;
2633  }
```

431. Finally the Hessian: for a functional we have $\nabla^2 -\log(-f) = (\nabla f \nabla^T f)/f^2 - (\nabla f)^2/f$.

```
⟨ Log barrier methods 427 ⟩ +≡
2634  matrix⟨T, symmetric, dense⟩ &hess(const matrix⟨T, unstructured, dense⟩
          &x, matrix⟨T, symmetric, dense⟩ &dest)
2635  {
2636 #define hess_aux dense_aux
2637    ⟨ Compute Hessian for log barrier 432 ⟩;
2638 #undef hess_aux
2639  }
2640  matrix⟨T, symmetric, sparse⟩ &hess(const matrix⟨T, unstructured, dense⟩
          &x, matrix⟨T, symmetric, sparse⟩ &dest)
2641  {
2642 #define hess_aux sparse_aux
2643    ⟨ Compute Hessian for log barrier 432 ⟩;
2644 #undef hess_aux
2645  }
```

432. ⟨ Compute Hessian for log barrier 432 ⟩ ≡

```
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;
```

This code is used in section 431.

433. Sequential unconstrained minimization. function defined, we are ready to tackle our main problem, namely

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \prec 0, \qquad i = 1, \ldots, p \\ & g_i(x) = 0, \qquad i = 1, \ldots, q. \end{array}$$

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 ϕ_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. ⟨ `sumt.h` 434 ⟩ ≡

```
#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.

⟨ SUMT functions 435 ⟩ ≡

```
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.

⟨ Build unconstrained objective function 436 ⟩ ≡

```
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 a 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.

⟨ SUMT Phase one 438 ⟩ ≡

```
2727    T aux_var ← 0;
2728    for (sumt_iterator i ← barrier_functions.begin( ); i ≠ barrier_functions.end( ); ++i)
          aux_var ← max(aux_var, (∗i)→addt(x));
2729    aux_var ← 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.

⟨ SUMT Phase one 438 ⟩ +≡

```
2730    x.resize(x.rows( ) + 1, 1);
2731    x.entry(x.rows( )) ← aux_var;
2732    matrix⟨T, unstructured, dense⟩ phasei_cost(x.rows( ), 1);
2733    phasei_cost.entry(x.rows( )) ← 1.0;
2734    functional::gaxpy⟨T, unstructured, dense⟩ phasei(phasei_cost, 0.0);
2735    f.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.

```
⟨ SUMT Phase one 438 ⟩ +≡
2736    mu ← 1;
2737    T cost ← 2 ∗ abstol;
2738    T lagrange ← 0; while (cost − lagrange ≥ abstol ∧ x(x.rows( )) > −0.1) { f.set_weight(0, mu); try {
                x ← fmin (&f, x, line , dir, abstol, reltol, 0.0 ) ; } catch(error::maxiterations e)
2739    { }
2740    cost ← x(x.rows( ));
2741    lagrange ← cost;
2742    for (int i ← 1; i ≠ f.size( ); ++i)
          lagrange += static_cast⟨barrier::base⟨T⟩ ∗⟩(f.get_term(i))→dual_times_f(x)/mu;
2743    if (disp) disp(1, cost, cost − lagrange);
2744    if (lagrange ≥ 0) throw error::infeasible( );
2745    mu ← mu ∗ 50; }
2746    if (x(x.rows( )) ≥ 0) throw error::infeasible( );
2747    x.resize(x.rows( ) − 1, 1);
2748    for (sumt_iterator i ← barrier_functions.begin( ); i ≠ barrier_functions.end( ); ++i) (∗i)→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 procceed almost exactly as in phase I.

```
⟨ SUMT Phase two 441 ⟩ ≡
2749    if (¬obj) return;      /∗ It was a feasibility problem. ∗/
2750    f.set_term(0, obj);
2751    bool stop ← false;
2752    mu ← 1; while (¬stop) { f.set_weight(0, mu); x ← fmin (&f, x, line , dir, abstol, reltol, 0.0 ) ;
2753    T cost ← obj→eval(x);
2754    T slack ← 0;
2755    for (int i ← 1; i ≠ f.size( ); ++i)
          slack −= static_cast⟨barrier::base⟨T⟩ ∗⟩(f.get_term(i))→dual_times_f(x)/mu;
2756    if (disp) disp(2, cost, slack);
2757    mu ← mu ∗ 50;
2758    stop ← true;
2759  #ifdef __MATH_DEBUG
2760    cout ≪ "[math]:";
2761    cout ≪ "slack=" ≪ slack ≪ ",␣cost=" ≪ cost ≪ ",␣abstol=" ≪ abstol ≪ ",␣reltol=" ≪ reltol ≪
          '\n';
2762  #endif
2763    if (abstol) stop ← (slack ≤ abstol);
2764    if (reltol ∧ cost ∧ stop) stop ← (fabs(slack/cost) ≤ reltol);
2765    }
```

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.

⟨ Log barrier methods 427, 428, 429, 430, 431 ⟩ Used in section 426.
⟨ Logarithm of a functional class methods 364, 365, 367, 369 ⟩ Used in section 362.
⟨ 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 methods 19, 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, 115 ⟩ 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 $Qy = b$ 231 ⟩ Used in section 230.
⟨ Solve $R^T y = b$ 234 ⟩ Used in section 233.
⟨ Solve $Rx = y$ 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.

⟨ **relentr** Hessian 341 ⟩ Used in section 340.

The MATH Library

(Version alpha 3-22-2002)