Chapter 4

Linear algebra

> In the training of programming for scientific computation the emphasis has historically been on squeezing out every drop of floating point performance for a given algorithm. ..... This practice, however, leads to highly tuned racecarlike software codes: delicate, easily broken and difficult to maintain, but capable of outperforming more user-friendly family cars. *Smith, Bjorstad and Gropp, An introduction to MPI [16]*

4.1 Introduction

In this chapter we deal with basic matrix operations, such as the solution of linear equations, calculate the inverse of a matrix, its determinant etc. The solution of linear equations is an important part of numerical mathematics and arises in many applications in the sciences. Here we focus in particular on so-called direct or elimination methods, which are in principle determined through a finite number of arithmetic operations. Iterative methods will be discussed in connection with eigenvalue problems in chapter 12.

This chapter serves also the purpose of introducing important programming details such as handling memory allocation for matrices, classes and the usage of the libraries which follow these lectures. The algorithms[1] we describe and their original source codes are taken from the widely used software package LAPACK [23], which follows two other popular packages developed in the 1970s, namely EISPACK and LINPACK. The latter was developed for linear equations and least square problems while the former was developed for solving symmetric, unsymmetric and generalized eigenvalue problems. From LAPACK's website `http://www.netlib.org` it is possible to download for free all source codes from this library. Both C++ and Fortran versions are available. Another important library is BLAS [24], which stands for Basic Linear Algebra Subprogram. It contains efficient codes for algebraic operations on vectors, matrices and vectors and matrices. Basically all modern supercomputer include this library, with efficient algorithms. Else, Matlab offers a very efficient programming environment for dealing with matrices. The classic text from where we have taken most of the formalism exposed here is the book on matrix computations by Golub and Van Loan [25]. Good recent introductory texts are Kincaid and Cheney [26] and Datta [27]. For more advanced ones see Trefethen and Bau III [28], Kress [29] and Demmel [30]. Ref. [25] contains an extensive list of textbooks on eigenvalue problems and linear algebra. LAPACK [23] contains also extensive listings to the research literature on matrix computations. You may also look up the lecture notes of INF-MAT3350 (Numerical Linear Algebra) at

[1] The various methods included in the library files are taken from LAPACK and Numerical Recipes [22] and have been rewritten in Fortran 90/95 and C++ by us.

http://www.uio.no/studier/emner/matnat/ifi/INF-MAT3350. For the introduction of the auxiliary library Blitz++ [31] we refer to the online manual at http://www.oonumerics.org.

4.2 Mathematical intermezzo

The matrices we will deal with are primarily square real symmetric or hermitian ones, assuming thereby that an $n \times n$ matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ for a real matrix[2] and $\mathbf{A} \in \mathbb{C}^{n\times n}$ for a complex matrix. For the sake of simplicity, we take a matrix $\mathbf{A} \in \mathbb{R}^{4\times 4}$ and a corresponding identity matrix $\mathbf{I}$

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \qquad \mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{4.1}$$

where $a_{ij} \in \mathbb{R}$. The inverse of a matrix, if it exists, is defined by

$$\mathbf{A}^{-1} \cdot \mathbf{A} = I.$$

In the following discussion, matrices are always two-dimensional arrays while vectors are one-dimensional arrays. In our nomenclature we will restrict boldfaced capitals letters such as $\mathbf{A}$ to represent a general matrix, which is a two-dimensional array, while a_{ij} refers to a matrix element with row number i and column number j. Similarly, a vector being a one-dimensional array, is labelled $\mathbf{x}$ and represented as (for a real vector)

$$\mathbf{x} \in \mathbb{R}^n \iff \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix},$$

with pertinent vector elements $x_i \in \mathbb{R}$. Note that this notation implies $x_i \in \mathbb{R}^{4\times 1}$ and that the members of $\mathbf{x}$ are column vectors. The elements of $x_i \in \mathbb{R}^{1\times 4}$ are row vectors.

Table 4.2 lists some essential features of various types of matrices one may encounter. Some of the

Table 4.1: Matrix properties

Relations	Name	matrix elements
$\mathbf{A} = \mathbf{A}^T$	symmetric	$a_{ij} = a_{ji}$
$\mathbf{A} = \left(\mathbf{A}^T\right)^{-1}$	real orthogonal	$\sum_k a_{ik}a_{jk} = \sum_k a_{ki}a_{kj} = \delta_{ij}$
$\mathbf{A} = \mathbf{A}^*$	real matrix	$a_{ij} = a^*_{ij}$
$\mathbf{A} = \mathbf{A}^\dagger$	hermitian	$a_{ij} = a^*_{ji}$
$\mathbf{A} = \left(\mathbf{A}^\dagger\right)^{-1}$	unitary	$\sum_k a_{ik}a^*_{jk} = \sum_k a^*_{ki}a_{kj} = \delta_{ij}$

matrices we will encounter are listed here

[2] A reminder on mathematical symbols may be appropriate here. The symbol $\mathbb{R}$ is the set of real numbers. Correspondingly, $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{C}$ represent the set of natural, integer and complex numbers, respectively. A symbol like $\mathbb{R}^n$ stands for an n-dimensional real Euclidean space, while $C[a,b]$ is the space of real or complex-valued continuous functions on the interval $[a,b]$, where the latter is a closed interval. Similalry, $C^m[a,b]$ is the space of m-times continuously differentiable functions on the interval $[a,b]$. For more symbols and notations, see the main text.

1. Diagonal if $a_{ij} = 0$ for $i \neq j$,

2. Upper triangular if $a_{ij} = 0$ for $i > j$, which for a 4×4 matrix is of the form

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{nn} \end{pmatrix}$$

3. Lower triangular if $a_{ij} = 0$ for $i < j$

$$\begin{pmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

4. Upper Hessenberg if $a_{ij} = 0$ for $i > j + 1$, which is similar to a upper triangular except that it has non-zero elements for the first subdiagonal row

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{pmatrix}$$

5. Lower Hessenberg if $a_{ij} = 0$ for $i < j + 1$

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

6. Tridiagonal if $a_{ij} = 0$ for $|i - j| > 1$

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{pmatrix}$$

There are many more examples, such as lower banded with bandwidth p for $a_{ij} = 0$ for $i > j + p$, upper banded with bandwidth p for $a_{ij} = 0$ for $i < j + p$, block upper triangular, block lower triangular etc.

For a real $n \times n$ matrix $\mathbf{A}$ the following properties are all equivalent

1. If the inverse of $\mathbf{A}$ exists, $\mathbf{A}$ is nonsingular.

2. The equation $\mathbf{Ax} = 0$ implies $\mathbf{x} = 0$.

3. The rows of $\mathbf{A}$ from a basis of $\mathbb{R}^n$.

4. The columns of $\mathbf{A}$ from a basis of $\mathbb{R}^n$.

5. $\mathbf{A}$ is a product of elementary matrices.
6. 0 is not an eigenvalue of $\mathbf{A}$.

The basic matrix operations that we will deal with are addition and subtraction

$$\mathbf{A} = \mathbf{B} \pm \mathbf{C} \Longrightarrow a_{ij} = b_{ij} \pm c_{ij}, \tag{4.2}$$

scalar-matrix multiplication

$$\mathbf{A} = \gamma\mathbf{B} \Longrightarrow a_{ij} = \gamma b_{ij}, \tag{4.3}$$

vector-matrix multiplication

$$\mathbf{y} = \mathbf{A}\mathbf{x} \Longrightarrow y_i = \sum_{j=1}^{n} a_{ij}x_j, \tag{4.4}$$

matrix-matrix multiplication

$$\mathbf{A} = \mathbf{B}\mathbf{C} \Longrightarrow a_{ij} = \sum_{k=1}^{n} b_{ik}c_{kj}, \tag{4.5}$$

transposition

$$\mathbf{A} = \mathbf{B}^T \Longrightarrow a_{ij} = b_{ji}, \tag{4.6}$$

and if $\mathbf{A} \in \mathbb{C}^{n\times n}$, conjugation results in

$$\mathbf{A} = \overline{\mathbf{B}}^T \Longrightarrow a_{ij} = \overline{b}_{ji}, \tag{4.7}$$

where a variable $\overline{z} = x - \imath y$ denotes the complex conjugate of $z = x + \imath y$. In a similar way we have the following basic vector operations, namely addition and subtraction

$$\mathbf{x} = \mathbf{y} \pm \mathbf{z} \Longrightarrow x_i = y_i \pm z_i, \tag{4.8}$$

scalar-vector multiplication

$$\mathbf{x} = \gamma\mathbf{y} \Longrightarrow x_i = \gamma y_i, \tag{4.9}$$

vector-vector multiplication (called Hadamard multiplication)

$$\mathbf{x} = \mathbf{y}\mathbf{z} \Longrightarrow x_i = y_i z_i, \tag{4.10}$$

the inner or so-called dot product

$$c = \mathbf{y}^T\mathbf{z} \Longrightarrow c = \sum_{j=1}^{n} y_j z_j, \tag{4.11}$$

with a c a constant and the outer product, which yields a matrix,

$$\mathbf{A} = \mathbf{y}\mathbf{z}^T \Longrightarrow a_{ij} = y_i z_j, \tag{4.12}$$

Other important operations are vector and matrix norms. A class of vector norms are the so-called p-norms

$$||\mathbf{x}||_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{\frac{1}{p}}, \tag{4.13}$$

where $p \geq 1$. The most important are the 1, 2 and ∞ norms given by

$$||\mathbf{x}||_1 = |x_1| + |x_2| + \cdots + |x_n|, \tag{4.14}$$

$$||\mathbf{x}||_2 = (|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2)^{\frac{1}{2}} = (\mathbf{x}^T\mathbf{x})^{\frac{1}{2}}, \tag{4.15}$$

and

$$||\mathbf{x}||_\infty = \max |x_i|, \tag{4.16}$$

for $1 \leq i \leq n$. From these definitions, one can derive several important relations, of which the so-called Cauchy-Schwartz inequality is of great importance for many algorithms. It reads for any $\mathbf{x}$ and $\mathbf{y}$ in a real or complex inner product space satisfy

$$|\mathbf{x}^T\mathbf{y}| \leq ||\mathbf{x}||_2||\mathbf{y}||_2, \tag{4.17}$$

and the equality is obeyed only if $\mathbf{x}$ and $\mathbf{y}$ are linearly dependent. An important relation which follows from the Cauchy-Schwartz relation is the famous triangle relation, which states that for any $\mathbf{x}$ and $\mathbf{y}$ in a real or complex inner product space satisfy

$$||\mathbf{x} + \mathbf{y}||_2 \leq ||\mathbf{x}||_2 + ||\mathbf{y}||_2. \tag{4.18}$$

Proofs can be found in for example Ref. [25]. As discussed in chapter 2, the analysis of the relative error is important in our studies of loss of numerical precision. Using a vector norm we can define the relative error for the machine representation of a vector $\mathbf{x}$. We assume that $fl(\mathbf{x}) \in \mathbb{R}^n$ is the machine representation of a vector $\mathbf{x} \in \mathbb{R}^n$. If $\mathbf{x} \neq 0$, we define the relative error as

$$\epsilon = \frac{||fl(\mathbf{x}) - \mathbf{x}||}{||\mathbf{x}||}. \tag{4.19}$$

Using the ∞-norm one can define a relative error that can be translated into a statement on the correct significant digits of $fl(\mathbf{x})$,

$$\frac{||fl(\mathbf{x}) - \mathbf{x}||_\infty}{||\mathbf{x}||_\infty} \approx 10^{-l}, \tag{4.20}$$

where the largest component of $fl(\mathbf{x})$ has roughly l correct significant digits.

We can define similar matrix norms as well. The most frequently used are the Frobenius norm

$$||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}, \tag{4.21}$$

and the p-norms

$$||\mathbf{A}||_p = \frac{||\mathbf{A}\mathbf{x}||_p}{||\mathbf{x}||_p}, \tag{4.22}$$

assuming that $\mathbf{x} \neq 0$. We refer the reader to the text of Golub and Van Loan [25] for a further discussion of these norms.

The way we implement these operations will be discussed below, as it depends on the programming language we opt for.

4.3 Programming details

Many programming problems arise from improper treatment of arrays. In this section we will discuss some important points such as array declaration, memory allocation and array transfer between functions. We distinguish between two cases: (a) array declarations where the array size is given at compilation time, and (b) where the array size is determined during the execution of the program, so-called dymanic memory allocation. Useful references on C++ programming details, in particular on the use of pointers and memory allocation, are Reek's text [32] on pointers in C, Berryhill's monograph [33] on scientific programming in C++ and finally Franek's text [34] on memory as a programming concept in C and C++. Good allround texts on C++ programming in engineering and science are the books by Flowers [19] and Barton and Nackman [20]. See also the online lecture notes on C++ at `http://heim.ifi.uio.no/~hpl/INF-VERK4830`. For Fortran 90/95 we recommend the online lectures at `http://folk.uio.no/gunnarw/INF-VERK4820`. These web pages contain extensive references to other C++ and Fortran 90/95 resources. Both web pages contain enough material, lecture notes and exercises, in order to serve as material for own studies.

Figure 4.1: Segmentation fault, again and again! Alas, this is a situation you must likely will end up in, unless you initialize, access, allocate or deallocate properly your arrays. Many program development environments such as Dev C++ at `www.bloodshed.net` provide debugging possibilities. Another possibility, discussed in appendix A is to use the debugger GDB within the text editor emacs. Beware however that there may be segmentation errors which occur due to errors in libraries of the operating system. This author spent two weeks on tracing a segmentation error from a program which run perfectly prior to an upgrade of the operating system. There was a bug in the library glibc of the new linux distribution. (Drawing: courtesy by Victoria Popsueva 2003.)

4.3.1 Declaration of fixed-sized vectors and matrices

Table 4.2 presents a small program which treats essential features of vector and matrix handling where the dimensions are declared in the program code.

In **line a** we have a standard C++ declaration of a vector. The compiler reserves memory to store five integers. The elements are vec[0], vec[1],....,vec[4]. Note that the numbering of elements starts with zero. Declarations of other data types are similar, including structure data.

The symbol vec is an element in memory containing the address to the first element vec[0] and is a pointer to a vector of five integer elements.

In **line b** we have a standard fixed-size C++ declaration of a matrix. Again the elements start with zero, matr[0][0], matr[0][1], ..., matr[0][4], matr[1][0],.... . This sequence of elements also shows how data are stored in memory. For example, the element matr[1][0] follows matr[0][4]. This is important in order to produce an efficient code and avoid memory stride.

There is one further important point concerning matrix declaration. In a similar way as for the symbol **vec**, **matr** is an element in memory which contains an address to a vector of three elements, but now these elements are not integers. Each element is a vector of five integers. This is the correct way to understand the declaration in **line b**. With respect to pointers this means that matr is *pointer-to-a-pointer-to-an-integer* which we can write $**$matr. Furthermore $*$matr is *a-pointer-to-a-pointer* of five integers. This interpretation is important when we want to transfer vectors and matrices to a function.

In **line c** we transfer vec[] and matr[][] to the function sub_1(). To be specific, we transfer the addresses of vec[] and matr[][] to sub_1().

In **line d** we have the function definition of sub_1(). The **int** vec[] is a pointer to an integer. Alternatively we could write **int** $*$vec. The first version is better. It shows that it is a vector of several integers, but not how many. The second version could equally well be used to transfer the address to a single integer element. Such a declaration does not distinguish between the two cases.

The next definition is **int** matr[][5]. This is a pointer to a vector of five elements and the compiler must be told that each vector element contains five integers. Here an alternative version could be int ($*$matr)[5] which clearly specifies that matr is a pointer to a vector of five integers.

There is at least one drawback with such a matrix declaration. If we want to change the dimension of the matrix and replace 5 by something else we have to do the same change in all functions where this matrix occurs.

There is another point to note regarding the declaration of variables in a function which includes vectors and matrices. When the execution of a function terminates, the memory required for the variables is released. In the present case memory for all variables in main() are reserved during the whole program execution, but variables which are declared in sub_1() are released when the execution returns to main().

4.3.2 Runtime declarations of vectors and matrices in C++

As mentioned in the previous subsection a fixed size declaration of vectors and matrices before compilation is in many cases bad. You may not know beforehand the actually needed sizes of vectors and matrices. In large projects where memory is a limited factor it could be important to reduce memory requirement for matrices which are not used any more. In C an C++ it is possible and common to postpone size declarations of arrays untill you really know what you need and also release memory reservations when it is not needed any more. The details are shown in Table 4.3.

In **line a** we declare a pointer to an integer which later will be used to store an address to the first element of a vector. Similarily, **line b** declares a pointer-to-a-pointer which will contain the address to a pointer of row vectors, each with col integers. This will then become a matrix[col][col]

Table 4.2: Matrix handling program where arrays are defined at compilation time

```
int main()
{
    int k,m, row = 3, col = 5;
    int vec[5];                                                    // line a
    int matr[3][5];                                                // line b

    for(k = 0; k < col; k++) vec[k] = k;                    // data into vector[]
    for(m = 0; m < row; m++) {                              // data into matr[][]
        for(k = 0; k < col ; k++) matr[m][k] = m + 10 * k;
    }
    printf("\n\nVector data in main():\n");                 // print vector data
    for(k = 0; k < col; k++) printf("vector[%d] = %d ",k, vec[k]);
     printf("\n\nMatrix data in main():");
    for(m = 0; m < row; m++) {
        printf("\n");
        for(k = 0; k < col; k++)
            printf("matr[%d][[%d] = %d ",m,k,matr[m][k]);
        }
    }
    printf("\n");
    sub_1(row, col, vec, matr);                                    // line c
    return 0;
}  // End: function main()

void sub_1(int row, int col, int vec[], int matr[][5])             // line d
{
    int k,m;

    printf("\n\nVector data in sub_1():\n");                  // print vector data
    for(k = 0; k < col; k++) printf("vector[%d] = %d ",k, vec[k]);
     printf("\n\nMatrix data in sub_1():");
    for(m = 0; m < row; m++) {
        printf("\n");
        for(k = 0; k < col; k++) {
            printf("matr[%d][[%d] = %d ",m, k, matr[m][k]);
        }
    }
    printf("\n");
}  // End: function sub_1()
```

Table 4.3: Matrix handling program with dynamic array allocation.

```
int main()
{
    int *vec;                                                    // line a
    int **matr;                                                  // line b
    int m, k, row, col, total = 0;

    printf("\n\nRead in number of rows = ");                     // line c
    scanf("%d",&row);
    printf("\n\nRead in number of column = ");
    scanf("%d", &col);

    vec = new int [col];                                         // line d
    matr = (int **)matrix(row, col, sizeof(int));                // line e
    for(k = 0; k < col; k++) vec[k] = k;                  // store data in vector[]
    for(m = 0; m < row; m++) {                           // store data in array[][]
        for(k = 0; k < col; k++) matr[m][k] = m + 10 * k;
    }
    printf("\n\nVector data in main():\n");              // print vector data
    for(k = 0; k < col; k++) printf("vector[%d] = %d ",k,vec[k]);
    printf("\n\nArray data in main():");
    for(m = 0; m < row; m++) {
        printf("\n");
        for(k = 0; k < col; k++) {
            printf("matrix[%d][[%d] = %d ",m, k, matr[m][k]);
        }
    }
    printf("\n");
    for(m = 0; m < row; m++) {                                   // access the array
        for(k = 0; k < col; k++) total += matr[m][k];
    }
    printf("\n\nTotal = %d\n",total);
    sub_1(row, col, vec, matr);
    free_matrix((void **)matr);                                  // line f
    delete [] vec;                                               // line g
    return 0;
}  // End: function main()

void sub_1(int row, int col, int vec[], int **matr)              // line h
{
    int k,m;

    printf("\n\nVector data in sub_1():\n");              // print vector data
    for(k = 0; k < col; k++) printf("vector[%d] = %d ",k, vec[k]);
    printf("\n\nMatrix data in sub_1():");
    for(m = 0; m < row; m++) {
        printf("\n");
        for(k = 0; k < col; k++) {
            printf("matrix[%d][[%d] = %d ",m,k,matr[m][k]);
        }
    }
    printf("\n");
}  // End: function sub_1()
```

In **line c** we read in the size of vec[] and matr[][] through the numbers row and col.

Next we reserve memory for the vector in **line d**. In **line e** we use a user-defined function to reserve necessary memory for matrix[row][col] and again matr contains the address to the reserved memory location.

The remaining part of the function main() are as in the previous case down to **line f**. Here we have a call to a user-defined function which releases the reserved memory of the matrix. In this case this is not done automatically.

In **line g** the same procedure is performed for vec[]. In this case the standard C++ library has the necessary function.

Next, in **line h** an important difference from the previous case occurs. First, the vector declaration is the same, but the matr declaration is quite different. The corresponding parameter in the call to sub_1[] in **line g** is a double pointer. Consequently, matr in **line h** must be a double pointer.

Except for this difference sub_1() is the same as before. The new feature in Table 4.3 is the call to the user-defined functions **matrix** and **free_matrix**. These functions are defined in the library file **lib.cpp**. The code for the dynamic memory allocation is given below.

http://folk.uio.no/mhjensen/fys3150/2005/programs/cplusplusLibrary/lib.cpp

```
/*
 * The function
 *      void  **matrix()
 * reserves dynamic memory for a two-dimensional matrix
 * using the C++ command new . No initialization of the elements.
 * Input data:
 *  int row      - number of  rows
 *  int col      - number of columns
 *  int num_bytes- number of bytes for each
 *                 element
 * Returns a void  **pointer to the reserved memory location.
 */

void **matrix(int row, int col, int num_bytes)
{
int       i, num;
char      **pointer, *ptr;

pointer = new(nothrow) char* [row];
if(!pointer) {
  cout << "Exception handling: Memory allocation failed";
  cout << " for "<< row << "row addresses !" << endl;
  return NULL;
}
i = (row * col * num_bytes)/sizeof(char);
pointer[0] = new(nothrow) char [i];
if(!pointer[0]) {
  cout << "Exception handling: Memory allocation failed";
  cout << " for address to " << i << " characters !" << endl;
  return NULL;
}
ptr = pointer[0];
num = col * num_bytes;
for(i = 0; i < row; i++, ptr += num )    {
```

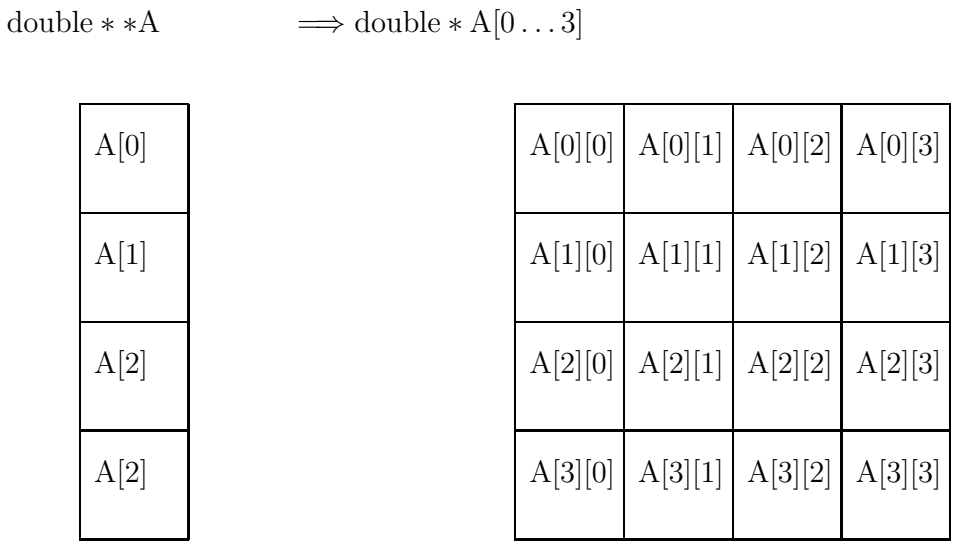

Figure 4.2: Conceptual representation of the allocation of a matrix in C++.

```
    pointer[i] = ptr;
  }
  return  (void **)pointer;
  } // end: function void **matrix()
```

As an alternative, you could write your own allocation and deallocation of matrices. This can be done rather straightforwardly with the following statements. Recall first that a matrix is represented by a double pointer that points to a contiguous memory segment holding a sequence of double* pointers in case our matrix is a double precision variable. Then each double* pointer points to a row in the matrix. A declaration like **double**** A; means that A[i] is a pointer to the $i+1$-th row A[i] and A[i][j] is matrix entry (i,j). The way we would allocate memory for such a matrix of dimensionality $n \times n$ is for example using the following piece of code

```
int n;
double **  A;

A = new double*[n]
for ( i = 0; i < n; i++)
    A[i] = new double[N];
```

When we declare a matrix (a two-dimensional array) we must first declare an array of double variables. To each of this variables we assign an allocation of a single-dimensional array. A conceptual picture on how a matrix **A** is stored in memory is shown in Fig. 4.2.

Allocated memory should always be deleted when it is no longer needed. We free memory using the

statements

```
for ( i = 0; i < n; i++)
    delete[] A[i];
delete[] A;
```

delete [] A;, which frees an array of pointers to matrix rows.

However, including a library like Blitz++ `http://www.oonumerics.org` makes life much easier when dealing with matrices. This is discussed below.

4.3.3 Matrix operations and C++ and Fortran 90/95 features of matrix handling

Many program libraries for scientific computing are written in Fortran, often also in older version such Fortran 77. When using functions from such program libraries, there are some differences between C++ and Fortran 90/95 encoding of matrices and vectors worth noticing. Here are some simple guidelines in order to avoid some of the most common pitfalls.

First of all, when we think of an $n \times n$ matrix in Fortran and C++, we typically would have a mental picture of a two-dimensional block of stored numbers. The computer stores them however as sequential strings of numbers. The latter could be stored as row-major order or column-major order. What do we mean by that? Recalling that for our matrix elements a_{ij}, i refers to rows and j to columns, we could store a matrix in the sequence $a_{11}a_{12}\dots a_{1n}a_{21}a_{22}\dots a_{2n}\dots a_{nn}$ if it is row-major order (we go along a given row i and pick up all column elements j) or it could be stored in column-major order $a_{11}a_{21}\dots a_{n1}a_{12}a_{22}\dots a_{n2}\dots a_{nn}$.

Fortran stores matrices in the latter way, i.e., by column-major, while C++ stores them by row-major. It is crucial to keep this in mind when we are dealing with matrices, because if we were to organize the matrix elements in the wrong way, important properties like the transpose of a real matrix or the inverse can be wrong, and obviously yield wrong physics. Fortran subscripts begin typically with 1, although it is no problem in starting with zero, while C++ starts with 0 for the first element. This means that $A(1,1)$ in Fortran is equivalent to $A[0][0]$ in C++. Moreover, since the sequential storage in memory means that nearby matrix elements are close to each other in the memory locations (and thereby easier to fetch) , operations involving e.g., additions of matrices may take more time if we do not respect the given ordering.

To see this, consider the following coding of matrix addition in C++ and Fortran 90/95. We have $n \times n$ matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ and we wish to evaluate $\mathbf{A} = \mathbf{B} + \mathbf{C}$ according to Eq. (4.2). In C++ this would be coded like

```
for(i=0 ; i < n ; i++) {
    for(j=0 ; j < n ; j++) {
        a[i][j]=b[i][j]+c[i][j]
    }
}
```

while in Fortran 90/95 we would have

```
DO   j=1,   n
   DO i=1, n
      a(i,j)=b(i,j)+c(i,j)
   ENDDO
ENDDO
```

Fig. 4.3 shows how a 3×3 matrix $\mathbf{A}$ is stored in both row-major and column-major ways.

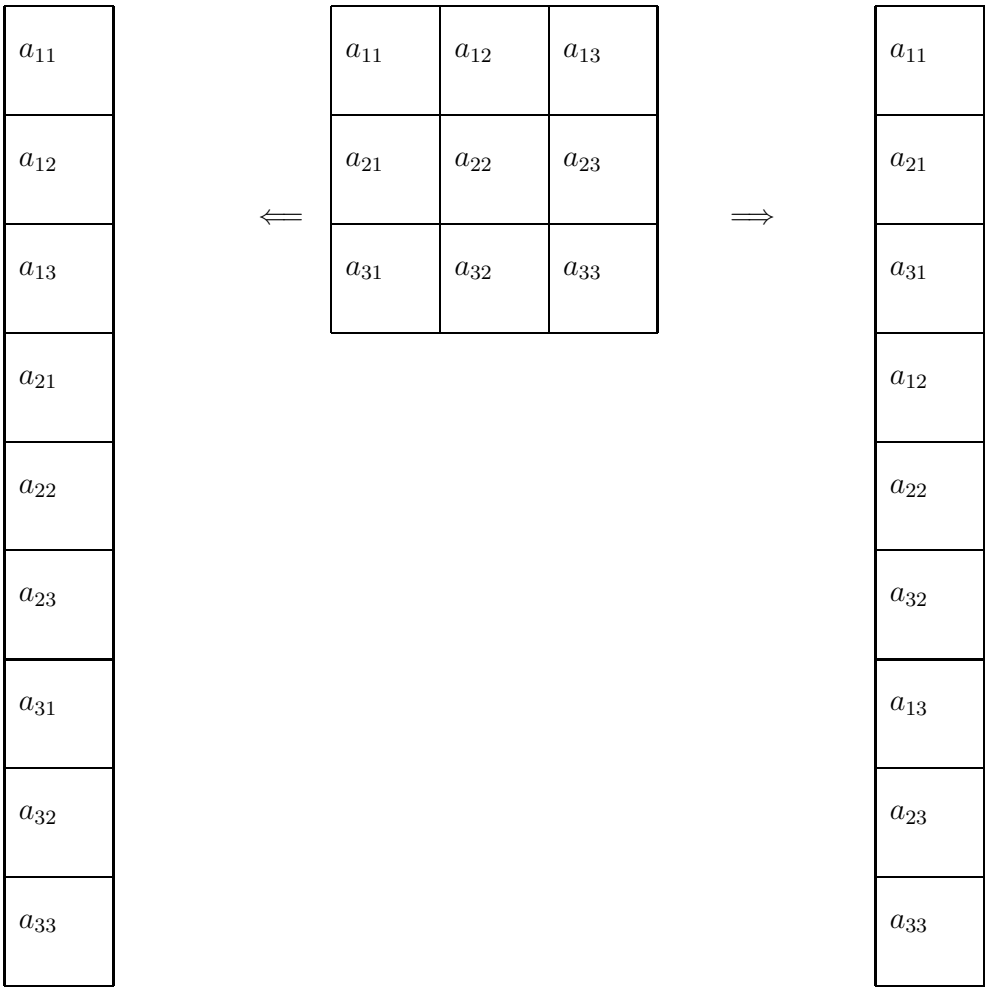

Figure 4.3: Row-major storage of a matrix to the left (C++ way) and column-major to the right (Fortran way).

Interchanging the order of i and j can lead to a considerable enhancement in process time. In Fortran 90/95 we would write the above statements in a much simpler way a=b+c. However, the addition still involves $\sim n^2$ operations. Matrix multiplication or taking the inverse requires $\sim n^3$ operations. The matrix multiplication of Eq. (4.5) of two matrices $\mathbf{A} = \mathbf{BC}$ could then take the following form in C++

```
for(i=0 ; i < n ; i++) {
    for(j=0 ; j < n ; j++) {
        for(k=0 ; k < n ; k++) {
            a[i][j]+=b[i][k]*c[k][j]
        }
    }
}
```

and in Fortran 90/95 we have

```
DO   j=1,   n
   DO i=1, n
      DO k = 1, n
         a(i,j)=a(i,j)+b(i,k)*c(i,j)
      ENDDO
   ENDDO
ENDDO
```

However, Fortran 90/95 has an intrisic function called MATMUL, and the above three loops can be coded in a single statement a=MATMUL(b,c). Fortran 90/95 contains several array manipulation statements, such as dot product of vectors, the transpose of a matrix etc etc. The outer product of two vectors is however not included in Fortran 90/95. The coding of Eq. (4.12) takes then the following form in C++

```
for(i=0 ; i < n ; i++) {
    for(j=0 ; j < n ; j++) {
        a[i][j]+=x[i]*y[j]
    }
}
```

and in Fortran 90/95 we have

```
DO   j=1,   n
   DO i=1, n
        a(i,j)=a(i,j)+x(j)*y(i)
   ENDDO
ENDDO
```

A matrix-matrix multiplication of a general $n \times n$ matrix with

$$a(i,j) = a(i,j) + b(i,k) * c(i,j),$$

in its inner loops requires a multiplication and an addition. We define now a flop (floating point operation) as one of the following floating point arithmetic operations, viz addition, subtraction, multiplication and division. The above two floating point operations (flops) are done n^3 times meaning that a general matrix multiplication requires $2n^3$ flops if we have a square matrix. If we assume that our computer performs 10^9 flops per second, then to perform a matrix multiplication of a 1000×1000 case should take two

seconds. This can be reduced if we multiply two matrices which are upper triangular such as

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{44} \end{pmatrix}.$$

The multiplication of two upper triangular matrices $\mathbf{BC}$ yields another upper triangular matrix $\mathbf{A}$, resulting in the following C++ code

```
for(i=0 ; i < n ; i++) {
   for(j=i ; j < n ; j++) {
      for(k=i ; k < j ; k++) {
         a[i][j]+=b[i][k]*c[k][j]
      }
   }
}
```

The fact that we have the constraint $i \leq j$ leads to the requirement for the computation of a_{ij} of $2(j-i+1)$ flops. The total number of flops is then

$$\sum_{i=1}^{n}\sum_{j=1}^{n} 2(j-i+1) = \sum_{i=1}^{n}\sum_{j=1}^{n-i+1} 2j \approx \sum_{i=1}^{n} \frac{2(n-i+1)^2}{2},$$

where we used that $\sum_{j=1}^{n} j = n(n+1)/2 \approx n^2/2$ for large n values. Using in addition that $\sum_{j=1}^{n} j^2 \approx n^3/3$ for large n values, we end up with approximately $n^3/3$ flops for the multiplication of two upper triangular matrices. This means that if we deal with matrix multiplication of upper triangular matrices, we reduce the number of flops by a factor six if we code our matrix multiplication in an efficient way.

It is also important to keep in mind that computers are finite, we can thus not store infinitely large matrices. To calculate the space needed in memory for an $n \times n$ matrix with double precision, 64 bits or 8 bytes for every matrix element, one needs simply compute $n \times n \times 8$ bytes . Thus, if $n = 10000$, we will need close to 1GB of storage. Decreasing the precision to single precision, only halves our needs.

A further point we would like to stress, is that one should in general avoid fixed (at compilation time) dimensions of matrices. That is, one could always specify that a given matrix $\mathbf{A}$ should have size $A[100][100]$, while in the actual execution one may use only $A[10][10]$. If one has several such matrices, one may run out of memory, while the actual processing of the program does not imply that. Thus, we will always recommend that you use dynamic memory allocation, and deallocation of arrays when they are no longer needed. In Fortran 90/95 one uses the intrisic functions **ALLOCATE** and **DEALLOCATE**, while C++ employs the functions **new** and **delete**.

Fortran 90/95 allocate statement and mathematical operations on arrays

An array is declared in the declaration section of a program, module, or procedure using the dimension attribute. Examples include

```
REAL, DIMENSION (10) :: x,y
REAL, DIMENSION (1:10) :: x,y
INTEGER, DIMENSION (-10:10) :: prob
INTEGER, DIMENSION (10,10) :: spin
```

The default value of the lower bound of an array is 1. For this reason the first two statements are equivalent to the first. The lower bound of an array can be negative. The last two statements are examples of two-dimensional arrays.

Rather than assigning each array element explicitly, we can use an array constructor to give an array a set of values. An array constructor is a one-dimensional list of values, separated by commas, and delimited by "(/" and "/)". An example is

```
a(1:3) = (/ 2.0, -3.0, -4.0 /)
```

is equivalent to the separate assignments

```
a(1) = 2.0
a(2) = -3.0
a(3) = -4.0
```

One of the better features of Fortran 90/95 is dynamic storage allocation. That is, the size of an array can be changed during the execution of the program. To see how the dynamic allocation works in Fortran 90/95, consider the following simple example where we set up a 4×4 unity matrix.

```
      ......
      IMPLICIT NONE
!     The definition of the matrix, using dynamic allocation
      REAL, ALLOCATABLE, DIMENSION(:,:) :: unity
!     The size of the matrix
      INTEGER :: n
!     Here we set the dim n=4
      n=4
!  Allocate now place in memory for the matrix
      ALLOCATE ( unity(n,n) )
!  all elements are set equal zero
      unity=0.
!     setup identity matrix
      DO i=1,n
         unity(i,i)=1.
      ENDDO
      DEALLOCATE ( unity)
      .......
```

We always recommend to use the deallocation statement, since this frees space in memory. If the matrix is transferred to a function from a calling program, one can transfer the dimensionality n of that matrix with the call. Another possibility is to determine the dimensionality with the `SIZE` function. Writing a statement like n=**SIZE**(unity,DIM=1) gives the number of rows, while using DIM=2 gives the number of columns. Note however that this involves an extra call to a function. If speed matters, one should avoid such calls.

4.4 Linear Systems

In this section we outline some of the most used algorithms to solve sets of linear equations. These algorithms are based on Gaussian elimination [25, 29] and will allow us to catch several birds with a

stone. We will show how to rewrite a matrix $\mathbf{A}$ in terms of an upper and a lower triangular matrix, from which we easily can solve linear equation, compute the inverse of $\mathbf{A}$ and obtain the determinant. We start with Gaussian elimination, move to the more efficient LU-algorithm, which forms the basis for many linear algebra applications, and end the discussion with special cases such as the Cholesky decomposition and linear system of equations with a tridiagonal matrix.

We begin however with an example which demonstrates the importance of being able to solve linear equations. Suppose we want to solve the following boundary value equation

$$-\frac{d^2u(x)}{dx^2} = f(x, u(x)),$$

with $x \in (a, b)$ and with boundary conditions $u(a) = u(b) = 0$. We assume that f is a continuous function in the domain $x \in (a, b)$. Since, except the few cases where it is possible to find analytic solutions, we will seek after approximate solutions, we choose to represent the approximation to the second derivative from the previous chapter

$$f'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} + O(h^2).$$

We subdivide our interval $x \in (a, b)$ into n subintervals by setting $x_i = ih$, with $i = 0, 1, \ldots, n+1$. The step size is then given by $h = (b-a)/(n+1)$ with $n \in \mathbb{N}$. For the internal grid points $i = 1, 2, \ldots n$ we replace the differential operator with the above formula resulting in

$$u''(x_i) \approx \frac{u(x_i + h) - 2u(x_i) + u(x_i - h)}{h^2},$$

which we rewrite as

$$u_i'' \approx \frac{u_{i+1} - 2u_i + u_{i-i}}{h^2}.$$

We can rewrite our original differential equation in terms of a discretized equation with approximations to the derivatives as

$$-\frac{u_{i+1} - 2u_i + u_{i-i}}{h^2} = f(x_i, u(x_i)),$$

with $i = 1, 2, \ldots, n$. We need to add to this system the two boundary conditions $u(a) = u_0$ and $u(b) = u_{n+1}$. If we define a matrix

$$\mathbf{A} = \frac{1}{h^2}\begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & \cdots & \cdots & \cdots & \cdots & \cdots \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{pmatrix}$$

and the corresponding vectors $\mathbf{u} = (u_1, u_2, \ldots, u_n)^T$ and $\mathbf{f}(\mathbf{u}) = f(x_1, x_2, \ldots, x_n, u_1, u_2, \ldots, u_n)^T$ we can rewrite the differential equation including the boundary conditions as a system of linear equations with a large number of unknowns

$$\mathbf{A}\mathbf{u} = \mathbf{f}(\mathbf{u}). \tag{4.23}$$

We assume that the solution u exists and is unique for the exact differential equation, viz that the boundary value problem has a solution. But the discretization of the above differential equation leads to several

questions, such as how well does the approximate solution resemble the exact one as $h \rightarrow 0$, or does a given small value of h allow us to establish existence and uniqueness of the solution.

Here we specialize to two particular cases. Assume first that the function f does not depend on $u(x)$. Then our linear equation reduces to

$$\mathbf{Au} = \mathbf{f}, \tag{4.24}$$

which is nothing but a simple linear equation with a tridiagonal matrix $\mathbf{A}$. We will solve such a system of equations in subsection 4.4.3.

If we assume that our boundary value problem is that of a quantum mechanical particle confined by a harmonic oscillator potential, then our function f takes the form (assuming that all constants $m = \hbar = \omega = 1$) $f(x_i, u(x_i)) = -x_i^2 u(x_i) + 2\lambda u(x_i)$ with λ being the eigenvalue. Inserting this into our equation, we define first a new matrix $\mathbf{A}$ as

$$\mathbf{A} = \begin{pmatrix} \frac{2}{h^2} + x_1^2 & -\frac{1}{h^2} & & & & & \\ -\frac{1}{h^2} & \frac{2}{h^2} + x_2^2 & -\frac{1}{h^2} & & & & \\ & -\frac{1}{h^2} & \frac{2}{h^2} + x_3^2 & -\frac{1}{h^2} & & & \\ & \cdots & \cdots & \cdots & \cdots & \cdots & \\ & & & -\frac{1}{h^2} & \frac{2}{h^2} + x_{n-1}^2 & -\frac{1}{h^2} & \\ & & & & -\frac{1}{h^2} & \frac{2}{h^2} + x_n^2 & \end{pmatrix}, \tag{4.25}$$

which leads to the following eigenvalue problem

$$\begin{pmatrix} \frac{2}{h^2} + x_1^2 & -\frac{1}{h^2} & & & & & \\ -\frac{1}{h^2} & \frac{2}{h^2} + x_2^2 & -\frac{1}{h^2} & & & & \\ & -\frac{1}{h^2} & \frac{2}{h^2} + x_3^2 & -\frac{1}{h^2} & & & \\ & \cdots & \cdots & \cdots & \cdots & \cdots & \\ & & & -\frac{1}{h^2} & \frac{2}{h^2} + x_{n-1}^2 & -\frac{1}{h^2} & \\ & & & & -\frac{1}{h^2} & \frac{2}{h^2} + x_n^2 & \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \\ \\ \\ u_n \end{pmatrix} = 2\lambda \begin{pmatrix} u_1 \\ u_2 \\ \\ \\ \\ u_n \end{pmatrix}.$$

We will solve this type of equations in chapter 12. These lecture notes contain however several other examples of rewriting mathematical expressions into matrix problems. In chapter 7 we show how a set of linear integral equation when discretized can be transformed into a simple matrix inversion problem. The specific example we study in that chapter is the rewriting of Schrödinger's equation for scattering problems. Other examples of linear equations will appear in our discussion of ordinary and partial differential equations.

4.4.1 Gaussian elimination

Any discussion on the solution of linear equations should start with Gaussian elimination. This text is no exception. We start with the linear set of equations

$$\mathbf{Ax} = \mathbf{w}.$$

We assume also that the matrix $\mathbf{A}$ is non-singular and that the matrix elements along the diagonal satisfy $a_{ii} \neq 0$. We discuss later how to handle such cases. In the discussion we limit ourselves again to a matrix $\mathbf{A} \in \mathbb{R}^{4\times 4}$, resulting in a set of linear equations of the form

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix}.$$

or

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 = &\quad w_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 = &\quad w_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 = &\quad w_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 = &\quad w_4. \end{aligned}$$

The basic idea of Gaussian elimination is to use the first equation to eliminate the first unknown x_1 from the remaining $n-1$ equations. Then we use the new second equation to eliminate the second unknown x_2 from the remaining $n-2$ equations. With $n-1$ such eliminations we obtain a so-called upper triangular set of equations of the form

$$\begin{aligned} b_{11}x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 = &\quad y_1 \\ b_{22}x_2 + b_{23}x_3 + b_{24}x_4 = &\quad y_2 \\ b_{33}x_3 + b_{34}x_4 = &\quad y_3 \\ b_{44}x_4 = &\quad y_4. \end{aligned}$$

We can solve this system of equations recursively starting from x_n (in our case x_4) and proceed with what is called a backward substitution. This process can be expressed mathematically as

$$x_m = \frac{1}{b_{mm}}\left(y_m - \sum_{k=m+1}^{n} b_{mk}x_k\right) \qquad m = n-1, n-2, \ldots, 1.$$

To arrive at such an upper triangular system of equations, we start by eliminating the unknown x_1 for $j = 2, n$. We achieve this by multiplying the first equation by a_{j1}/a_{11} and then subtract the result from the jth equation. We assume obviously that $a_{11} \neq 0$ and that $\mathbf{A}$ is not singular. We will come back to this problem below.

Our actual 4×4 example reads after the first operation

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & (a_{22} - \frac{a_{21}a_{12}}{a_{11}}) & (a_{23} - \frac{a_{21}a_{13}}{a_{11}}) & (a_{24} - \frac{a_{21}a_{14}}{a_{11}}) \\ 0 & (a_{32} - \frac{a_{31}a_{12}}{a_{11}}) & (a_{33} - \frac{a_{31}a_{13}}{a_{11}}) & (a_{34} - \frac{a_{31}a_{14}}{a_{11}}) \\ 0 & (a_{42} - \frac{a_{41}a_{12}}{a_{11}}) & (a_{43} - \frac{a_{41}a_{13}}{a_{11}}) & (a_{44} - \frac{a_{41}a_{14}}{a_{11}}) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ w_2^{(2)} \\ w_3^{(2)} \\ w_4^{(2)} \end{pmatrix}.$$

or

$$\begin{aligned} b_{11}x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 = &\quad y_1 \\ a_{22}^{(2)}x_2 + a_{23}^{(2)}x_3 + a_{24}^{(2)}x_4 = &\quad w_2^{(2)} \\ a_{32}^{(2)}x_2 + a_{33}^{(2)}x_3 + a_{34}^{(2)}x_4 = &\quad w_3^{(2)} \\ a_{42}^{(2)}x_2 + a_{43}^{(2)}x_3 + a_{44}^{(2)}x_4 = &\quad w_4^{(2)}, \end{aligned} \tag{4.26}$$

with the new coefficients

$$b_{1k} = a_{1k}^{(1)} \; k = 1, \ldots, n,$$

where each $a_{1k}^{(1)}$ is equal to the original a_{1k} element. The other coefficients are

$$a_{jk}^{(2)} = a_{jk}^{(1)} - \frac{a_{j1}^{(1)}a_{1k}^{(1)}}{a_{11}^{(1)}} \; j, k = 2, \ldots, n,$$

with a new right-hand side given by

$$y_1 = w_1^{(1)},\ w_j^{(2)} = w_j^{(1)} - \frac{a_{j1}^{(1)} w_1^{(1)}}{a_{11}^{(1)}}\ j = 2, \ldots, n.$$

We have also set $w_1^{(1)} = w_1$, the original vector element. We see that the system of unknowns $x_1, \ldots, x_n$ is transformed into an $(n-1) \times (n-1)$ problem.

This step is called forward substitution. Proceeding with these substitutions, we obtain the general expressions for the new coefficients

$$a_{jk}^{(m+1)} = a_{jk}^{(m)} - \frac{a_{jm}^{(m)} a_{mk}^{(m)}}{a_{mm}^{(m)}}\ j, k = m+1, \ldots, n,$$

with $m = 1, \ldots, n-1$ and a right-hand side given by

$$w_j^{(m+1)} = w_j^{(m)} - \frac{a_{jm}^{(m)} w_m^{(m)}}{a_{mm}^{(m)}}\ j = m+1, \ldots, n.$$

This set of $n-1$ elimations leads us to Eq. (4.26), which is solved by back substitution. If the arithmetics is exact and the matrix $\mathbf{A}$ is not singular, then the computed answer will be exact. However, as discussed in the two preceeding chapters, computer arithmetics is not necessarily exact. We will always have to cope with truncations and possible losses of precision. Even though the matrix elements along the diagonal are not zero, numerically small numbers may appear and subsequent divisions may lead to large numbers, which, if added to a small number may yield losses of precision. Suppose for example that our first division in $(a_{22} - a_{21}a_{12}/a_{11})$ results in -10^{-7} and that a_{22} is one. one. We are then adding $10^7 + 1$. With single precision this results in 10^7. Already at this stage we see the potential for producing wrong results.

The solution to this set of problems is called pivoting, and we distinguish between partial and full pivoting. Pivoting means that if small values (especially zeros) do appear on the diagonal we remove them by rearranging the matrix and vectors by permuting rows and columns. As a simple example, let us assume that at some stage during a calculation we have the following set of linear equations

$$\begin{pmatrix} 1 & 3 & 4 & 6 \\ 0 & 10^{-8} & 198 & 19 \\ 0 & -91 & 51 & 9 \\ 0 & 7 & 76 & 541 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}.$$

The element at row $i = 2$ and column 2 is 10^{-8} and may cause problems for us in the next forward substitution. The element $i = 2, j = 3$ is the largest in the second row and the element $i = 3, j = 2$ is the largest in the third row. The small element can be removed by rearranging the rows and/or columns to bring a larger value into the $i = 2, j = 2$ element.

In partial or column pivoting, we rearrange the rows of the matrix and the right-hand side to bring the numerically largest value in the column onto the diagonal. For our example matrix the largest value of column two is in element $i = 3, j = 2$ and we interchange rows 2 and 3 to give

$$\begin{pmatrix} 1 & 3 & 4 & 6 \\ 0 & -91 & 51 & 9 \\ 0 & 10^{-8} & 198 & 19 \\ 0 & 7 & 76 & 541 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_3 \\ y_2 \\ y_4 \end{pmatrix}.$$

Note that our unknown variables x_i remain in the same order which simplifies the implementation of this procedure. The right-hand side vector, however, has been rearranged. Partial pivoting may be implemented for every step of the solution process, or only when the diagonal values are sufficiently small as to potentially cause a problem. Pivoting for every step will lead to smaller errors being introduced through numerical inaccuracies, but the continual reordering will slow down the calculation.

The philosophy behind full pivoting is much the same as that behind partial pivoting. The main difference is that the numerically largest value in the column or row containing the value to be replaced. In our example above the magnitude of element $i = 2, j = 3$ is the greatest in row 2 or column 2. We could rearrange the columns in order to bring this element onto the diagonal. This will also entail a rearrangement of the solution vector x. The rearranged system becomes, interchanging columns two and three,

$$\begin{pmatrix} 1 & 6 & 3 & 4 \\ 0 & 198 & 10^{-8} & 19 \\ 0 & 51 & -91 & 9 \\ 0 & 76 & 7 & 541 \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_2 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}.$$

The ultimate degree of accuracy can be provided by rearranging both rows and columns so that the numerically largest value in the submatrix not yet processed is brought onto the diagonal. This process may be undertaken for every step, or only when the value on the diagonal is considered too small relative to the other values in the matrix. In our case, the matrix element at $i = 4, j = 4$ is the largest. We could here interchange rows two and four and then columns two and four to bring this matrix element at the diagonal position $i = 2, j = 2$. When interchanging columns and rows, one needs to keep track of all permutations performed. Partial and full pivoting are discussed in most texts on numerical linear algebra. For an in depth discussion we recommend again the text of Golub and Van Loan [25], in particular chapter three. See also the discussion of chapter two in Ref. [22]. The library functions you end up using, be it via Matlab, the library included with this text or other ones, do all include pivoting.

If it is not possible to rearrange the columns or rows to remove a zero from the diagonal, then the matrix A is singular and no solution exists.

Gaussian elimination requires however many floating point operations. An $n \times n$ matrix requires for the simultaneous solution of a set of r different right-hand sides, a total of $n^3/3 + rn^2 - n/3$ multiplications. Adding the cost of additions, we end up with $2n^3/3 + O(n^2)$ floating point operations, see Kress [29] for a proof. An $n \times n$ matrix of dimensionalty $n = 10^3$ requires, on a modern PC with a processor that allows for something like 10^9 floating point operations per second (flops), approximately one second. If you increase the size of the matrix to $n = 10^4$ you need 1000 seconds, or roughly 16 minutes.

Although the direct Gaussian elmination algorithm allows you to compute the determinant of $\mathbf{A}$ via the product of the diagonal matrix of the triangular matrix, it is seldomly used in normal applications. The more practical elimination is provided by what is called lower and upper decomposition. Once decomposed, one can use this matrix to solve many other linear systems which use the same matrix $\mathbf{A}$, viz with different right-hand sides. With an LU decomposed matrix, the number of floating point operations for solving a set of linear equations scales as $O(n^2)$. One should however remember that to obtain the LU decompsed matrix requires roughly $O(n^3)$ floating point operations. Finally, LU decomposition allows for an efficient computation of the inverse of $\mathbf{A}$.

4.4.2 LU decomposition of a matrix

A frequently used form of Gaussian elimination is L(ower)U(pper) factorisation also known as LU Decomposition or Crout or Dolittle factorisation. In this section we describe how one can decompose a

matrix A in terms of a matrix B with elements only below the diagonal (and thereby the naming lower) and a matrix C which contains both the diagonal and matrix elements above the diagonal (leading to the labelling upper). Consider again the matrix $\mathbf{A}$ given in Eq. (4.1). The LU decomposition method means that we can rewrite this matrix as the product of two matrices $\mathbf{B}$ and $\mathbf{C}$ where

$$\mathbf{A} = \mathbf{BC} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ b_{21} & 1 & 0 & 0 \\ b_{31} & b_{32} & 1 & 0 \\ b_{41} & b_{42} & b_{43} & 1 \end{pmatrix} \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ 0 & c_{22} & c_{23} & c_{24} \\ 0 & 0 & c_{33} & c_{34} \\ 0 & 0 & 0 & c_{44} \end{pmatrix}. \quad (4.27)$$

LU decomposition forms the backbone of other algorithms in linear algebra, such as the solution of linear equations given by

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= w_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= w_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= w_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= w_4. \end{aligned}$$

The above set of equations is conveniently solved by using LU decomposition as an intermediate step, see the next subsection for more details on how to solve linear equations with an LU decomposed matrix.

The matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ has an LU factorization if the determinant is different from zero. If the LU factorization exists and $\mathbf{A}$ is non-singular, then the LU factorization is unique and the determinant is given by

$$det\{\mathbf{A}\} = c_{11}c_{22}\dots c_{nn}.$$

For a proof of this statement, see chapter 3.2 of Ref. [25].

The algorithm for obtaining B and C is actually quite simple. We start always with the first column. In our simple (4×4) case we obtain then the following equations for the first column

$$\begin{aligned} a_{11} &= c_{11} \\ a_{21} &= b_{21}c_{11} \\ a_{31} &= b_{31}c_{11} \\ a_{41} &= b_{41}c_{11}, \end{aligned}$$

which determine the elements c_{11}, b_{21}, b_{31} and b_{41} in $\mathbf{B}$ and $\mathbf{C}$. Writing out the equations for the second column we get

$$\begin{aligned} a_{12} &= c_{12} \\ a_{22} &= b_{21}c_{12} + c_{22} \\ a_{32} &= b_{31}c_{12} + b_{32}c_{22} \\ a_{42} &= b_{41}c_{12} + b_{42}c_{22}. \end{aligned}$$

Here the unknowns are c_{12}, c_{22}, b_{32} and b_{42} which all can be evaluated by means of the results from the first column and the elements of $\mathbf{A}$. Note an important feature. When going from the first to the second column we do not need any further information from the matrix elements a_{i1}. This is a general property throughout the whole algorithm. Thus the memory locations for the matrix $\mathbf{A}$ can be used to store the calculated matrix elements of $\mathbf{B}$ and $\mathbf{C}$. This saves memory.

We can generalize this procedure into three equations

$$\begin{aligned} i < j: \quad & b_{i1}c_{1j} + b_{i2}c_{2j} + \cdots + b_{ii}c_{ij} = a_{ij} \\ i = j: \quad & b_{i1}c_{1j} + b_{i2}c_{2j} + \cdots + b_{ii}c_{jj} = a_{ij} \\ i > j: \quad & b_{i1}c_{1j} + b_{i2}c_{2j} + \cdots + c_{ij}c_{jj} = a_{ij} \end{aligned}$$

which gives the following algorithm:
Calculate the elements in **B** and **C** columnwise starting with column one. For each column (j):

- Compute the first element c_{1j} by
$$c_{1j} = a_{1j}.$$
- Next, we calculate all elements $c_{ij}, i = 2, \ldots, j-1$
$$c_{ij} = a_{ij} - \sum_{k=1}^{i-1} b_{ik}c_{kj}.$$
- Then calculate the diagonal element c_{jj}
$$c_{jj} = a_{jj} - \sum_{k=1}^{j-1} b_{jk}c_{kj}. \tag{4.28}$$
- Finally, calculate the elements $b_{ij}, i > j$
$$b_{ij} = \frac{1}{c_{jj}}\left(a_{ij} - \sum_{k=1}^{i-1} b_{ik}c_{kj}\right), \tag{4.29}$$

The algorithm is known as Doolittle's algorithm since the diagonal matrix elements of $\mathbf{B}$ are 1 along the diagonal. For the case where the diagonal elements of $\mathbf{C}$ are 1 along the diagonal, we have what is called Crout's algorithm. For the case where $\mathbf{C} = \mathbf{B}^T$ so that $c_{ii} = b_{ii}$ for $1 \leq i \leq n$ we can use what is called the Cholesky factorization algorithm. In this case the matrix $\mathbf{A}$ has to fulfil several features; namely, it should be real, symmetric and positive definite. A matrix is positive definite if the quadratic form $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$. Establishing this feature is not easy since it implies the use of an arbitrary vector $\mathbf{x} \neq 0$. If the matrix is positive definite and symmetric, its eigenvalues are always real and positive. We discuss the Cholesky factorization below.

A crucial point in the LU decomposition is obviously the case where c_{jj} is close to or equals zero, a case which can lead to serious problems. Consider the following simple 2×2 example taken from Ref. [28]
$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$
The algorithm discussed above fails immediately, the first step simple states that $c_{11} = 0$. We could change slightly the above matrix by replacing 0 with 10^{-20} resulting in
$$\mathbf{A} = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 1 \end{pmatrix},$$
yielding
$$\begin{aligned} c_{11} &= 10^{-20} \\ b_{21} &= 10^{20} \end{aligned}$$
and $c_{12} = 1$ and
$$c_{11} = a_{11} - b_{21} = 1 - 10^{20},$$

we obtain

$$\mathbf{B} = \begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix},$$

and

$$\mathbf{C} = \begin{pmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{pmatrix},$$

With the change from 0 to a small number like 10^{-20} we see that the LU decomposition is now stable, but it is not backward stable. What do we mean by that? First we note that the matrix $\mathbf{C}$ has an element $c_{22} = 1 - 10^{20}$. Numerically, since we do have a limited precision, which for double precision is approximately $\epsilon_M \sim 10^{-16}$ it means that this number is approximated in the machine as $c_{22} \sim -10^{20}$ resulting in a machine representation of the matrix as

$$\mathbf{C} = \begin{pmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{pmatrix}.$$

If we multiply the matrices $\mathbf{BC}$ we have

$$\begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix} \begin{pmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{pmatrix} = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 0 \end{pmatrix} \neq \mathbf{A}.$$

We do not get back the original matrix $\mathbf{A}$!

The solution is pivoting (interchanging rows) around the largest element in a column j. Then we are actually decomposing a rowwise permutation of the original matrix $\mathbf{A}$. The key point to notice is that Eqs. (4.28) and (4.29) are equal except for the case that we divide by c_{jj} in the latter one. The upper limits are always the same $k = j - 1 (= i - 1)$. This means that we do not have to choose the diagonal element c_{jj} as the one which happens to fall along the diagonal in the first instance. Rather, we could promote one of the undivided b_{ij}'s in the column $i = j + 1, \ldots N$ to become the diagonal of C. The partial pivoting in Crout's or Doolittle's methods means then that we choose the largest value for c_{jj} (the pivot element) and then do the divisions by that element. Then we need to keep track of all permutations performed. For the above matrix $\mathbf{A}$ it would have sufficed to interchange the two rows and start the LU decomposition with

$$\mathbf{A} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

The error which is done in the LU decomposition of an $n \times n$ matrix if no zero pivots are encountered is given by, see chapter 3.3 of Ref. [25],

$$\mathbf{BC} = \mathbf{A} + \mathbf{H},$$

with

$$|\mathbf{H}| \leq 3(n-1)\mathbf{u}\left(|\mathbf{A}| + |\mathbf{B}||\mathbf{C}|\right) + O(\mathbf{u}^2),$$

with $|\mathbf{H}|$ being the absolute value of a matrix and $\mathbf{u}$ is the error done in representing the matrix elements of the matrix $\mathbf{A}$ as floating points in a machine with a given precision ϵ_M, viz. every matrix element of $\mathbf{u}$ is

$$|fl(a_{ij}) - a_{ij}| \leq u_{ij},$$

with $|u_{ij}| \leq \epsilon_M$ resulting in

$$|fl(\mathbf{A}) - \mathbf{A}| \leq \mathbf{u}|\mathbf{A}|.$$

The programs which perform the above described LU decomposition are called as follows

C++: ludcmp(double $**$a, int n, int $*$indx, double $*$d)
Fortran 90/95: CALL lu_decompose(a, n, indx, d)

Both the C++ and Fortran 90/95 programs receive as input the matrix to be LU decomposed. In C++ this is given by the double pointer $**$a. Further, both functions need the size of the matrix n. It returns the determinant d, a pointer $indx$ with the number of permutations and the LU decomposed matrix. Note that the original matrix is destroyed. If you need to care of it during the calculations, you should transfer another matrix variable.

The codes are listed in the program libraries, see under programs/cplusplus Library/lib.cpp and programs/Fortran90 Library/f90lib.f90 for the C++ and Fortran 90/95 libraries, respectively.

Cholesky's factorization

If the matrix A is real, symmetric and positive definite, then it has a unique factorization (called Cholesky factorization)

$$A = LU = LL^T$$

where L^T is the upper matrix, implying that

$$L^T_{ij} = L_{ji}.$$

The algorithm for the Cholesky decomposition is a special case of the general LU-decomposition algorithm. The algorithm of this decomposition is as follows

- Calculate the diagonal element L_{ii} by setting up a loop for $i = 0$ to $i = n - 1$ (C++ indexing of matrices and vectors)

$$L_{ii} = \left(A_{ii} - \sum_{k=0}^{i-1} L^2_{ik} \right)^{1/2}.$$

- within the loop over i, introduce a new loop which goes from $j = i + 1$ to $n - 1$ and calculate

$$L_{ji} = \frac{1}{L_{ii}} \left(A_{ij} - \sum_{k=0}^{i-1} L_{ik} l_{jk} \right).$$

For the Cholesky algorithm we have always that $L_{ii} > 0$ and the problem with exceedingly large matrix elements does not appear and hence there is no need for pivoting.

To decide whether a matrix is positive definite or not needs some careful analysis. To find criteria for positive definiteness, one needs two statements from matrix theory, see Golub and Van Loan [25] for examples. First, the leading principal submatrices of a positive definite matrix are positive definite and non-singular and secondly a matrix is positive definite if and only if it has an $\mathbf{LDL}^T$ factorization with positive diagonal elements only in the diagonal matrix $\mathbf{D}$. A positive definite matrix has to be symmetric and have only positive eigenvalues.

The easiest way therefore to test whether a matrix is positive definite or not is to solve the eigenvalue problem $\mathbf{Ax} = \lambda\mathbf{x}$ and check that all eigenvalues are positive.

4.4.3 Solution of linear systems of equations

With the LU decomposition it is rather simple to solve a system of linear equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= w_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= w_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= w_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= w_4. \end{aligned}$$

This can be written in matrix form as

$$\mathbf{A}\mathbf{x} = \mathbf{w}.$$

where $\mathbf{A}$ and $\mathbf{w}$ are known and we have to solve for $\mathbf{x}$. Using the LU dcomposition we write

$$\mathbf{A}\mathbf{x} \equiv \mathbf{B}\mathbf{C}\mathbf{x} = \mathbf{w}. \tag{4.30}$$

This equation can be calculated in two steps

$$\mathbf{B}\mathbf{y} = \mathbf{w}; \qquad \mathbf{C}\mathbf{x} = \mathbf{y}. \tag{4.31}$$

To show that this is correct we use to the LU decomposition to rewrite our system of linear equations as

$$\mathbf{B}\mathbf{C}\mathbf{x} = \mathbf{w},$$

and since the determinat of $\mathbf{B}$ is equal to 1 (by construction since the diagonals of $\mathbf{B}$ equal 1) we can use the inverse of $\mathbf{B}$ to obtain

$$\mathbf{C}\mathbf{x} = \mathbf{B}^{-\mathbf{1}}\mathbf{w} = \mathbf{y},$$

which yields the intermediate step

$$\mathbf{B}^{-\mathbf{1}}\mathbf{w} = \mathbf{y}$$

and multiplying with $\mathbf{B}$ on both sides we reobtain Eq. (4.31). As soon as we have $\mathbf{y}$ we can obtain $\mathbf{x}$ through $\mathbf{C}\mathbf{x} = \mathbf{y}$.

For our four-dimentional example this takes the form

$$\begin{aligned} y_1 &= w_1 \\ b_{21}y_1 + y_2 &= w_2 \\ b_{31}y_1 + b_{32}y_2 + y_3 &= w_3 \\ b_{41}y_1 + b_{42}y_2 + b_{43}y_3 + y_4 &= w_4. \end{aligned}$$

and

$$\begin{aligned} c_{11}x_1 + c_{12}x_2 + c_{13}x_3 + c_{14}x_4 &= y_1 \\ c_{22}x_2 + c_{23}x_3 + c_{24}x_4 &= y_2 \\ c_{33}x_3 + c_{34}x_4 &= y_3 \\ c_{44}x_4 &= y_4 \end{aligned}$$

This example shows the basis for the algorithm needed to solve the set of n linear equations. The algorithm goes as follows

– Set up the matrix **A** and the vector **w** with their correct dimensions. This determines the dimensionality of the unknown vector **x**.

– Then LU decompose the matrix **A** through a call to the function

C++:	ludcmp(double $**$a, int n, int $*$indx, double $*$d)
Fortran 90/95:	CALL lu_decompose(a, n, indx, d)

This functions returns the LU decomposed matrix **A**, its determinant and the vector indx which keeps track of the number of interchanges of rows. If the determinant is zero, the solution is malconditioned.

– Thereafter you call the function

C++:	lubksb(double $**$a, int n, int $*$indx, double $*$w
Fortran 90/95:	CALL lu_linear_equation(a, n, indx, w)

which uses the LU decomposed matrix **A** and the vector **w** and returns **x** in the same place as **w**. Upon exit the original content in **w** is destroyed. If you wish to keep this information, you should make a backup of it in your calling function.

The codes are listed in the program libraries, see under programs/cplusplus Library/lib.cpp and programs/Fortran90 Library/f90lib.f90 for the C++ and Fortran 90/95 libraries, respectively.

4.4.4 Inverse of a matrix and the determinant

The basic definition of the determinant of **A** is

$$det\{\mathbf{A}\} = \sum_p (-)^p a_{1p_1} \cdot a_{2p_2} \cdots a_{np_n},$$

where the sum runs over all permutations p of the indices $1, 2, \ldots, n$, altogether $n!$ terms. Also to calculate the inverse of **A** is a formidable task. Here we have to calculate *the complementary cofactor* a^{ij} of each element $a_i j$ which is the $(n-1)$determinant obtained by striking out the row i and column j in which the element a_{ij} appears. The inverse of **A** is then constructed as the transpose a matrix with the elements $(-)^{i+j}a^{ij}$. This involves a calculation of n^2 determinants using the formula above. A simplified method is highly needed.

With the LU decomposed matrix **A** in Eq. (4.27) it is rather easy to find the determinant

$$det\{\mathbf{A}\} = det\{\mathbf{B}\} \times det\{\mathbf{C}\} = det\{\mathbf{C}\},$$

since the diagonal elements of **B** equal 1. Thus the determinant can be written

$$det\{\mathbf{A}\} = \prod_{k=1}^{N} c_{kk}.$$

The inverse is slightly more difficult to obtain from the LU decomposition. It is formally defined as

$$\mathbf{A}^{-1} = \mathbf{C}^{-1}\mathbf{B}^{-1}.$$

We use this form since the computation of the inverse goes through the inverse of the matrices **B** and **C**. The reason is that the inverse of a lower (upper) triangular matrix is also a lower (upper) triangular matrix. If we call **D** for the inverse of **B**, we can determine the matrix elements of **D** through the equation

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ b_{21} & 1 & 0 & 0 \\ b_{31} & b_{32} & 1 & 0 \\ b_{41} & b_{42} & b_{43} & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ d_{21} & 1 & 0 & 0 \\ d_{31} & d_{32} & 1 & 0 \\ d_{41} & d_{42} & d_{43} & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

which gives the following general algorithm

$$d_{ij} = -b_{ij} - \sum_{k=j+1}^{i-1} b_{ik} d_{kj}, \tag{4.32}$$

which is valid for $i > j$. The diagonal is 1 and the upper matrix elements are zero. We solve this equation column by column (increasing order of j). In a similar way we can define an equation which gives us the inverse of the matrix **C**, labelled **E** in the equation below. This contains only non-zero matrix elements in the upper part of the matrix (plus the diagonal ones)

$$\begin{pmatrix} e_{11} & e_{12} & e_{13} & e_{14} \\ 0 & e_{22} & e_{23} & e_{24} \\ 0 & 0 & e_{33} & e_{34} \\ 0 & 0 & 0 & e_{44} \end{pmatrix} \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ 0 & c_{22} & c_{23} & c_{24} \\ 0 & 0 & c_{33} & c_{34} \\ 0 & 0 & 0 & c_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

with the following general equation

$$e_{ij} = -\frac{1}{c_{jj}} \sum_{k=1}^{j-1} e_{ik} c_{kj}. \tag{4.33}$$

for $i \leq j$.

A calculation of the inverse of a matrix could then be implemented in the following way:

- Set up the matrix to be inverted.
- Call the LU decomposition function.
- Check whether the determinant is zero or not.
- Then solve column by column Eqs. (4.32, 4.33).

The following codes compute the inverse of a matrix using either C++ or Fortran 90/95 as programming languages. They are both included in the library packages, but we include them explicitely here as well as two distinct programs. We list first the C++ code

`http://folk.uio.no/mhjensen/fys3150/2005/programs/chapter4/program1.cpp`

```
/* The function
**                  inverse()
```

```
** perform a mtx inversion of the input matrix a[][] with
** dimension n. The method is described in Numerical Recipes
** sect. 2.3, page 48.
*/
void inverse(double **a, int n)
{
  int           i,j, *indx;
  double        d, *col, **y;

  // allocate space in memory
  indx = new int[n];
  col  = new double[n];
  y    = (double **) matrix(n, n, sizeof(double));
  // first we need to LU decompose the matrix
  ludcmp(a, n, indx, &d);
  // find inverse of a[][] by columns
  for(j = 0; j < n; j++) {
    // initialize right-side of linear equations
    for(i = 0; i < n; i++) col[i] = 0.0;
    col[j] = 1.0;
    lubksb(a, n, indx, col);
    // save result in y[][]
    for(i = 0; i < n; i++) y[i][j] = col[i];
  }
  // return the inverse matrix in a[][]

  for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++) a[i][j] = y[i][j];
  }
  free_matrix((void **) y);      // release local memory
  delete [] col;
  delete []indx;

}  // End: function inverse()
```

We first need to LU decompose the matrix. Thereafter we solve Eqs. (4.32) and (4.33) by using the back substitution method calling the function **lubksb** and obtain finally the inverse matrix.

An example of a C++ function which calls this function is also given in the program and reads

`http://folk.uio.no/mhjensen/fys3150/2005/programs/chapter4/program1.cpp`

```
//  Simple matrix inversion example
#include <iostream>
#include <new>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cstring>
#include    "lib.h"

using namespace std;

/* function declarations */
```

```
void inverse(double **, int);
/*
** This program sets up a simple 3x3 symmetric matrix
** and finds its determinant and inverse
*/

int main()
{
  int           i, j, k, result, n = 3;
  double        **matr, sum,
    a[3][3]    = { {1.0, 3.0, 4.0},
                   {3.0, 4.0, 6.0},
                   {4.0, 6.0, 8.0}};
  // memory for  inverse matrix
  matr = (double **) matrix(n, n, sizeof(double));
  // various print statements in the original code are omitted

  inverse(matr, n);      // calculate and return inverse matrix
  ....
  return 0;
} // End: function main()
```

In order to use the program library you need to include the **lib.h** file using the `#include "lib.h"` statement. This function utilizes the library function **matrix** and **free_matrix** to allocate and free memory during execution. The matrix $a[3][3]$ is set at compilation time. The corresponding Fortran 90/95 program for the inverse of a matrix reads

`http://folk.uio.no/mhjensen/fys3150/2005/programs/F90library/f90lib.f90`

```
  !
  !             Routines to do mtx inversion, from Numerical
  !             Recipes, Teukolsky et al. Routines included
  !             below are MATINV, LUDCMP and LUBKSB. See chap 2
  !             of Numerical Recipes for further details
  !
  SUBROUTINE matinv(a,n, indx, d)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: n
    INTEGER :: i, j
    REAL(DP), DIMENSION(n,n), INTENT(INOUT)   :: a
    REAL(DP), ALLOCATABLE :: y(:,:)
    REAL(DP) :: d
    INTEGER, , INTENT(INOUT) :: indx(n)

    ALLOCATE (y( n, n))
    y=0.
    !      setup identity matrix
    DO i=1,n
       y(i,i)=1.
    ENDDO
    !      LU decompose the matrix just once
    CALL  lu_decompose(a,n,indx,d)

    !      Find inverse by columns
```

```
    DO j=1,n
       CALL lu_linear_equation(a,n,indx,y(:,j))
    ENDDO
    !      The original matrix a was destroyed, now we equate it with the
        inverse y
    a=y
    DEALLOCATE ( y )

  END SUBROUTINE matinv
```

The Fortran 90/95 program **matinv** receives as input the same variables as the C++ program and calls the function for LU decomposition **lu_decompose** and the function to solve sets of linear equations **lu_linear_equation**. The program listed under programs/chapter4/program1.f90 performs the same action as the C++ listed above. In order to compile and link these programs it is convenient to use a so-called **makefile**. Examples of these are found under the same catalogue as the above programs.

Inverse of the Vandermonde matrix

In chapter 6 we discuss how to interpolate a function f which is known only at $n+1$ points $x_0, x_1, x_2, \ldots, x_n$ with corresponding values $f(x_0), f(x_1), f(x_2), \ldots, f(x_n)$. The latter is often a typical outcome of a large scale computation or from an experiment. In most cases in the sciences we do not have a closed form expressions for a function f. The function is only known at specific points.

We seek a functional form for a function f which passes through the above pairs of values $(x_0, f(x_0))$,$(x_1, f(x_1))$, $(x_2, f(x_2)), \ldots, (x_n, f(x_n))$. This is normally achieved by expanding the function $f(x)$ in terms of well-known polynomials $\phi_i(x)$, such as Legendre, Chebyshev, Laguerre etc. The function is then approximated by a polynomial of degree n $p_n(x)$

$$f(x) \approx p_n(x) = \sum_{i=0}^{n} a_i \phi_i(x),$$

where a_i are unknown coefficients and $\phi_i(x)$ are a priori well-known functions. The simplest possible case is to assume that $\phi_i(x) = x^i$, resulting in an approximation

$$f(x) \approx a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.$$

Our function is known at the points $n+1$ points $x_0, x_1, x_2, \ldots, x_n$, leading to $n+1$ equations of the type

$$f(x_i) \approx a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_n x_i^n.$$

We can then obtain the unknown coefficients by rewriting our problem as

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & \ldots & x_1^n \\ 1 & x_2 & x_2^2 & \ldots & \ldots & x_2^n \\ 1 & x_3 & x_3^2 & \ldots & \ldots & x_3^n \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & x_n & x_n^2 & \ldots & \ldots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \ldots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ f(x_3) \\ \ldots \\ f(x_n) \end{pmatrix},$$

an expression which can be rewritten in a more compact form as

$$\mathbf{Xa} = \mathbf{f},$$

with

$$\mathbf{X} = \begin{pmatrix} 1 & x_0 & x_0^2 & \dots & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & \dots & x_2^n \\ 1 & x_3 & x_3^2 & \dots & \dots & x_3^n \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & x_n & x_n^2 & \dots & \dots & x_n^n \end{pmatrix}.$$

. This matrix is called a Vandermonde matrix and is by definition non-singular since all points x_i are different. The inverse exists and we can obtain the unknown coefficients by inverting $\mathbf{X}$, resulting in

$$\mathbf{a} = \mathbf{X}^{-1}\mathbf{f}.$$

Although this algorithm for obtaining an interpolating polynomial which approximates our data set looks very simple, it is an inefficient algorithm since the computation of the inverse requires $O(n^3)$ flops. The methods we will discuss in chapter 6 are much more effective from a numerical point of view. There is also another subtle point. Although we have a data set with $n+1$ points, this does not necessarily mean that our function $f(x)$ is well represented by a polynomial of degree n. On the contrary, our function $f(x)$ may be a parabola (second-order in n), meaning that we have a large excess of data points. In such cases a least-square fit or a spline interpolation may be better approaches to represent the function. These techniques are discussed in chapter 6.

4.4.5 Tridiagonal systems of linear equations

We start with the linear set of equations from Eq. (4.24), viz

$$\mathbf{A}\mathbf{u} = \mathbf{f},$$

where $\mathbf{A}$ is a tridiagonal matrix which we rewrite as

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \tag{4.34}$$

where a, b, c are one-dimensional arrays of length $1:n$. In the example of Eq. (4.24) the arrays a and c are equal, namely $a_i = c_i = -1/h^2$. We can rewrite Eq. (4.24) as

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \dots \\ \dots \\ \dots \\ f_n \end{pmatrix}.$$

A tridiagonal matrix is a special form of banded matrix where all the elements are zero except for those on and immediately above and below the leading diagonal. The above tridiagonal system can be written as

$$a_i u_{i-1} + b_i u_i + c_i u_{i+1} = f_i,$$

for $i = 1, 2, \ldots, n$. We see that u_{-1} and u_{n+1} are not required and we can set $a_1 = c_n = 0$. In many applications the matrix is symmetric and we have $a_i = c_i$. The algorithm for solving this set of equations is rather simple and requires two steps only, a forward substitution and a backward substitution. These steps are also common to the algorithms based on Gaussian elimination that we will discussed previously. However, due to its simplicity, the number of floating point operations is in this case proportional with $O(n)$ while Gaussian elimination requires $2n^3/3 + O(n^2)$ floating point operations. In case your system of equations leads to a tridiagonal matrix, it is clearly an overkill to employ Gaussian elimination or the standard LU decomposition. You will encounter several applications involving tridiagonal matrices in our discussion of partial differential equations in chapter 15.

Our algorithm starts with forward substitution with a loop over of the elements i and can be expressed via the following code piece of code taken from the Numerical Recipe text of Teukolsky *et al* [22]

```
btemp = b[1];
u[1] = f[1]/btemp;
for(i=2 ; i <= n ; i++) {
   temp[i] = c[i-1]/btemp;
   btemp = b[i]-a[i]*temp[i];
   u[i] = (f[i] - a[i]*u[i-1])/btemp;
}
```

Note that you should avoid cases with $b_1 = 0$. If that is the case, you should rewrite the equations as a set of order $n - 1$ with u_2 eliminated. Finally we perform the backsubstitution leading to the following code

```
for(i=n-1 ; i >= 1 ; i--) {
   u[i] -= temp[i+1]*u[i+1];
}
```

Note that our sums start with $i = 1$ and that one should avoid cases with $b_1 = 0$. If that is the case, you should rewrite the equations as a set of order $n - 1$ with u_2 eliminated. However, a tridiagonal matrix problem is not a guarantee that we can find a solution. The matrix $\mathbf{A}$ which rephrases a second derivative in a discretized form

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix},$$

fulfills the condition of a weak dominance of the diagonal, with $|b_1| > |c_1|$, $|b_n| > |a_n|$ and $|b_k| \geq |a_k| + |c_k|$ for $k = 2, 3, \ldots, n-1$. This is a relevant but not sufficient condition to guarantee that the matrix $\mathbf{A}$ yields a solution to a linear equation problem. The matrix needs also to be irreducible. A tridiagonal irreducible matrix means that all the elements a_i and c_i are non-zero. If these two conditions are present, then $\mathbf{A}$ is nonsingular and has a unique LU decomposition.

We can obviously extend our boundary value problem to include a first derivative as well

$$-\frac{d^2u(x)}{dx^2} + g(x)\frac{du(x)}{dx} + h(x)u(x) = f(x),$$

with $x \in [a, b]$ and with boundary conditions $u(a) = u(b) = 0$. We assume athat f, g and h are continuous functions in the domain $x \in [a, b]$ and that $h(x) \geq 0$. Then the differential equation has a unique solution. We subdivide our interval $x \in [a, b]$ into n subintervals by setting $x_i = ih$, with

$i = 0, 1, \ldots, n+1$. The step size is then given by $h = (b-a)/(n+1)$ with $n \in \mathbb{N}$. For the internal grid points $i = 1, 2, \ldots n$ we replace the differential operators with

$$u_i^{''} \approx \frac{u_{i+1} - 2u_i + u_{i-i}}{h^2}.$$

for the second derivative while the first derivative is given by

$$u_i^{'} \approx \frac{u_{i+1} - u_{i-i}}{2h}.$$

We rewrite our original differential equation in terms of a discretized equation as

$$-\frac{u_{i+1} - 2u_i + u_{i-i}}{h^2} + g_i\frac{u_{i+1} - u_{i-i}}{2h} + h_i u_i = f_i,$$

with $i = 1, 2, \ldots, n$. We need to add to this system the two boundary conditions $u(a) = u_0$ and $u(b) = u_{n+1}$. This equation can again be rewritten as a tridiagonal matrix problem. We leave it as an exercise to the reader to find the matrix elements, find the conditions for having weakly dominant diagonal elements and that the matrix is irreducible.

4.5 Singular value decomposition

In preparation, Fall 2008

4.6 QR decomposition

In preparation, Fall 2008

4.7 Handling sparse matrices

In preparation, Fall 2008

4.8 Classes, templates and Blitz++

In the above pseudocode for solving a system of linear equations, we started our indexing from $i = 1$. This was done because in many mathematical expressions the indices of vectors and matrices would typically start from $i = 1$. Quite often we would like our codes to mimick the mathematical expressions we derive as much as possible.

In Fortran a vector or matrix start with 1, but it is to change a vector so that it starts with zero or even a negative number. If we have a double precision Fortran vector which starts at -10 and ends at 10, we could declare it as REAL(KIND=8):: vector$(-10{:}10)$. Similarly, if we want to start at zero and end at 10 we could write REAL(KIND=8):: vector$(0{:}10)$. We have also seen that Fortran 90/95 allows us to write a matrix addition $\mathbf{A} = \mathbf{B} + \mathbf{C}$ as A = B + C. This means that we have overloaded the addition operator so that it translates this operation into two loops and an addition of two matrix elements $a_{ij} = b_{ij} + c_{ij}$.

The way the matrix addition is written is very close to the way we express this relation mathematically. The benefit for the programmer is that our code is easier to read. Furthermore, such a way of coding makes it more likely to spot eventual errors as well.

In Ansi C and C++ arrays start by default from $i = 0$. Moreover, if we wish to add two matrices we need to explicitely write out the two loops as

```
for(i=0 ; i < n ; i++) {
   for(j=0 ; j < n ; j++) {
      a[i][j]=b[i][j]+c[i][j]
   }
}
```

However, the strength of C++ over programming languages like C and Fortran 77 is the possibility to define new data types, tailored to some particular problem. Via new data types and overloading of operations such as addition and subtraction, we can easily define sets of operations and data types which allow us to write a matrix addition in exactly the same way as we would do in Fortran 90/95. We could also change the way we declare a C++ matrix elements a_{ij}, from $a[i][j]$ to say $a(i,j)$, as we would do in Fortran 90/95. Similarly, we could also change the default range from $0 : n-1$ to $1 : n$.

To achieve this we need to introduce two important entities in C++ programming, classes and templates. Till now, except for a brief encounter in the previous chapter on how to handle files in C++, we have not defined properly this programming feature.

The function and class declarations are fundamental concepts within C++. Functions are abstractions which encapsulate an algorithm or parts of it and perform specific tasks in a program. We have already met several examples on how to use functions. Classes can be defined as abstractions which encapsulate data and operations on these data. The data can be very complex data structures and the class can contain particular functions which operate on these data. Classes allow therefore for a higher level of abstraction in computing. The elements (or components) of the data type are the class data members, and the procedures are the class member functions.

Classes are user-defined tools used to create multi-purpose software which can be reused by other classes or functions. These user-defined data types contain data (variables) and functions operating on the data.

A simple example is that of a point in two dimensions. The data could be the x and y coordinates of a given point. The functions we define could be simple read and write functions or the possibility to compute the distance between two points.

The two examples we elaborate on below[3] demonstrate most of the features of classes. We develop first a class called Complex which allows us to perform various operations on complex variables. In appendix A we extend our discussion of classes to define a class vector_operations which allows us to perform various operations on a user-specified one-dimesional array, from declarations of a vector to mathematical operations such as additions of vectors.

The classes we define are easy to use in other codes and/or other classes and many of the details which would be present in C (or Fortran 77) codes are hidden inside the class. The reuse of a well-written and functional class is normally rather simple. However, to write a given class is often complicated, especially if we deal with complicated matrix operations. In this text we will rely on ready-made classes in C++ for dealing with matrix operations. We have chosen to use the Blitz++ library, discussed below. This library hides for us many low-level operations with matrices and vectors, such as matrix-vector multiplications

[3]These examples are taken from the course INF-VERK3830, see `http://heim.ifi.uio.no/~hpl/INF-VERK4830/` for more information.

or allocation and deallocation of memory. Such libraries make it then easier to build our own high-level classes out of well-tested lower-level classes.

The way we use classes in this text is close to the MODULE data type in Fortran90/95 and we provide some simple demonstrations of the latter as well in appendix A.

In this text we will mainly use classes to encapsulate specific operations, but will not use the full power such as inheritance and other object-oriented programming concepts. For examples of the latter see Refs. [20, 19, 21]

4.8.1 The Complex class

As remarked in chapter 2, C++ has a class complex in its standard template library (STL). The standard usage in a given function could then look like

```
// Program to calculate addition and multiplication of two complex numbers
using namespace std;
#include <iostream>
#include <cmath>
#include <complex>
int main()
{
  complex<double> x(6.1,8.2), y(0.5,1.3);
  // write out x+y
  cout << x + y << x*y  << endl;
  return 0;
}
```

where we add and multiply two complex numbers $x = 6.1 + i8.2$ and $y = 0.5 + i1.3$ with the obvious results $z = x + y = 6.6 + i9.5$ and $z = x \cdot y = -7.61 + i12.03$. In Fortran90/95 we would declare the above variables as COMPLEX(DPC):: x(6.1,8.2), y (0.5,1.3) .

The library Blitz++ includes an extension of the complex class to operations on vectors, matrices and higher-dimensional arrays. We recommend the use of Blitz++ when you develop your own codes. However, writing a complex class yourself is a good pedagogical exercise.

We proceed by splitting our task in three files.

- We define first a header file complex.h which contains the declarations of the class. The header file contains the class declaration (data and functions), declaration of stand-alone functions, and all inlined functions, starting as follows

```
#ifndef Complex_H
#define Complex_H
//   various include statements and definitions
#include <iostream>          // Standard ANSI-C++ include files
#include <new>
#include ....

class Complex
{...
definition of variables and their character
};
//   declarations of various functions used by the class
...
#endif
```

– Next we provide a file complex.cpp where the code and algorithms of different functions (except inlined functions) declared within the class are written. The files complex.h and complex.cpp are normally placed in a directory with other classes and libraries we have defined.

– Finally,we discuss here an example of a main program which uses this particular class. An example of a program which uses our complex class is given below. In particular we would like our class to perform tasks like declaring complex variables, writing out the real and imaginary part and performing algebraic operations such as adding or multiplying two complex numbers.

```
#include "Complex.h"
...  other include and declarations
int main ()
{
  Complex a(0.1,1.3);     // we declare a complex variable a
  Complex b(3.0), c(5.0,-2.3);  // we declare  complex variables b and
     c
  Complex d = b;          //  we declare  a new complex variable d
  cout << "d=" << d << ", a=" << a << ", b=" << b << endl;
  d = a*c + b/a;  //   we add, multiply and divide two complex numbers
  cout << "Re(d)=" << d.Re() << ", Im(d)=" << d.Im() << endl;  // write
       out of the real and imaginary parts
}
```

We include the header file complex.h and define four different complex variables. These are $a = 0.1 + i1.3$, $b = 3.0 + i0$ (note that if you don't define a value for the imaginary part this is set to zero), $c = 5.0 - i2.3$ and $d = b$. Thereafter we have defined standard algebraic operations and the member functions of the class which allows us to print out the real and imaginary part of a given variable.

To achieve these features, let us see how we could define the complex class. In C++ we could define a complex class as follows

```
class Complex
{
private:
   double re, im; // real and imaginary part
public:
   Complex ();                                  // Complex c;
   Complex (double re, double im = 0.0); // Definition of a complex variable
      ;
   Complex (const Complex& c);                  // Usage: Complex c(a);    //
      equate two complex variables
   Complex& operator= (const Complex& c); // c = a;    //  equate two complex
       variables, same as previous
  ~Complex () {}                           // destructor
   double    Re () const;          // double real_part = a.Re();
   double    Im () const;          // double imag_part = a.Im();
   double    abs () const;         // double m = a.abs(); // modulus
   friend Complex operator+ (const Complex&  a, const Complex& b);
   friend Complex operator- (const Complex&  a, const Complex& b);
   friend Complex operator* (const Complex&  a, const Complex& b);
   friend Complex operator/ (const Complex&  a, const Complex& b);
};
```

The class is defined via the statement **class** Complex. We must first use the key word **class**, which in turn is followed by the user-defined variable name Complex. The body of the class, data and functions, is encapsulated within the parentheses {...};.

Data and specific functions can be private, which means that they cannot be accessed from outside the class. This means also that access cannot be inherited by other functions outside the class. If we use **protected** instead of **private**, then data and functions can be inherited outside the class. The key word **public** means that data and functions can be accessed from outside the class. Here we have defined several functions which can be accessed by functions outside the class. The declaration **friend** means that stand-alone functions can work on privately declared variables of the type (re, im). Data members of a class should be declared as private variables.

The first public function we encounter is a so-called constructor, which tells how we declare a variable of type Complex and how this variable is initialized. We have chose three possibilities in the example above:

1. A declaration like Complex c; calls the member function Complex() which can have the following implementation

```
Complex:: Complex ()    { re = im = 0.0; }
```

   meaning that it sets the real and imaginary parts to zero. Note the way a member function is defined. The constructor is the first function that is called when an object is instantiated.

2. Another possibility is

```
Complex:: Complex ()    {}
```

   which means that there is no initialization of the real and imaginary parts. The drawback is that a given compiler can then assign random values to a given variable.

3. A call like Complex a(0.1,1.3); means that we could call the member function Complex(**double**, **double**)as

```
Complex:: Complex (double re_a, double im_a)
{ re = re_a; im = im_a; }
```

The simplest member function are those we defined to extract the real and imaginary part of a variable. Here you have to recall that these are private data, that is they invisible for users of the class. We obtain a copy of these variables by defining the functions

```
double Complex:: Re () const { return re; }} //  getting the real part
double Complex:: Im () const { return im; }  //   and the imaginary part
\end{lstlistingline}
Note that we have introduced  the declaration \lstinline{const}. What
    does it mean?
This declaration means that a varibale cannot be changed within  a called
    function.
If we define a variable as
\lstinline{const double p = 3;} and then try to change its value, we will
    get an error when we
compile our program. This means that constant arguments in functions cannot
    be changed.
\begin{lstlisting}
```

```
// const arguments (in functions) cannot be changed:
void myfunc (const Complex& c)
{ c.re = 0.2; /* ILLEGAL!! compiler error... */ }
```

If we declare the function and try to change the value to 0.2, the compiler will complain by sending an error message. If we define a function to compute the absolute value of complex variable like

```
double Complex::abs () { return sqrt(re*re + im*im);}
```

without the constant declaration and define thereafter a function myabs as

```
double myabs (const Complex& c)
{ return c.abs(); }    // Not ok because c.abs() is not a const func.
```

the compiler would not allow the c.abs() call in myabs since Complex::abs is not a constant member function. Constant functions cannot change the object's state. To avoid this we declare the function abs as

```
double Complex::abs () const { return sqrt(re*re + im*im); }
```

Overloading operators

C++ (and Fortran 90/95) allow for overloading of operators. That means we can define algebraic operations on for example vectors or any arbitrary object. As an example, a vector addition of the type $\mathbf{c} = \mathbf{a} + \mathbf{b}$ means that we need to write a small part of code with a for-loop over the dimension of the array. We would rather like to write this statement as c = a+b; as this makes the code much more readable and close to eventual equations we want to code. To achieve this we need to extend the definition of operators.

Let us study the declarations in our complex class. In our main function we have a statement like d = b;, which means that we call d.**operator**= (b) and we have defined a so-called assignment operator as a part of the class defined as

```
Complex& Complex::operator= (const Complex& c)
{
    re = c.re;
    im = c.im;
    return *this;
}
```

With this function, statements like Complex d = b; or Complex d(b); make a new object d, which becomes a copy of b. We can make simple implementations in terms of the assignment

```
Complex::Complex (const Complex& c)
{ *this = c; }
```

which is a pointer to "this object", ***this** is the present object, so ***this** = c; means setting the present object equal to c, that is **this**->**operator**= (c);.

The meaning of the addition operator + for Complex objects is defined in the function Complex **operator**+ (**const** Complex& a, **const** Complex& b); *// a+b* The compiler translates c = a + b; into c = **operator** + (a, b);. Since this implies the call to function, it brings in an additional overhead. If speed is crucial and this function call is performed inside a loop, then it is more difficult for a given compiler to perform optimizations of a loop. The solution to this is to inline functions. We discussed inlining in chapter 2.

Inlining means that the function body is copied directly into the calling code, thus avoiding calling the function. Inlining is enabled by the inline keyword

```
inline Complex operator+ (const Complex& a, const Complex& b)
{ return Complex (a.re + b.re, a.im + b.im); }
```

Inline functions, with complete bodies must be written in the header file complex.h. Consider the case c = a + b; that is, c.operator= (operator+ (a,b)); If operator+, operator= and the constructor Complex(r,i) all are inline functions, this transforms to

```
c.re = a.re + b.re;
c.im = a.im + b.im;
```

by the compiler, i.e., no function calls

The stand-alone function operator+ is a friend of the Complex class

```
class Complex
{
    ...
    friend Complex operator+ (const Complex& a, const Complex& b);
    ...
};
```

so it can read (and manipulate) the private data parts re and im via

```
inline Complex operator+ (const Complex& a, const Complex& b)
{ return Complex (a.re + b.re, a.im + b.im); }
```

Since we do not need to alter the re and im variables, we can get the values by Re() and Im(), and there is no need to be a friend function

```
inline Complex operator+ (const Complex& a, const Complex& b)
{ return Complex (a.Re() + b.Re(), a.Im() + b.Im()); }
```

The multiplication functionality can now be extended to imaginary numbers by the following code

```
inline Complex operator* (const Complex& a, const Complex& b)
{
  return Complex(a.re*b.re - a.im*b.im, a.im*b.re + a.re*b.im);
}
```

It will be convenient to inline all functions used by this operator. To inline the complete expression a*b;, the constructors and operator= must also be inlined. This can be achieved via the following piece of code

```
inline Complex::Complex () { re = im = 0.0; }
inline Complex::Complex (double re_, double im_)
{ ... }
inline Complex::Complex (const Complex& c)
{ ... }
inline Complex::operator= (const Complex& c)
{ ... }
// e, c, d are complex
e = c*d;
// first compiler translation:
e.operator= (operator* (c,d));
// result of nested inline functions
```

```
// operator=, operator*, Complex(double,double=0):
e.re = c.re*d.re - c.im*d.im;
e.im = c.im*d.re + c.re*d.im;
```

The definitions **operator**– and **operator**/ follow the same set up.

Finally, if we wish to write to file or another device a complex number using the simple syntax cout << c;, we obtain this by defining the effect of << for a Complex object as

```
ostream& operator<< (ostream& o, const Complex& c)
{ o << "(" << c.Re() << "," << c.Im() << ") "; return o;}
```

Templates

The reader may have noted that all variables and some of the functions defined in our class are declared as doubles. What if we wanted to make a class which takes integers or floating point numbers with single precision? A simple way to achieve this is copy and paste our class and replace **double** with for example **int**.

C++ allows us to do this automatically via the usage of templates, which are the C++ constructs for parameterizing parts of classes. Class templates is a template for producing classes. The declaration consists of the keyword **template** followed by a list of template arguments enclosed in brackets. We can therefore make a more general class by rewriting our original example as

```
template<class T>
class Complex
{
private:
   T re, im; // real and imaginary part
public:
   Complex ();                                  // Complex c;
   Complex (T re, T im = 0); // Definition of a complex variable;
   Complex (const Complex& c);                  // Usage: Complex c(a);    //
       equate two complex variables
   Complex& operator= (const Complex& c); // c = a;    //  equate two complex
        variables, same as previous
  ~Complex () {}                            // destructor
   T    Re () const;          // T real_part = a.Re();
   T    Im () const;          // T imag_part = a.Im();
   T    abs () const;         // T m = a.abs(); // modulus
   friend Complex operator+ (const Complex&  a, const Complex& b);
   friend Complex operator- (const Complex&  a, const Complex& b);
   friend Complex operator* (const Complex&  a, const Complex& b);
   friend Complex operator/ (const Complex&  a, const Complex& b);
};
```

What it says is that Complex is a parameterized type with T as a parameter and T has to be a type such as double or float. The class complex is now a class template and we would define variables in a code as

```
Complex<double> a(10.0,5.1);
Complex<int> b(1,0);
```

Member functions of our class are defined by preceding the name of the function with the **template** keyword. Consider the function we defined as Complex:: Complex (**double** re_a, **double** im_a). We would rewrite this function as

```
template<class T>
Complex<T>:: Complex (T re_a, T im_a)
{ re = re_a; im = im_a; }
```

The member functions are otherwise defined following ordinary member function definitions.

To write a class like the above is rather straightforward. The class for handling one-dimensional arrays, presented in appendix A shows some of the additional possibilities which C++ offers. However, it can be rather difficult to write good classes for handling matrices or more complex objects. For such applications we recommend therefore the usage of ready-made libraries like Blitz++

Blitz++ `http://www.oonumerics.org/blitz/` is a C++ library whose two main goals are to improve the numerical efficiency of C++ and to extend the conventional dense array model to incorporate new and useful features. Some examples of such extensions are flexible storage formats, tensor notation and index placeholders. It allows you also to write several operations involving vectors and matrices in a simple and clear (from a mathematical point of view) way. The way you would code the addition of two matrices looks very similar to the way it is done in Fortran90/95. The C++ programming language offers many features useful for tackling complex scientific computing problems: inheritance, polymorphism, generic programming, and operator overloading are some of the most important. Unfortunately, these advanced features came with a hefty performance pricetag: until recently, C++ lagged behind Fortran's performance by anywhere from 20% to a factor of ten. It was not uncommon to read in textbooks on high-performance computing that if performance matters, then one should resort to Fortran, preferentially Fortran 77. As a result, untill very recently, the adoption of C++ for scientific computing has been slow. This has changed quite a lot in the last years and modern C++ compilers with numerical libraries have improved the situation considerably. Recent benchmarks show C++ encroaching steadily on Fortran's high-performance monopoly, and for some benchmarks, C++ is even faster than Fortran! These results are being obtained not through better optimizing compilers, preprocessors, or language extensions, but through the use of template techniques. By using templates cleverly, optimizations such as loop fusion, unrolling, tiling, and algorithm specialization can be performed automatically at compile time.

The features of Blitz++ which are useful for our studies are the dynamical allocation of vectors and matrices and algebraic operations on these objects. In particular, if you access the Blitz++ webpage at `http://www.oonumerics.org/blitz/`, we recommend that you study chapters two and three.

In this section we discuss several applications of the Blitz++ library and demonstrate the benefits when handling arrays and mathematical expressions involving arrays.

At `http://folk.uio.no/mhjensen/fys3150/2005/programs/blitz` you will find examples of makefiles, simple examples like those discussed here and the C++ library which contains the algorithms discussed in this text. You can choose whether you want to employ Blitz++ fully or just use the more old-fashioned C++ codes.

The example included here shows some of the versatility of Blitz++ when handling matrices. Note that you need to define the path where you have store Blitz++. We recommend that you study the examples available at the Blitz++ web page and the examples which follow this text.

As an example, a float matrix is defined simply as `Array<float,2> A(r,r);`. As the example shows we can even change the range of the matrix from the standard which starts at 0 and ends at $n-1$ to one which starts at 1 and ends at n. This can be useful if you deal with matrices from a Fortran code or if you wish to code your matrix operation following the way you index the matrix elements.

You can also easily initialise to zero your matrix by simply writing `A=0.;`. Note also the way you can fill in matrix elements and print out elements using one single statement, instead of for example two for loops. The following example illustrates some of these features.

http://folk.uio.no/mhjensen/fys3150/2005/blitz/examples/example1.cpp

```
//      Simple test case of matrix operations
//      using Blitz++
#include <blitz/array.h>
#include <iostream>
using namespace std;
using namespace blitz;

int main()
{
  // Create two 4x4 arrays.  We want them to look like matrices, so
  // we'll make the valid index range 1..4 (rather than 0..3 which is
  // the default).

  Range r(1,4);
  Array<float,2> A(r,r), B(r,r);

  // The first will be a Hilbert matrix:
  //
  // a   =   1
  //  ij   -----
  //       i+j-1
  //
  // Blitz++ provides a set of types { firstIndex, secondIndex, ... }
  // which act as placeholders for indices.  These can be used directly
  // in expressions.  For example, we can fill out the A matrix like this:

  firstIndex i;     // Placeholder for the first index
  secondIndex j;    // Placeholder for the second index

  A = 1.0 / (i+j-1);
 cout << "A = " << A << endl;
  // Now the A matrix has each element equal to a_ij = 1/(i+j-1).
  //
  // The matrix B will be the permutation matrix
  //
  // [ 0 0 0 1 ]
  // [ 0 0 1 0 ]
  // [ 0 1 0 0 ]
  // [ 1 0 0 0 ]
  //
  // Here are two ways of filling out B:

  B = (i == (5-j));          // Using an equation -- a bit cryptic

  cout << "B = " << B << endl;

  B = 0, 0, 0, 1,            // Using an initializer list
    0, 0, 1, 0,
    0, 1, 0, 0,
    1, 0, 0, 0;
```

```
  cout << "B = " << B << endl;
}
```

More examples are discussed in appendix A.

4.9 Single-value decomposition

Topic for fall 2008.

4.10 QR decomposition

Topic for fall 2008.

4.11 Physics project, the one-dimensional Poisson equation

The aim of this project is to get familiar with various matrix operations, from dynamic memory allocation to the usage of programs in the library package of the course. For Fortran users memory handling and most matrix and vector operations are included in the ANSI standard of Fortran 90/95. For C++ user however, there are three possible options

1. Make your own functions for dynamic memory allocation of a vector and a matrix. Use then the library package lib.cpp with its header file lib.hpp for obtaining LU-decomposed matrices, solve linear equations etc.

2. Use the library package lib.cpp with its header file lib.hpp which includes a function `matrix` for dynamic memory allocation. This program package includes all the other functions discussed during the lectures for solving systems of linear equations, obatining the determinant, getting the inverse etc.

3. Finally, we provide on the web-page of the course a library package which uses Blitz++'s classes for array handling. You could then, since Blitz++ is installed on all machines at the lab, use these classes for handling arrays.

Your program, whether it is written in C++ or Fortran 90/95, should include dynamic memory handling of matrices and vectors. You should also read the matrix from a file and write your results to a file. Make sure your code includes these options.

(a) Consider the linear system of equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = & \quad w_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = & \quad w_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = & \quad w_3. \end{aligned}$$

This can be written in matrix form as

$$\mathbf{Ax} = \mathbf{w}.$$

Use the included programs for LU decomposition to solve the system of equations

$$\begin{aligned} -x_1 + x_2 - 4x_3 &= 0 \\ 2x_1 + 2x_2 &= 1 \\ 3x_1 + 3x_2 + 2x_3 &= \tfrac{1}{2}. \end{aligned}$$

Use first standard Gaussian elimination and compute the result analytically. Compare thereafter your analytical results with the numerical ones obtained using the LU programs in the program library.

(b) In the rest of this project we will solve the one-dimensional Poisson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations.

The three-dimensional Poisson equation is a partial differential equation,

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2} = -\frac{\rho(x,y,z)}{\epsilon_0},$$

whose solution we will discuss in chapter 15. The function $\rho(x,y,z)$ is the charge density and ϕ is the electrostatic potential. In this project we consider the one-dimensional case since there are a few situations, possessing a high degree of symmetry, where it is possible to find analytic solutions. Let us discuss some of these solutions.

Suppose, first of all, that there is no variation of the various quantities in the y- and z-directions. In this case, Poisson's equation reduces to an ordinary differential equation in x, the solution of which is relatively straightforward. Consider for example a vacuum diode, in which electrons are emitted from a hot cathode and accelerated towards an anode. The anode is held at a large positive potential V_0 with respect to the cathode. We can think of this as an essentially one-dimensional problem. Suppose that the cathode is at $x = 0$ and the anode at $x = d$. Poisson's equation takes the form

$$\frac{d^2\phi}{dx^2} = -\frac{\rho(x)}{\epsilon_0},$$

where $\phi(x)$ satisfies the boundary conditions $\phi(0) = 0$ and $\phi(d) = V_0$. By energy conservation, an electron emitted from rest at the cathode has an x-velocity $v(x)$ which satisfies

$$\frac{1}{2}m_e v^2(x) - e\phi(x) = 0.$$

Furthermore, we assume that the current I is independent of x between the anode and cathode, otherwise, charge will build up at some points. From electromagnetism one can then show that the current I is given by $I = -\rho(x)v(x)A$, where A is the cross-sectional area of the diode. The previous equations can be combined to give

$$\frac{d^2\phi}{dx^2} = \frac{I}{\epsilon_0 A}\left(\frac{m_e}{2e}\right)^{1/2}\phi^{-1/2}.$$

The solution of the above equation which satisfies the boundary conditions is

$$\phi = V_0\left(\frac{x}{d}\right)^{4/3},$$

with

$$I = \frac{4}{9}\frac{\epsilon_0 A}{d^2}\left(\frac{2e}{m_e}\right)^{1/2} V_0^{3/2}.$$

This relationship between the current and the voltage in a vacuum diode is called the Child-Langmuir law.

Another physics example in one dimension is the famous Thomas-Fermi model, widely used as a mean-field model in simulations of quantum mechanical systems [35, 36], see Lieb for a newer and updated discussion [37]. Thomas and Fermi assumed the existence of an energy functional, and derived an expression for the kinetic energy based on the density of electrons, $\rho(r)$ in an infinite potential well. For a large atom or molecule with a large number of electrons. Schrödinger's equation, which would give the exact density and energy, cannot be easily handled for large numbers of interacting particles. Since the Poisson equation connects the electrostatic potential with the charge density, one can derive the following equation for potential V

$$\frac{d^2V}{dx^2} = \frac{V^{3/2}}{\sqrt{x}},$$

with $V(0) = 1$.

In our case we will rewrite Poisson's equation in terms of dimensional variables. We can then rewrite the equation as

$$-u''(x) = f(x), \quad x \in (0,1), \quad u(0) = u(1) = 0.$$

and we define the discretized approximation to u as v_i with grid points $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$. The step length or spacing is defined as $h = 1/(n+1)$. We have then the boundary conditions $v_0 = v_{n+1} = 0$. We approximate the second derivative of u with

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{for } i = 1, \ldots, n,$$

where $f_i = f(x_i)$. Show that you can rewrite this equation as a linear set of equations of the form

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}},$$

where $\mathbf{A}$ is an $n \times n$ tridiagonal matrix which we rewrite as

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \ldots & \ldots & 0 \\ -1 & 2 & -1 & 0 & \ldots & \ldots \\ 0 & -1 & 2 & -1 & 0 & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & & -1 & 2 & -1 \\ 0 & \ldots & & 0 & -1 & 2 \end{pmatrix} \tag{4.35}$$

and $\tilde{b}_i = h^2 f_i$.

In our case we will assume that $f(x) = (3x + x^2)e^x$, and keep the same interval and boundary conditions. Then the above differential equation has an analytic solution given by $u(x) = x(1 - x)e^x$ (convince yourself that this is correct by inserting the solution in the Poisson equation). We will compare our numerical solution with this analytic result in the next exercise.

(c) We can rewrite our matrix $\mathbf{A}$ in terms of one-dimensional vectors a, b, c of length $1 : n$. Our linear equation reads

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{pmatrix}. \tag{4.36}$$

A tridiagonal matrix is a special form of banded matrix where all the elements are zero except for those on and immediately above and below the leading diagonal. The above tridiagonal system can be written as

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i, \tag{4.37}$$

for $i = 1, 2, \dots, n$. The algorithm for solving this set of equations is rather simple and requires two steps only, a decomposition and forward substitution and finally a backward substitution.

Your first task is to set up the algorithm for solving this set of linear equations. Find also the number of operations needed to solve the above equations. Show that they behave like $O(n)$ with n the dimensionality of the problem. Compare this with standard Gaussian elimination.

Then you should code the above algorithm and solve the problem for matrices of the size 10×10, 100×100 and 1000×1000. That means that you choose $n = 10$, $n = 100$ and $n = 1000$ grid points.

Compare your results (make plots) with the analytic results for the different number of grid points in the interval $x \in (0, 1)$. The different number of grid points corresponds to different step lengths h.

Compute also the maximal relative error in the data set $i = 1, \dots, n$,by setting up

$$\epsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right),$$

as function of $log_{10}(h)$ for the function values u_i and v_i. For each step length extract the max value of the relative error. Try to increase n to $n = 10000$ and $n = 10^5$. Comment your results.

(d) Compare your results with those from the LU decomposition codes for the matrix of size 1000×1000. Use for example the unix function *time* when you run your codes and compare the time usage between LU decomposition and your tridiagonal solver. Can you run the standard LU decomposition for a matrix of the size $10^5 \times 10^5$? Comment your results.

4.11.1 Solution to exercise c)

The program listed below encodes a possible solution to part c) of the above project. Note that we have employed Blitz++ as library and that the range of the various vectors are now shifted from their default ranges $(0 : n - 1)$ to $(1 : n)$ and that we access vector elements as $a(i)$ instead of the standard C++ declaration $a[i]$.

The program reads from screen the name of the ouput file and the dimension of the problem, which in our case corresponds to the number of mesh points as well, in addition to the two endpoints. The function $f(x) = (3x + x^2)\exp(x)$ is included explicitely in the code. An obvious change is to define a separate function, allowing thereby for a generalization to other function $f(x)$.

```
/*
    Program to solve the one-dimensional Poisson equation
    -u''(x) = f(x)  rewritten as a set of linear equations
    A u = f   where A is an n x n matrix, and u and f are 1 x n vectors
    In this problem f(x) = (3x+x*x)exp(x)  with solution u(x) = x(1-x)exp(x)
    The program reads  from screen the name of the output file.
    Blitz++ is used here, with arrays starting from 1 to n
*/
#include <iomanip>
#include <fstream>
#include <blitz/array.h>
#include <iostream>
using namespace std;
using namespace blitz;

ofstream ofile;
//  Main program only, no other functions
int main(int argc, char* argv[])
{
  char *outfilename;
  int i, j, n;
  double h, btemp;
  // Read in output file, abort if there are too few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  cout << "Read in number of mesh points" << endl;
  cin >> n;
  h =  1.0/( (double) n+1);
  //  Use Blitz to allocate arrays
  //  Use range to change default arrays from 0:n-1 to 1:n
  Range r(1,n);
  Array<double,1> a(r), b(r), c(r), y(r), f(r), temp(r);
  //  set up the matrix defined by three arrays, diagonal, upper and lower
     diagonal band
  b = 2.0;  a = -1.0  ; c = -1.0;
  // Then define the value of the right hand side f (multiplied by h*h)
  for(i=1; i <= n; i++){
    // Explicit expression for f, could code as separate function
    f(i) = h*h*(i*h*3.0+(i*h)*(i*h))*exp(i*h);
  }
  // solve the tridiagonal system, first forward substitution
  btemp = b(1);
  for(i = 2; i <= n; i++) {
    temp(i) = c(i-1) / btemp;
    btemp = b(i) - a(i) * temp(i);
```

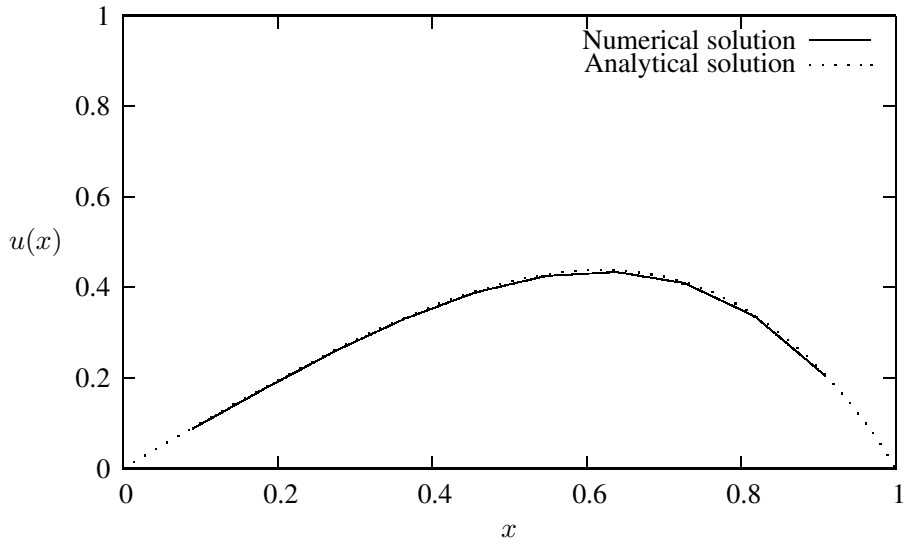

Figure 4.4: Numerical solution obtained with $n = 10$ compared with the analytical solution.

```
    y(i) = (f(i) - a(i) * y(i-1)) / btemp;
  }
  // then backward substitution, the solution is in y()
  for(i = n-1; i >= 1; i--) {
    y(i) -= temp(i+1) * y(i+1);
  }
  // write results to the output file
  for(i = 1; i <= n; i++){
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << setw(15) << setprecision(8) << i*h;
    ofile << setw(15) << setprecision(8) << y(i);
    ofile << setw(15) << setprecision(8) << i*h*(1.0-i*h)*exp(i*h) <<endl;
  }
  ofile.close();
}
```

The program writes also the exact solution to file. In Fig. 4.4 we show the results obtained with $n = 10$. Even with so few points, the numerical solution is very close to the analytic answer. With $n = 100$ it is almost impossible to distinguish the numerical solution from the analytical one, as shown in Fig. 4.5. It is therefore instructive to study the relative error, which we display in Table 4.4 as function of the step length $h = 1/(n + 1)$.

The mathematical truncation we made when computing the second derivative goes like $O(h^2)$. Our results for n from $n = 10$ to somewhere between $n = 10^4$ and $n = 10^5$ result in a slope which is almost exactly equal 2,in good agreement with the mathematical truncation made. Beyond $n = 10^5$ the relative error becomes bigger, telling us that there is no point in increasing n. For most practical application a relative error between 10^{-6} and 10^{-8} is more than sufficient, meaning that $n = 10^4$ may be an acceptable number of mesh points. Beyond $n = 10^5$, numerical round off errors take over, as discussed in the previous chapter as well.

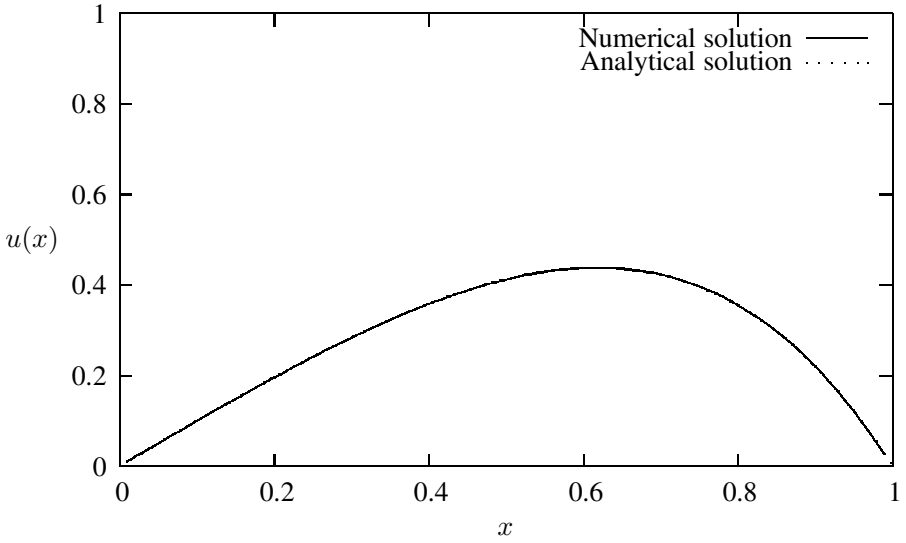

Figure 4.5: Numerical solution obtained with $n = 10$ compared with the analytical solution.

Table 4.4: log_{10} values for the relative error and the step length h computed at $x = 0.5$.

n	$log_{10}(h)$	$\epsilon_i = log_{10}\left(\left\|(v_i - u_i)/u_i\right\|\right)$
10	-1.04	-2.29
100	-2.00	-4.19
1000	-3.00	-6.18
10^4	-4.00	-8.18
10^5	-5.00	-9.19
10^6	-6.00	-6.08