Chapter 6
Linear Algebra

Abstract This chapter introduces several matrix related topics, from the solution of linear equations, computing determinants, conjugate-gradient methods, spline interpolation to efficient handling of matrices.

6.1 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 also be discussed.

This chapter serves also the purpose of introducing important programming details such as handling memory allocation for matrices and the usage of the libraries which follow these lectures.

The algorithms we describe and their original source codes are taken from the widely used software package LAPACK [26], 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 [27], 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 [28]. Good recent introductory texts are Kincaid and Cheney [23] and Datta [29]. For more advanced ones see Trefethen and Bau III [30], Kress [24] and Demmel [31]. Ref. [28] contains an extensive list of textbooks on eigenvalue problems and linear algebra. LAPACK [26] contains also extensive listings to the research literature on matrix computations. For the introduction of the auxiliary library Blitz++ [32], which allows for a very efficient way of handling arrays in C++ we refer to the online manual at http://www.onumerics.org. A library we highly recommend is Armadillo, see http://arma.sourceforge.org. Armadillo is an open-source C++ linear algebra library aiming towards a good balance between speed and ease of use. Integer, floating point and complex numbers are supported, as well as a subset of trigonometric and statistics functions.
Various matrix and vector operations are provided through optional integration with BLAS and LAPACK.

6.2 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\(^1\) 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 \( \mathbf{I} \)

\[
\mathbf{A} = \begin{pmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{pmatrix},
\]

\[
\mathbf{I} = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}, \tag{6.1}
\]

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

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

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 6.2 lists some essential features of various types of matrices one may encounter. Some of the matrices we will encounter are listed here

<table>
<thead>
<tr>
<th>Relations</th>
<th>Name</th>
<th>matrix elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbf{A} = \mathbf{A}' )</td>
<td>symmetric</td>
<td>( a_{ij} = a_{ji} )</td>
</tr>
<tr>
<td>( \mathbf{A} = (\mathbf{A}')^{-1} )</td>
<td>real orthogonal</td>
<td>( \sum_i a_{ik}a_{jk} = \sum_i a_{ki}a_{kj} = \delta_{ij} )</td>
</tr>
<tr>
<td>( \mathbf{A} = \mathbf{A}' )</td>
<td>real matrix</td>
<td>( a_{ij} = a_{ij}' )</td>
</tr>
<tr>
<td>( \mathbf{A} = \mathbf{A}' )</td>
<td>hermitian</td>
<td>( a_{ij} = a_{ji}' )</td>
</tr>
<tr>
<td>( \mathbf{A} = (\mathbf{A}')^{-1} )</td>
<td>unitary</td>
<td>( \sum_i a_{ik}a_{ij}' = \sum_i a_{kj}'a_{kj} = \delta_{ij} )</td>
</tr>
</tbody>
</table>

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

\(^1\) 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 \( \mathbb{C}[a,b] \) is the space of real or complex-valued continuous functions on the interval \([a,b]\), where the latter is a closed interval. Similarly, \( \mathbb{C}^m[a,b] \) is the space of \( m \)-times continuously differentiable functions on the interval \([a,b]\). For more symbols and notations, see the main text.
2. Upper triangular if \( a_{ij} = 0 \) for \( i > j \), which for a \( 4 \times 4 \) matrix is of the form
\[
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
0 & a_{22} & a_{23} & a_{24} \\
0 & 0 & a_{33} & a_{34} \\
0 & 0 & 0 & a_{44}
\end{pmatrix}
\]

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 \( A \) the following properties are all equivalent
1. If the inverse of \( A \) exists, \( A \) is nonsingular.
2. The equation \( Ax = 0 \) implies \( x = 0 \).
3. The rows of \( A \) form a basis of \( \mathbb{R}^n \).
4. The columns of \( A \) form a basis of \( \mathbb{R}^n \).
5. \( A \) is a product of elementary matrices.
6. 0 is not an eigenvalue of \( A \).

The basic matrix operations that we will deal with are addition and subtraction
\[
A = B \pm C \implies a_{ij} = b_{ij} \pm c_{ij},
\] (6.2)
scalar-matrix multiplication
\[
A = \gamma B \implies a_{ij} = \gamma b_{ij},
\]
vector-matrix multiplication
\[ y = Ax \implies y_i = \sum_{j=1}^{n} a_{ij} x_j, \quad (6.3) \]

matrix-matrix multiplication

\[ A = BC \implies a_{ij} = \sum_{k=1}^{n} b_{ik} c_{kj}, \quad (6.4) \]

transposition

\[ A = B^T \implies a_{ij} = b_{ji}, \]

and if \( A \in \mathbb{C}^{n \times n} \), conjugation results in

\[ A = B^T \implies a_{ij} = b_{ji}, \]

where a variable \( z = x - iy \) denotes the complex conjugate of \( z = x + iy \). In a similar way we have the following basic vector operations, namely addition and subtraction

\[ x = y \pm z \implies x_i = y_i \pm z_i, \]

scalar-vector multiplication

\[ x = \gamma y \implies x_i = \gamma y_i, \]

vector-vector multiplication (called Hadamard multiplication)

\[ x = yz \implies x_i = y_i z_i, \]

the inner or so-called dot product

\[ c = y^T z \implies c = \sum_{j=1}^{n} y_j z_j, \quad (6.5) \]

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

\[ A = yz^T \implies a_{ij} = y_i z_j, \quad (6.6) \]

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

\[ ||x||_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{\frac{1}{p}}, \]

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

\[ ||x||_1 = |x_1| + |x_2| + \cdots + |x_n|, \]

\[ ||x||_2 = (|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2)^{\frac{1}{2}} = (x^T x)^{\frac{1}{2}}, \]

and

\[ ||x||_\infty = \max |x_i|, \]

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. For any \( x \) and \( y \) being real-valued or complex-valued quantities, the inner product space satisfies

\[ |x^T y| \leq ||x||_2 ||y||_2, \]

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

\[ ||x+y|| \leq ||x|| + ||y||. \]
6.3 Programming Details

\[ |x + y|_2 \leq |x|_2 + |y|_2. \]

Proofs can be found in for example Ref. [28]. 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 \( x \). We assume that \( \text{fl}(x) \in \mathbb{R}^n \) is the machine representation of a vector \( x \in \mathbb{R}^n \). If \( x \neq 0 \), we define the relative error as

\[ \varepsilon = \frac{||\text{fl}(x) - x||}{||x||}. \]

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

\[ \frac{||\text{fl}(x) - x||_\infty}{||x||_\infty} \approx 10^{-l}, \]

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

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

\[ ||A||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}, \]

and the \( p \)-norms

\[ ||A||_p = \frac{||Ax||_p}{||x||_p}, \]

assuming that \( x \neq 0 \). We refer the reader to the text of Golub and Van Loan [28] 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.

6.3 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 dynamic memory allocation. Useful references on C++ programming details, in particular on the use of pointers and memory allocation, are Reek’s text [33] on pointers in C, Berryhill’s monograph [34] on scientific programming in C++ and finally Franek’s text [35] 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 [18] and Barton and Nackman [19]. See also the online lecture notes on C++ at http://heim.ifi.uio.no/~hpl/INF-VERK4830. For Fortran we recommend the online lectures at http://folk.uio.no/gunnarw/INF-VERK4820. These web pages contain extensive references to other C++ and Fortran resources. Both web pages contain enough material, lecture notes and exercises, in order to serve as material for own studies.
6.3.1 Declaration of fixed-sized vectors and matrices

In the program below we discuss some 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 subfunction(). 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.

```c
int main()
{
    int k, m, row = 3, col = 5;
    int vec[5];      // line a
    int matr[3][5];  // line b
    // Fill in vector vec
    for (k = 0; k < col; k++) vec[k] = k;
    // fill in matr
    for (m = 0; m < row; m++)
        for (k = 0; k < col; k++) matr[m][k] = m + 10 * k;
    // write out the vector
    cout << "Content of vector vec:
    for (k = 0; k < col; k++)
        cout << vec[k] << endl;
    // Then write out the matrix
    cout << "Content of matrix matr:
    for (m = 0; m < row; m++)
        for (k = 0; k < col; k++)
            cout << matr[m][k] << endl;
    subfunction(row, col, vec, matr);  // line c
    return 0;
}  // end main function

void subfunction(int row, int col, int vec[], int matr[][5]);  // line d
{
    int k, m;
    // write out the vector
    cout << "Content of vector vec in subfunction:
    for (k = 0; k < col; k++)
        cout << vec[k] << endl;
    // Then write out the matrix
    cout << "Content of matrix matr in subfunction:
    for (m = 0; m < row; m++)
        for (k = 0; k < col; k++)
            cout << matr[m][k] << endl;
}  // end of function subfunction
```

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 re-
quired 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 subfunction() are released when the execution returns to main().

6.3.2 Runtime Declarations of Vectors and Matrices in C++

We change thereafter our program in order to include dynamic allocation of arrays. 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 and C++ it is possible and common to postpone size declarations of arrays until you really know what you need and also release memory reservations when it is not needed any more. The following program shows how we could change the previous one with static declarations to dynamic allocation of arrays.

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

    cout << "Read in number of rows:\n" << endl; // line c
    cin >> row;
    cout << "Read in number of columns:\n" << endl;
    cin >> col;

    vec = new int[col]; // line d
    matr = (int **)(matrix(row, col, sizeof(int))); // line e
    // Fill in vector vec
    for (k = 0; k < col; k++) vec[k] = k;
    // fill in matr
    for (m = 0; m < row; m++){
        for (k = 0; k < col ; k++) matr[m][k] = m + 10*k;
    }
    // write out the vector
    cout << "Content of vector vec:\n" << endl;
    for (k = 0; k < col; k++)
    {
        cout << vec[k] << endl;
    }
    // Then write out the matrix
    cout << "Content of matrix matr:\n" << endl;
    for (m = 0; m < row; m++){
        for (k = 0; k < col ; k++)
        {
            cout << matr[m][k] << endl;
        }
    }
    subfunction(row, col, vec, matr); // line f
    free_matrix((void **)matr); // line g
    delete vec[];
    return 0;
} // end main function

void subfunction(int row, int col, int vec[], int matr[][5]); // line h
{
    int k, m;
    // write out the vector
```
cout << `Content of vector vec in subfunction:' << endl;
for (k = 0; k < col; k++){
    cout << vec[k] << endl;
}
// Then write out the matrix
cout << `Content of matrix matr in subfunction:' << endl;
for (m = 0; m < row; m++){
    for (k = 0; k < col ; k++){
        cout << matr[m][k] << endl;
    }
}
} // end of function subfunction

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. Similarly, 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 with dimensionality [col][col]

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 the program below 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.

```c++
/*
 * 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;
    }
```
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

```c++
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. 6.2.

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

```c++
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.onumerics.org or Armadillo makes life much easier when dealing with matrices.

### 6.3.3 Matrix Operations and C++ and Fortran Features of Matrix handling

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

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 $a_{11}a_{12}\ldots a_{1n}a_{21}a_{22}\ldots a_{2n}\ldots a_{nn}$ 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 $a_{11}a_{21}\ldots a_{1n}a_{22}\ldots a_{2n}\ldots a_{nn}$.

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. We have $n \times n$ matrices $A$, $B$ and $C$ and we wish to evaluate $A = B + C$ according to Eq. (6.2). In C++ this would be coded like

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

**Fig. 6.2** Conceptual representation of the allocation of a matrix in C++.

|--------|---------|---------|---------|---------|
while in Fortran we would have

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

Fig. 6.3 shows how a $3 \times 3$ matrix $A$ is stored in both row-major and column-major ways.

![Matrix Storage Diagram](image)

**Fig. 6.3** 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 we 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. (6.4) of two matrices $A = BC$ could then take the following form in C++

```
for(i=0 ; i < n ; i++) {
  for(j=0 ; j < n ; j++) {

```
6.3 Programming Details

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

and in Fortran we have

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

However, Fortran has an intrinsic function called MATMUL, and the above three loops can be coded in a single statement `a=MATMUL(b,c)`. Fortran 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. The coding of Eq. (6.6) takes then the following form in C++

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

and in Fortran we have

```fortran
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

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

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 $1000 \times 1000$ case should take two seconds. This can be reduced if we multiply two matrices which are upper triangular such as

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

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

```cpp
for(i=0 ; i < n ; i++) {
    for(j=0 ; j < n ; j++) {
        for(k=i ; k < j ; k++) {
            
```
and need seven multiplications,

\begin{verbatim}
   a[i][j] += b[i][k] * c[k][j]
\end{verbatim}

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 \( \sum_{j=1}^{n} j = \frac{n(n+1)}{2} \) for large \( n \) values. Using in addition that \( \sum_{j=1}^{n} j^2 \approx \frac{n^3}{3} \) 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 \( A \) 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 one uses the intrinsic functions \texttt{ALLOCATE} and \texttt{DEALLOCATE}, while C++ employs the functions \texttt{new} and \texttt{delete}.

### 6.3.3.1 Strassen’s algorithm

As we have seen, the straightforward algorithm for matrix-matrix multiplication will require \( p \) multiplications and \( p - 1 \) additions for each of the \( m \times n \) elements. The total number of floating-point operations is then \( mn(2p - 1) \sim \mathcal{O}(mn^2) \). When the matrices \( A \) and \( B \) can be divided into four equally sized blocks,

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
= \begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix},
\]

we get eight multiplications of smaller blocks,

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
= \begin{bmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{bmatrix}.
\]

Strassen discovered in 1968 how the number of multiplications could be reduced from eight to seven [28]. Following Strassen’s approach we define some intermediates,

\[
\begin{align*}
S_1 &= A_{21} + A_{22}, & T_1 &= B_{12} - B_{11}, \\
S_2 &= S_1 - A_{11}, & T_2 &= B_{22} - T_1, \\
S_3 &= A_{11} - A_{21}, & T_3 &= B_{22} - B_{12}, \\
S_4 &= A_{12} - S_2, & T_4 &= B_{21} - T_2,
\end{align*}
\]

and need seven multiplications,
\[ P_1 = A_{11}B_{11}, \quad U_1 = P_1 + P_2, \]
\[ P_2 = A_{12}B_{21}, \quad U_2 = P_1 + P_4, \]
\[ P_3 = S_1T_1, \quad U_3 = U_2 + P_3, \]
\[ P_4 = S_2T_2, \quad U_4 = U_3 + P_7, \]
\[ P_5 = S_3T_3, \quad U_5 = U_3 + P_3, \]
\[ P_6 = S_4B_{22}, \quad U_6 = U_2 + P_3, \]
\[ P_7 = A_{22}T_4, \quad U_7 = U_6 + P_6, \]

(6.10)

to find the resulting \( C \) matrix as
\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix} =
\begin{bmatrix}
U_1 & U_7 \\
U_4 & U_5
\end{bmatrix}.
\]

(6.11)

In spite of the seemingly additional work, we have reduced the number of multiplications from eight to seven. Since the multiplications are the computational bottleneck compared to addition and subtraction, the number of flops are reduced.

In the case of square \( n \times n \) matrices with \( n \) equal to a power of two, \( n = 2^m \), the divided blocks will have \( \frac{n}{2} = 2^{m-1} \). Letting \( f(m) \) be the number of flops needed for the full matrix and applying Strassen recursively we find the total number of flops to be
\[
f(m) = 7f(m-1) = 7^2f(m-2) = \cdots = 7^mf(0),
\]

(6.12)

where \( f(0) \) is the one floating-point operation needed for multiplication of two numbers (two \( 2^0 \times 2^0 \) matrices). For large matrices this can prove efficient, yielding a much better scaling,
\[
O(7^m) = O(2^{m \log_2 7}) = O(n^{\log_2 7}) \approx O(n^{2.807}),
\]

(6.13)

effectively saving \( 7/8 = 12.5\% \) each time it is applied.

### 6.3.3.2 Fortran 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 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, consider the following simple example where we set up a $4 \times 4$ unity matrix.

```fortran
      IMPLICIT NONE
! The definition of the matrix, using dynamic allocation
      REAL, ALLOCATABLE, DIMENSION(:,:,1) :: 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.

### 6.4 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 [24,28] and will allow us to catch several birds with a stone. We will show how to rewrite a matrix $A$ in terms of an upper and a lower triangular matrix, from which we easily can solve linear equation, compute the inverse of $A$ 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

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

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 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 \( x_i = a + ih \), 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

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

which we rewrite as

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

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

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

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

\[
A = \frac{1}{h^2} \begin{pmatrix}
2 & -1 & \cdots & \cdots & -1 \\
-1 & 2 & -1 & \cdots & \cdots \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
-1 & 2 & -1 & \cdots & \cdots \\
-1 & 2 & -1 & \cdots & \cdots
\end{pmatrix}
\]

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

\[
Au = f(u).
\]  

(6.14)

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 \( h \to 0 \), 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

\[
Au = f,
\]  

(6.15)

which is nothing but a simple linear equation with a tridiagonal matrix \( A \). We will solve such a system of equations in subsubsection 6.4.3.

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 \( m = \hbar = \omega = 1 \) ) \( f(x_i, u(x_i)) = -x_i^2u(x_i) + 2\lambda u(x_i) \) with \( \lambda \) being the eigenvalue. Inserting this into our equation, we define first a new matrix \( A \) as
we obtain a so-called upper triangular set of equations of the form

\[ A = \begin{pmatrix}
\frac{2}{h^2} + x_1^2 & -\frac{1}{h^2} & \cdots & 0 \\
-\frac{1}{h^2} & \frac{2}{h^2} + x_2^2 & -\frac{1}{h^2} & \cdots \\
\vdots & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & \frac{2}{h^2} + x_n^2
\end{pmatrix}, \quad (6.16)

which leads to the following eigenvalue problem

\[ \begin{pmatrix}
\frac{2}{h^2} + x_1^2 & -\frac{1}{h^2} & \cdots & 0 \\
-\frac{1}{h^2} & \frac{2}{h^2} + x_2^2 & -\frac{1}{h^2} & \cdots \\
\vdots & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & \frac{2}{h^2} + x_n^2
\end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = 2\lambda \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}. \]

We will solve this type of equations in chapter 7. These lecture notes contain however several other examples of rewriting mathematical expressions into matrix problems. In chapter 5 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.

### 6.4.1 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

\[ Ax = w. \]

We assume also that the matrix \( A \) 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 \( A \in \mathbb{R}^{4 \times 4} \), 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

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

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
6.4 Linear Systems

\[ b_{11} x_1 + b_{12} x_2 + b_{13} x_3 + b_{14} x_4 = y_1 \\
\]
\[ b_{22} x_2 + b_{23} x_3 + b_{24} x_4 = y_2 \\
\]
\[ b_{33} x_3 + b_{34} x_4 = y_3 \\
\]
\[ b_{44} x_4 = y_4. \]

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

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

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 \( a_{1j}/a_{11} \) and then subtract the result from the \( j \)th equation. We assume obviously that \( a_{11} \neq 0 \) and that \( A \) is not singular. We will come back to this problem below.

Our actual 4 \( \times \) 4 example reads after the first operation

\[
\begin{pmatrix}
  a_{11} & a_{12} & a_{13} & a_{14} \\
  0 & (a_{22} - a_{21}a_{11}) & (a_{23} - a_{21}a_{13}) & (a_{24} - a_{21}a_{14}) \\
  0 & (a_{32} - a_{31}a_{11}) & (a_{33} - a_{31}a_{13}) & (a_{34} - a_{31}a_{14}) \\
  0 & (a_{42} - a_{41}a_{11}) & (a_{43} - a_{41}a_{13}) & (a_{44} - a_{41}a_{14})
\end{pmatrix}
\begin{pmatrix}
  x_1 \\
  x_2 \\
  x_3 \\
  x_4
\end{pmatrix} =
\begin{pmatrix}
  y_1 \\
  y_2 \\
  y_3 \\
  y_4
\end{pmatrix}
\]

or

\[
\begin{align*}
  b_{11} x_1 + b_{12} x_2 + b_{13} x_3 + b_{14} x_4 &= y_1 \\
  a_{22} x_2 + a_{21} x_1 + a_{23} x_3 + a_{24} x_4 &= w_2 \\
  a_{32} x_2 + a_{31} x_1 + a_{33} x_3 + a_{34} x_4 &= w_3 \\
  a_{42} x_2 + a_{41} x_1 + a_{43} x_3 + a_{44} x_4 &= w_4,
\end{align*}
\]

(6.17)

with the new coefficients

\[ b_{1k} = a_{1k}^{(1)} \quad k = 1, \ldots, n, \]

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

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

with a new right-hand side given by

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

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

\[
a_{jk}^{(m+1)} = a_{jk}^{(m)} - \frac{a_{jm}^{(m)} a_{mk}^{(m)}}{a_{mm}^{(m)}}, \quad j, k = m + 1, \ldots, n,
\]
with \( m = 1, \ldots, n - 1 \) and a right-hand side given by
\[
W_j^{(m+1)} = W_j^{(m)} - \frac{a_{jm}^{(m)} W_m^{(m)}}{a_{mm}^{(m)}} \quad j = m + 1, \ldots, n.
\]

This set of \( n - 1 \) eliminations leads us to Eq. (6.17), which is solved by back substitution. If the arithmetics is exact and the matrix \( A \) is not singular, then the computed answer will be exact. However, as discussed in the two preceeding chapters, computer arithmetics is not 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 large numbers. Suppose for example that our first division in \((a_{22} - a_{23}a_{12}/a_{11})\) results in \(-10^7\), that is \(a_{23}a_{12}/a_{11}\). Assume also that \(a_{22}\) is 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,
6.4 Linear Systems

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

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 [28], in particular chapter three. See also the discussion of chapter two in Ref. [36]. 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 \( n^3/3 + r n^2 - n/3 \) multiplications. Adding the cost of additions, we end up with \( 2n^3/3 + O(n^2) \) floating point operations, see Kress [24] for a proof. An \( n \times n \) matrix of dimensionality \( n = 10^3 \) 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 elimination algorithm allows you to compute the determinant of \( A \) via the product of the diagonal matrix elements 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, 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 note that to obtain the LU decomposed matrix requires roughly \( O(n^3) \) floating point operations. Finally, LU decomposition allows for an efficient computation of the inverse of \( A \).

6.4.2 LU Decomposition of a Matrix

A frequently used form of Gaussian elimination is L(ower)U(pper) factorization 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 \( L \) with elements only below the diagonal (and thereby the naming lower) and a matrix \( U \) which contains both the diagonal and matrix elements above the diagonal (leading to the labelling upper). Consider again the matrix \( A \) given in Eq. (6.1). The LU decomposition method means that we can rewrite this matrix as the product of two matrices \( L \) and \( U \) where

\[
A = LU = \begin{pmatrix}
    a_{11} & a_{12} & a_{13} & a_{14} \\
    a_{21} & a_{22} & a_{23} & a_{24} \\
    a_{31} & a_{32} & a_{33} & a_{34} \\
    a_{41} & a_{42} & a_{43} & a_{44}
\end{pmatrix}
\begin{pmatrix}
    l_{11} & 0 & 0 & 0 \\
    l_{21} & l_{22} & 0 & 0 \\
    l_{31} & l_{32} & l_{33} & 0 \\
    l_{41} & l_{42} & l_{43} & l_{44}
\end{pmatrix}
\begin{pmatrix}
    u_{11} & u_{12} & u_{13} & u_{14} \\
    0 & u_{22} & u_{23} & u_{24} \\
    0 & 0 & u_{33} & u_{34} \\
    0 & 0 & 0 & u_{44}
\end{pmatrix}
\] .

(6.18)
LU decomposition forms the backbone of other algorithms in linear algebra, such as the solution of linear equations given by

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

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

\[
\det \{ A \} = u_{11} u_{22} \ldots u_{nn}.
\]

For a proof of this statement, see chapter 3.2 of Ref. [28].

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

\[
\begin{align*}
    a_{11} &= u_{11} \\
    a_{21} &= l_{21} u_{11} \\
    a_{31} &= l_{31} u_{11} \\
    a_{41} &= l_{41} u_{11},
\end{align*}
\]

which determine the elements \( u_{11}, l_{21}, l_{31} \) and \( l_{41} \) in \( L \) and \( U \). Writing out the equations for the second column we get

\[
\begin{align*}
    a_{12} &= u_{12} \\
    a_{22} &= l_{21} u_{12} + u_{22} \\
    a_{32} &= l_{31} u_{12} + l_{32} u_{22} \\
    a_{42} &= l_{41} u_{12} + l_{42} u_{22}.
\end{align*}
\]

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

We can generalize this procedure into three equations

\[
\begin{align*}
    i < j : & \quad l_{i1} u_{1j} + l_{i2} u_{2j} + \cdots + l_{ij} u_{ij} = a_{ij} \\
    i = j : & \quad l_{i1} u_{1j} + l_{i2} u_{2j} + \cdots + l_{ij} u_{jj} = a_{ij} \\
    i > j : & \quad l_{i1} u_{1j} + l_{i2} u_{2j} + \cdots + l_{ij} u_{jj} = a_{ij}
\end{align*}
\]

which gives the following algorithm:

Calculate the elements in \( L \) and \( U \) columnwise starting with column one. For each column \((j)\):

- Compute the first element \( u_{1j} \) by
  \[
  u_{1j} = a_{1j}.
  \]
- Next, we calculate all elements \( u_{ij}, i = 2, \ldots, j - 1 \)
6.4 Linear Systems

\[ u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}. \]

- Then calculate the diagonal element \( u_{jj} \)

\[ u_{jj} = a_{jj} - \sum_{k=1}^{j-1} l_{jk} u_{kj}. \] (6.19)

- Finally, calculate the elements \( l_{ij}, i > j \)

\[ l_{ij} = \frac{1}{u_{jj}} \left( a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right), \] (6.20)

The algorithm is known as Doolittle’s algorithm since the diagonal matrix elements of \( L \) are 1. For the case where the diagonal elements of \( U \) are 1, we have what is called Crout’s algorithm. For the case where \( U = L^T \) so that \( u_{ii} = l_{ii} \) for \( 1 \leq i \leq n \) we can use what is called the Cholesky factorization algorithm. In this case the matrix \( A \) has to fulfill several features; namely, it should be real, symmetric and positive definite. A matrix is positive definite if the quadratic form \( x^T A x > 0 \). Establishing this feature is not easy since it implies the use of an arbitrary vector \( x \neq 0 \). 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 \( u_{jj} \) is close to or equals zero, a case which can lead to serious problems. Consider the following simple \( 2 \times 2 \) example taken from Ref. [30]

\[ A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}. \]

The algorithm discussed above fails immediately, the first step simple states that \( u_{11} = 0 \). We could change slightly the above matrix by replacing 0 with \( 10^{-20} \) resulting in

\[ A = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 1 \end{pmatrix}, \]

yielding

\[ u_{11} = 10^{-20} \]
\[ l_{21} = 10^{20} \]

and \( u_{12} = 1 \) and

\[ u_{22} = a_{11} - l_{21} = 1 - 10^{20}, \]

we obtain

\[ L = \begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix}, \]

and

\[ U = \begin{pmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{pmatrix}, \]

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 \( U \) has an element \( u_{22} = 1 - 10^{20} \). Numerically, since we do have a limited precision, which for double precision is approximately \( \varepsilon_M \sim 10^{-16} \) it means that this number is approximated in the machine as \( u_{22} \sim -10^{20} \) resulting in a machine representation of the matrix as

\[ U = \begin{pmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{pmatrix}. \]
If we multiply the matrices \( LU \) we have
\[
\begin{pmatrix}
1 & 0 \\
10^{-20} & 1
\end{pmatrix}
\begin{pmatrix}
10^{-20} & 1 \\
0 & -10^{20}
\end{pmatrix}
= \begin{pmatrix}
10^{-20} & 1 \\
1 & 0
\end{pmatrix}
\neq A.
\]

We do not get back the original matrix \( A \)!

The solution is pivoting (interchanging rows in this case) around the largest element in a column \( j \). Then we are actually decomposing a rowwise permutation of the original matrix \( A \). The key point to notice is that Eqs. (6.19) and (6.20) are equal except for the case that we divide by \( u_{jj} \) 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 \( u_{jj} \) as the one which happens to fall along the diagonal in the first instance. Rather, we could promote one of the undivided \( l_{ij} \)'s in the column \( i = j + 1, \ldots N \) to become the diagonal of \( U \). The partial pivoting in Crout’s or Doolittle’s methods means then that we choose the largest value for \( u_{jj} \) (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 \( A \) it would have sufficed to interchange the two rows and start the LU decomposition with
\[
A = \begin{pmatrix}
1 & 1 \\
0 & 1
\end{pmatrix}.
\]

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. [28],
\[
LU = A + H,
\]
with
\[
|H| \leq 3(n - 1)u(|A| + |L||U|) + O(u^2),
\]
with \( |H| \) being the absolute value of a matrix and \( u \) is the error done in representing the matrix elements of the matrix \( A \) as floating points in a machine with a given precision \( \varepsilon_M \), viz. every matrix element of \( u \) is
\[
|fl(a_{ij}) - a_{ij}| \leq u_{ij},
\]
with \( |u_{ij}| \leq \varepsilon_M \) resulting in
\[
|fl(A) - A| \leq u|A|.
\]

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

| C++: | ludcmp(double **a, int n, int *indx, double *d) |
| Fortran: | 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 variable \( d \), which is \( \pm 1 \) depending on whether we have an even or odd number of row interchanges, a pointer \( indx \) that records the row permutation which has been effected and the LU decomposed matrix. Note that the original matrix is destroyed.

### 6.4.2.1 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)
\[ L_{ii} = \left( A_{ii} - \sum_{k=0}^{i-1} L_{ik}^2 \right)^{1/2}. \]

- 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 [28] 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 $LDL^T$ factorization with positive diagonal elements only in the diagonal matrix $D$. 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 $Ax = \lambda x$ and check that all eigenvalues are positive.

### 6.4.3 Solution of Linear Systems of Equations

With the LU decomposition it is rather simple to solve a system of linear equations
\[
\begin{align*}
    a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= w_1 \\
    a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= w_2 \\
    a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= w_3 \\
    a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= w_4.
\end{align*}
\]

This can be written in matrix form as
\[ Ax = w. \]

where $A$ and $w$ are known and we have to solve for $x$. Using the LU decomposition we write
\[ Ax = LUx = w. \]  \hspace{1cm} (6.21)

This equation can be calculated in two steps
\[ Ly = w; \quad Ux = y. \]  \hspace{1cm} (6.22)

To show that this is correct we use to the LU decomposition to rewrite our system of linear equations as
\[ LUx = w, \]
and since the determinant of \( L \) is equal to 1 (by construction since the diagonals of \( L \) equal 1) we can use the inverse of \( L \) to obtain

\[
Ux = L^{-1}w = y,
\]

which yields the intermediate step

\[
L^{-1}w = y
\]

and multiplying with \( L \) on both sides we reobtain Eq. (6.22). As soon as we have \( y \) we can obtain \( x \) through \( Ux = y \).

For our four-dimentional example this takes the form

\[
\begin{align*}
y_1 &= w_1 \\
l_{21}y_1 + y_2 &= w_2 \\
l_{31}y_1 + l_{32}y_2 + y_3 &= w_3 \\
l_{41}y_1 + l_{42}y_2 + l_{43}y_3 + y_4 &= w_4,
\end{align*}
\]

and

\[
\begin{align*}
u_{11}x_1 + u_{12}x_2 + u_{13}x_3 + u_{14}x_4 &= y_1 \\
u_{22}x_2 + u_{23}x_3 + u_{24}x_4 &= y_2 \\
u_{33}x_3 + u_{34}x_4 &= y_3 \\
u_{44}x_4 &= y_4
\end{align*}
\]

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++: \( \text{ludcmp(double a, int n, int indx, double d)} \)
  
  Fortran: 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++: \( \text{lubksb(double a, int n, int indx, double w)} \)
  
  Fortran: 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.

### 6.4.4 Inverse of a Matrix and the Determinant

The basic definition of the determinant of \( A \) is
\[ \text{det} \{ A \} = \sum_p (-1)^p a_{1p_1} a_{2p_2} \cdots a_{np_n}, \]

where the sum runs over all permutations \( p \) of the indices 1, 2, \ldots, \( n \), altogether \( n! \) terms. To calculate the inverse of \( A \) is a formidable task. Here we have to calculate the complementary cofactor \( a_{ij} \) of each element \( a_{ij} \) 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 of a matrix with the elements \( (-1)^{i+j} a_{ij} \). 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. (6.18) it is rather easy to find the determinant

\[ \text{det} \{ A \} = \text{det} \{ L \} \times \text{det} \{ U \} = \text{det} \{ U \}, \]

since the diagonal elements of \( L \) equal 1. Thus the determinant can be written

\[ \text{det} \{ A \} = \prod_{k=1}^{n} u_{kk}. \]

The inverse is slightly more difficult. However, with an LU decomposed matrix this reduces to solving a set of linear equations. To see this, we recall that if the inverse exists then

\[ A^{-1}A = I, \]

the identity matrix. With an LU decomposed matrix we can rewrite the last equation as

\[ LUA^{-1} = I. \]

If we assume that the first column (that is column 1) of the inverse matrix can be written as a vector with unknown entries

\[ A^{-1}_1 = \begin{pmatrix} a_{11}^{-1} \\ a_{21}^{-1} \\ \vdots \\ a_{n1}^{-1} \end{pmatrix}, \]

then we have a linear set of equations

\[ LU \begin{pmatrix} a_{11}^{-1} \\ a_{21}^{-1} \\ \vdots \\ a_{n1}^{-1} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}. \]

In a similar way we can compute the unknown entries of the second column,

\[ LU \begin{pmatrix} a_{12}^{-1} \\ a_{22}^{-1} \\ \vdots \\ a_{n2}^{-1} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix}, \]

and continue till we have solved all \( n \) sets of linear equations.

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 the sets of linear equations.

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

http://folk.uio.no/compphys/programs/chapter06/cpp/program1.cpp

```cpp
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 linear equations 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 following program and reads

http://folk.uio.no/compphys/programs/chapter06/cpp/program1.cpp

```cpp
#include <iostream>
#include <new>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cstring>
```

```cpp
// Simple matrix inversion example
```
#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. Alternatively, you could have used either Blitz++ or Armadillo.

The corresponding Fortran program for the inverse of a matrix reads:

http://folk.uio.no/compphys/programs/FYS3150/f90library/f90lib.f90

! Routines to do mtx inversion, from Numerical
! Recipes, Teukolsky et al. Routines included
! below are MATINV, LUDCMP and LUBKS. 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
The Fortran 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.

### 6.4.4.1 Scattering Equation and Principal Value Integrals via Matrix Inversion

In quantum mechanics, it is often common to rewrite Schrödinger’s equation in momentum space, after having made a so-called partial wave expansion of the interaction. We will not go into the details of these expressions but limit ourselves to study the equivalent problem for so-called scattering states, meaning that the total energy of two particles which collide is larger than or equal zero. The benefit of rewriting the equation in momentum space, after having performed a Fourier transformation, is that the coordinate space equation, being an integro-differential equation, is transformed into an integral equation. The latter can be solved by standard matrix inversion techniques. Furthermore, the results of solving these equation can be related directly to experimental observables like the scattering phase shifts. The latter tell us how much the incoming two-particle wave function is modified by a collision. Here we take a more technical stand and consider the technical aspects of solving an integral equation with a principal value.

For scattering states, \( E > 0 \), the corresponding equation to solve is the so-called Lippman-Schwinger equation. This is an integral equation where we have to deal with the amplitude \( R(k,k') \) (reaction matrix) defined through the integral equation

\[
R_l(k,k') = V_l(k,k') + \frac{2}{\pi} \mathcal{P} \int_0^\infty dq q^2 V_l(k,q) \frac{1}{E - q^2/m} R_l(q,k'),
\]

(6.23)

where the total kinetic energy of the two incoming particles in the center-of-mass system is

\[
E = \frac{k_0^2}{m}.
\]

(6.24)

The symbol \( \mathcal{P} \) indicates that Cauchy’s principal-value prescription is used in order to avoid the singularity arising from the zero of the denominator. We will discuss below how to solve this problem. Equation (6.23) represents then the problem you will have to solve numerically. The interaction between the two particles is given by a partial-wave decomposed version \( V_l(k,k') \), where \( l \) stands for a quantum number like the orbital momentum. We have assumed that interaction does not coupled to partial waves with different orbital momenta. The variables \( k \) and \( k' \) are the outgoing and incoming relative momenta of the two interacting particles.

The matrix \( R_l(k,k') \) relates to the experimental the phase shifts \( \delta_l \) through its diagonal elements as

\[
R_l(k_0,k_0) = -\frac{\tan \delta_l}{mk_0},
\]

(6.25)
where \( m \) is the reduced mass of the interacting particles. Furthermore, the interaction between the particles, \( V \), carries.

In order to solve the Lippman-Schwinger equation in momentum space, we need first to write a function which sets up the integration points. We need to do that since we are going to approximate the integral through

\[
\int_{a}^{b} f(x)dx \approx \sum_{i=1}^{N} w_i f(x_i),
\]

where we have fixed \( N \) integration points through the corresponding weights \( w_i \) and points \( x_i \). These points can for example be determined using Gaussian quadrature.

The principal value in Eq. (6.23) is rather tricky to evaluate numerically, mainly since computers have limited precision. We will here use a subtraction trick often used when dealing with singular integrals in numerical calculations. We use the calculus relation from the previous section

\[
\int_{-\infty}^{\infty} \frac{dk}{k-k_0} = 0,
\]

or

\[
\int_{0}^{\infty} \frac{dk}{k^2-k_0^2} = 0.
\]

We can use this to express a principal values integral as

\[
\mathcal{P} \int_{0}^{\infty} f(k)\frac{dk}{k^2-k_0^2} = \int_{0}^{\infty} \frac{(f(k) - f(k_0))dk}{k^2-k_0^2}, \tag{6.26}
\]

where the right-hand side is no longer singular at \( k = k_0 \), it is proportional to the derivative \( df/dk \), and can be evaluated numerically as any other integral.

We can then use the trick in Eq. (6.26) to rewrite Eq. (6.23) as

\[
R(k,k') = V(k,k') + \frac{2}{\pi} \int_{0}^{\infty} dq q^2 V(q,q)R(q,k') - \frac{k_0^2 V(k,k_0)R(k_0,k')}{(k_0^2 - q^2)/m}. \tag{6.27}
\]

We are interested in obtaining \( R(k_0,k_0) \), since this is the quantity we want to relate to experimental data like the phase shifts.

How do we proceed in order to solve Eq. (6.27)?

1. Using the mesh points \( k_j \) and the weights \( \omega_j \), we can rewrite Eq. (6.27) as

\[
R(k,k') = V(k,k') + \frac{2}{\pi} \sum_{j=1}^{N} \omega_j k_j^2 V(k,k_j)R(k_j,k') - \frac{2}{\pi} k_0^2 V(k,k_0)R(k_0,k') \sum_{n=1}^{N} \frac{\omega_n}{(k_0^2 - k_n^2)/m}. \tag{6.28}
\]

This equation contains now the unknowns \( R(k_i,k_j) \) (with dimension \( N \times N \)) and \( R(k_0,k_0) \).

2. We can turn Eq. (6.28) into an equation with dimension \( (N+1) \times (N+1) \) with an integration domain which contains the original mesh points \( k_j \) for \( j = 1,N \) and the point which corresponds to the energy \( k_0 \). Consider the latter as the ‘observable’ point. The mesh points become then \( k_j \) for \( j = 1,n \) and \( k_{N+1} = k_0 \).

3. With these new mesh points we define the matrix

\[
A_{i,j} = \delta_{i,j} - V(k_i,k_j)u_j, \tag{6.29}
\]

where \( \delta \) is the Kronecker \( \delta \) and

\[
u_j = \frac{2}{\pi} \frac{\omega_j k_j^2}{(k_0^2 - k_j^2)/m} \quad j = 1,N \tag{6.30}
\]
and
\[ u_{N+1} = -2 \frac{\sum_{j=1}^{N} k_0^2 \omega_j}{\pi (k_0^2 - k_j^2)/m}. \] (6.31)

The first task is then to set up the matrix \( A \) for a given \( k_0 \). This is an \((N+1) \times (N+1)\) matrix. It can be convenient to have an outer loop which runs over the chosen observable values for the energy \( k_0^2/m \). Note that all mesh points \( k_j \) for \( j = 1, N \) must be different from \( k_0 \). Note also that \( V(k_i, k_j) \) is an \((N+1) \times (N+1)\) matrix.

4. With the matrix \( A \) we can rewrite Eq. (6.28) as a matrix problem of dimension \((N+1) \times (N+1)\). All matrices \( R, A \) and \( V \) have this dimension and we get
\[ A_{i,j} R_{i,j} = V_{i,j}, \] (6.32)
or just
\[ AR = V. \] (6.33)

5. Since we already have defined \( A \) and \( V \) (these are stored as \((N+1) \times (N+1)\) matrices) Eq. (6.33) involves only the unknown \( R \). We obtain it by matrix inversion, i.e.,
\[ R = A^{-1}V. \] (6.34)
Thus, to obtain \( R \), we need to set up the matrices \( A \) and \( V \) and invert the matrix \( A \). With the inverse \( A^{-1} \) we perform a matrix multiplication with \( V \) and obtain \( R \).

With \( R \) we can in turn evaluate the phase shifts by noting that
\[ R(k_{N+1}, k_{N+1}) = R(k_0, k_0), \] (6.35)
and we are done.

### 6.4.4.2 Inverse of the Vandermonde Matrix

In chapter 3 we discussed 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 expression 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
6.4 Linear Systems

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

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

\[
\begin{pmatrix}
1 & x_0 & x_0^2 & \cdots & x_0^n \\
1 & x_1 & x_1^2 & \cdots & x_1^n \\
1 & x_2 & x_2^2 & \cdots & x_2^n \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_n & x_n^2 & \cdots & x_n^n
\end{pmatrix}
\begin{pmatrix}
a_0 \\
a_1 \\
a_2 \\
\vdots \\
a_n
\end{pmatrix}
= \begin{pmatrix}
f(x_0) \\
f(x_1) \\
f(x_2) \\
\vdots \\
f(x_n)
\end{pmatrix},
\]

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

\[ X a = f, \]

with

\[ X = \begin{pmatrix}
1 & x_0 & x_0^2 & \cdots & x_0^n \\
1 & x_1 & x_1^2 & \cdots & x_1^n \\
1 & x_2 & x_2^2 & \cdots & x_2^n \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_n & x_n^2 & \cdots & x_n^n
\end{pmatrix}. \]

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 \( X \), resulting in

\[ a = X^{-1} f. \]

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 discussed in chapter 3, together with spline interpolation discussed in the next section, 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. Spline interpolation will be discussed in the next section.

6.4.5 Tridiagonal Systems of Linear Equations

We start with the linear set of equations from Eq. (6.15), viz

\[ A u = f, \]

where \( A \) is a tridiagonal matrix which we rewrite as

\[
A = \begin{pmatrix}
b_1 & c_1 & 0 & \cdots & \cdots \\
a_2 & b_2 & c_2 & \cdots & \cdots \\
a_3 & b_3 & c_3 & \cdots & \cdots \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
a_{n-2} & b_{n-2} & c_{n-2} & \cdots & a_{n-1} \\
a_{n-1} & b_{n-1} & c_{n-1} & \cdots & \cdots \\
a_{n} & 0 & \cdots & \cdots & a_{n}
\end{pmatrix}
\]
where \( a, b, c \) are one-dimensional arrays of length \( 1 : n \). In the example of Eq. (6.15) the arrays \( a \) and \( c \) are equal, namely \( a_i = c_i = -1/h^2 \). We can rewrite Eq. (6.15) as

\[
A u = \begin{pmatrix}
    b_1 & c_1 & 0 & \cdots & \cdots & \cdots \\
    a_2 & b_2 & c_2 & \cdots & \cdots & \cdots \\
    & a_3 & b_3 & c_3 & \cdots & \cdots \\
    & \cdots & \cdots & \cdots & \cdots & \cdots \\
    & a_{n-2} & b_{n-1} & c_{n-1} & \cdots & \cdots \\
    & a_{n-1} & b_n & & & & \\
\end{pmatrix}
\begin{pmatrix}
    u_1 \\
    u_2 \\
    \vdots \\
    \vdots \\
    \vdots \\
    u_n \\
\end{pmatrix}
= \begin{pmatrix}
    f_1 \\
    f_2 \\
    \vdots \\
    \vdots \\
    \vdots \\
    f_n \\
\end{pmatrix}.
\]

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 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 10.

Our algorithm starts with forward substitution with a loop over of the elements \( i \) and can be expressed via the following piece of code taken from the Numerical Recipe text of Teukolsky et al [36]

```python
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

```python
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 \( A \) which rephrases a second derivative in a discretized form

\[
A = \begin{pmatrix}
    2 & -1 & 0 & 0 & 0 & 0 \\
    -1 & 2 & -1 & 0 & 0 & 0 \\
    0 & -1 & 2 & -1 & 0 & 0 \\
    \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
    0 & 0 & 0 & -1 & 2 & -1 \\
    0 & 0 & 0 & 0 & -1 & 2 \\
\end{pmatrix}
\]
6.5 Spline Interpolation

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 that \( f, g \) 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 \( x_i = a + ih \), 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

\[
\frac{u^\prime\prime}{h^2} \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2},
\]

for the second derivative while the first derivative is given by

\[
\frac{u^\prime}{h} \approx \frac{u_{i+1} - u_{i-1}}{2h}.
\]

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

\[
-\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + g_i\frac{u_{i+1} - u_{i-1}}{2h} + hu_i = f_i,
\]

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.

6.5 Spline Interpolation

Cubic spline interpolation is among one of the most used methods for interpolating between data points where the arguments are organized as ascending series. In the library program we supply such a function, based on the so-called cubic spline method to be described below. The linear equation solver we developed in the previous section for tridiagonal matrices can be reused for spline interpolation.

A spline function consists of polynomial pieces defined on subintervals. The different subintervals are connected via various continuity relations.

Assume we have at our disposal \( n + 1 \) points \( x_0, x_1, \ldots, x_n \) arranged so that \( x_0 < x_1 < x_2 < \ldots x_{n-1} < x_n \) (such points are called knots). A spline function \( s \) of degree \( k \) with \( n + 1 \) knots is defined as follows

- On every subinterval \([x_{i-1}, x_i)\) \( s \) is a polynomial of degree \( \leq k \).
- \( s \) has \( k - 1 \) continuous derivatives in the whole interval \([x_0, x_n]\).

As an example, consider a spline function of degree \( k = 1 \) defined as follows
The number of continuous derivatives is the one we deal with straight lines. Such a polynomial is quite easy to construct given \( n + 1 \) points \( x_0, x_1, \ldots, x_n \) and their corresponding function values.

The most commonly used spline function is the one with \( k = 3 \), the so-called cubic spline function. Assume that we have in addition to the \( n + 1 \) knots a series of functions values \( y_0 = f(x_0), y_1 = f(x_1), \ldots, y_n = f(x_n) \). By definition, the polynomials \( s_{i-1} \) and \( s_i \) are hence supposed to interpolate the same point \( i \), i.e.,

\[
s_{i-1}(x_i) = y_i = s_i(x_i),
\]
with \( 1 \leq i \leq n - 1 \). In total we have \( n \) polynomials of the type

\[
s_i(x) = a_{i0} + a_{i1}x + a_{i2}x^2 + a_{i3}x^3,
\]
yielding \( 4n \) coefficients to determine. Every subinterval provides in addition two conditions

\[
y_i = s(x_i),
\]
and

\[
y_{i+1} = s(x_{i+1}),
\]
to be fulfilled. If we also assume that \( s' \) and \( s'' \) are continuous, then

\[
s'_{i-1}(x_i) = s'_i(x_i),
\]
yields \( n - 1 \) conditions. Similarly,

\[
s''_{i-1}(x_i) = s''_i(x_i),
\]
results in additional \( n - 1 \) conditions. In total we have \( 4n \) coefficients and \( 4n - 2 \) equations to determine them, leaving us with 2 degrees of freedom to be determined.

Using the last equation we define two values for the second derivative, namely

\[
s''_i(x_i) = f_i,
\]
and

\[
s''_i(x_{i+1}) = f_{i+1},
\]
and setting up a straight line between \( f_i \) and \( f_{i+1} \) we have

\[
s'_i(x) = \frac{f_i}{x_{i+1} - x_i}(x_{i+1} - x_i) + \frac{f_{i+1}}{x_{i+1} - x_i}(x - x_i),
\]
and integrating twice one obtains

\[
s_i(x) = \frac{f_i}{6(x_{i+1} - x_i)}(x_{i+1} - x_i)^3 + \frac{f_{i+1}}{6(x_{i+1} - x_i)}(x_{i+1} - x_i)^3 + c(x - x_i) + d(x_{i+1} - x).
\]

Using the conditions \( s_i(x_i) = y_i \) and \( s_i(x_{i+1}) = y_{i+1} \) we can in turn determine the constants \( c \) and \( d \) resulting in
6.6 Iterative Methods

\[ s_i(x) = \frac{f_i}{6(x_{i+1} - x)}(x_{i+1} - x)^3 + \frac{f_{i+1}}{6(x_{i+1} - x)}(x - x_i)^3 + \frac{y_{i+1}}{h_{i+1}}(x - x_i) + \frac{y_i}{h_i} - \frac{f_i(x_{i+1} - x)}{6h_i}(x_{i+1} - x). \] (6.47)

How to determine the values of the second derivatives \( f_i \) and \( f_{i+1} \)? We use the continuity assumption of the first derivatives.

\[ s'_{i-1}(x_i) = s'_i(x_i), \] (6.48)

and set \( x = x_i \). Defining \( h_i = x_{i+1} - x_i \) we obtain finally the following expression

\[ h_{i-1}f_{i-1} + 2(h_i + h_{i-1})f_i + h_if_{i+1} = \frac{6}{h_i}(y_{i+1} - y_i) - \frac{6}{h_{i-1}}(y_i - y_{i-1}), \] (6.49)

and introducing the shorthands \( u_i = 2(h_i + h_{i-1}) \), \( v_i = \frac{6}{h_i}(y_{i+1} - y_i) - \frac{6}{h_{i-1}}(y_i - y_{i-1}) \), we can reformulate the problem as a set of linear equations to be solved through e.g., Gaussian elimination, namely

\[
\begin{bmatrix}
  u_1 & h_1 & 0 & \ldots \\
  h_1 & u_2 & h_2 & 0 & \ldots \\
  0 & h_2 & u_3 & h_3 & 0 & \ldots \\
  \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
  \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\
  \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & u_{n-2} & h_{n-2} & 0 \\
  \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
  \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
  \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
  \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
  0 & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\end{bmatrix}
\begin{bmatrix}
  f_1 \\
  f_2 \\
  f_3 \\
  \vdots \\
  f_{n-2} \\
  f_{n-1} \\
\end{bmatrix}
= 
\begin{bmatrix}
  v_1 \\
  v_2 \\
  v_3 \\
  \vdots \\
  v_{n-2} \\
  v_{n-1} \\
\end{bmatrix}.
\] (6.50)

Note that this is a set of tridiagonal equations and can be solved through only \( O(n) \) operations.

It is easy to write your own program for the cubic spline method when you have written a solver for tridiagonal equations. We split the program into two tasks, one which finds the polynomial approximation and one which uses the polynomials approximation to find an interpolated value for a function. These functions are included in the programs of this chapter; see the codes cubicpsline.cpp and cubicsinterpol.cpp. Alternatively, you can solve exercise 6.4!

6.6 Iterative Methods

Till now we have dealt with so-called direct solvers such as Gaussian elimination and LU decomposition. Iterative solvers offer another strategy and are much used in partial differential equations. We start with a guess for the solution and then iterate till the solution does not change anymore.

6.6.1 Jacobi’s method

It is a simple method for solving

\[ \hat{A}x = b, \]

where \( \hat{A} \) is a matrix and \( x \) and \( b \) are vectors. The vector \( x \) is the unknown.

It is an iterative scheme where we start with a guess for the unknown, and after \( k + 1 \) iterations we have

\[ x^{(k+1)} = \hat{D}^{-1}(b - (\hat{L} + \hat{U})x^{(k)}), \]

with \( \hat{A} = \hat{D} + \hat{U} + \hat{L} \) and \( \hat{D} \) being a diagonal matrix, \( \hat{U} \) an upper triangular matrix and \( \hat{L} \) a lower triangular matrix.
If the matrix $\hat{A}$ is positive definite or diagonally dominant, one can show that this method will always converge to the exact solution.

We can demonstrate Jacobi’s method by a $4 \times 4$ matrix problem. We assume a guess for the initial vector elements, labeled $x_i^{(0)}$. This guess represents our first iteration. The new values are obtained by substitution

$$
x_1^{(1)} = \frac{b_1 - a_{12}x_2^{(0)} - a_{13}x_3^{(0)} - a_{14}x_4^{(0)}}{a_{11}}
$$
$$
x_2^{(1)} = \frac{b_2 - a_{21}x_1^{(0)} - a_{23}x_3^{(0)} - a_{24}x_4^{(0)}}{a_{22}}
$$
$$
x_3^{(1)} = \frac{b_3 - a_{31}x_1^{(0)} - a_{32}x_2^{(0)} - a_{34}x_4^{(0)}}{a_{33}}
$$
$$
x_4^{(1)} = \frac{b_4 - a_{41}x_1^{(0)} - a_{42}x_2^{(0)} - a_{43}x_3^{(0)}}{a_{44}},
$$

which after $k + 1$ iterations result in

$$
x_1^{(k+1)} = \frac{b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)}}{a_{11}}
$$
$$
x_2^{(k+1)} = \frac{b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)}}{a_{22}}
$$
$$
x_3^{(k+1)} = \frac{b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)} - a_{34}x_4^{(k)}}{a_{33}}
$$
$$
x_4^{(k+1)} = \frac{b_4 - a_{41}x_1^{(k)} - a_{42}x_2^{(k)} - a_{43}x_3^{(k)}}{a_{44}},
$$

We can generalize the above equations to

$$
x_i^{(k+1)} = \frac{b_i - \sum_{j=1, j\neq i}^n a_{ij}x_j^{(k)}}{a_{ii}}
$$

or in an even more compact form as

$$
x^{(k+1)} = D^{-1}(b - (\hat{L} + \hat{U})x^{(k)}),
$$

with $\hat{A} = \hat{D} + \hat{U} + \hat{L}$ and $\hat{D}$ being a diagonal matrix, $\hat{U}$ an upper triangular matrix and $\hat{L}$ a lower triangular matrix.

### 6.6.2 Gauss-Seidel

Our $4 \times 4$ matrix problem

$$
x_1^{(k+1)} = \frac{b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)}}{a_{11}}
$$
$$
x_2^{(k+1)} = \frac{b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)}}{a_{22}}
$$
$$
x_3^{(k+1)} = \frac{b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)} - a_{34}x_4^{(k)}}{a_{33}}
$$
$$
x_4^{(k+1)} = \frac{b_4 - a_{41}x_1^{(k)} - a_{42}x_2^{(k)} - a_{43}x_3^{(k)}}{a_{44}},
$$

can be rewritten as

$$
x_1^{(k+1)} = \frac{b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)}}{a_{11}}
$$
$$
x_2^{(k+1)} = \frac{b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)}}{a_{22}}
$$
$$
x_3^{(k+1)} = \frac{b_3 - a_{31}x_1^{(k+1)} - a_{32}x_2^{(k+1)} - a_{34}x_4^{(k)}}{a_{33}}
$$
$$
x_4^{(k+1)} = \frac{b_4 - a_{41}x_1^{(k+1)} - a_{42}x_2^{(k+1)} - a_{43}x_3^{(k+1)}}{a_{44}},
$$
which allows us to utilize the preceding solution (forward substitution). This improves normally the convergence behavior and leads to the Gauss-Seidel method!

We can generalize these equations to the following form

\[ x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} - \sum_{j \neq i} a_{ij} x_j^{(k+1)} \right), \quad i = 1, 2, \ldots, n. \]

The procedure is generally continued until the changes made by an iteration are below some tolerance.

The convergence properties of the Jacobi method and the Gauss-Seidel method depend on the matrix \( \hat{A} \). These methods converge when the matrix is symmetric positive-definite, or is strictly or irreducibly diagonally dominant. Both methods sometimes converge even if these conditions are not satisfied.

### 6.6.3 Successive over-relaxation

We can rewrite the above in a slightly more formal way and extend the methods to what is called successive over-relaxation. Given a square system of \( n \) linear equations with unknown \( x \):

\[ \hat{A}x = b \]

where:

\[ \hat{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}. \]

Then \( A \) can be decomposed into a diagonal component \( D \), and strictly lower and upper triangular components \( L \) and \( U \):

\[ \hat{A} = D + L + U, \]

where

\[ D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & a_{21} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}. \]

The system of linear equations may be rewritten as:

\[ (D + \omega L)x = \omega b - [\omega U + (\omega - 1)D]x \]

for a constant \( \omega > 1 \). The method of successive over-relaxation is an iterative technique that solves the left hand side of this expression for \( x \), using previous value for \( x \) on the right hand side. Analytically, this may be written as:

\[ x^{(k+1)} = (D + \omega L)^{-1} (\omega b - [\omega U + (\omega - 1)D]x^{(k)}). \]

However, by taking advantage of the triangular form of \( (D + \omega L) \), the elements of \( x^{(k+1)} \) can be computed sequentially using forward substitution:

\[ x_i^{(k+1)} = (1 - \omega)x_i^{(k)} + \frac{\omega}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} - \sum_{j < i} a_{ij} x_j^{(k+1)} \right), \quad i = 1, 2, \ldots, n. \]
The choice of relaxation factor is not necessarily easy, and depends upon the properties of the coefficient matrix. For symmetric, positive-definite matrices it can be proven that $0 < \omega < 2$ will lead to convergence, but we are generally interested in faster convergence rather than just convergence.

### 6.6.4 Conjugate Gradient Method

The success of the Conjugate Gradient method for finding solutions of non-linear problems is based on the theory of conjugate gradients for linear systems of equations. It belongs to the class of iterative methods for solving problems from linear algebra of the type

$$\hat{A}\hat{x} = \hat{b}.$$

In the iterative process we end up with a problem like

$$\hat{r} = \hat{b} - \hat{A}\hat{x},$$

where $\hat{r}$ is the so-called residual or error in the iterative process.

The residual is zero when we reach the minimum of the quadratic equation

$$P(\hat{x}) = \frac{1}{2}\hat{x}^T\hat{A}\hat{x} - \hat{x}^T\hat{b},$$

with the constraint that the matrix $\hat{A}$ is positive definite and symmetric. If we search for a minimum of the quantum mechanical variance, then the matrix $\hat{A}$, which is called the Hessian, is given by the second-derivative of the variance. This quantity is always positive definite. If we vary the energy, the Hessian may not always be positive definite.

In the Conjugate Gradient method we define so-called conjugate directions and two vectors $\hat{s}$ and $\hat{t}$ are said to be conjugate if

$$\hat{s}^T\hat{A}\hat{t} = 0.$$

The philosophy of the Conjugate Gradient method is to perform searches in various conjugate directions of our vectors $\hat{x}$, obeying the above criterion, namely

$$\hat{x}_i^T\hat{A}\hat{x}_j = 0.$$

Two vectors are conjugate if they are orthogonal with respect to this inner product. Being conjugate is a symmetric relation: if $\hat{s}$ is conjugate to $\hat{t}$, then $\hat{t}$ is conjugate to $\hat{s}$.

An example is given by the eigenvectors of the matrix

$$\hat{v}_i^T\hat{A}\hat{v}_j = \lambda\hat{v}_i^T\hat{v}_j,$$

which is zero unless $i = j$.

Assume now that we have a symmetric positive-definite matrix $\hat{A}$ of size $n \times n$. At each iteration $i + 1$ we obtain the conjugate direction of a vector

$$\hat{x}_{i+1} = \hat{x}_i + \alpha_i\hat{p}_i.$$

We assume that $\hat{p}_i$ is a sequence of $n$ mutually conjugate directions. Then the $\hat{p}_i$ form a basis of $\mathbb{R}^n$ and we can expand the solution $\hat{A}\hat{x} = \hat{b}$ in this basis, namely

$$\hat{x} = \sum_{i=1}^{n} \alpha_i\hat{p}_i.$$
6.6 Iterative Methods

The coefficients are given by

\[ Ax = \sum_{i=1}^{n} \alpha_i A p_i = b. \]

Multiplying with \( \hat{p}_k^T \) from the left gives

\[ \hat{p}_k^T \hat{A} \hat{x} = \sum_{i=1}^{n} \alpha_i \hat{p}_k^T \hat{A} \hat{p}_i = \hat{p}_k^T b, \]

and we can define the coefficients \( \alpha_k \) as

\[ \alpha_k = \frac{\hat{p}_k^T \hat{b}}{\hat{p}_k^T \hat{A} \hat{p}_k}. \]

If we choose the conjugate vectors \( \hat{p}_k \) carefully, then we may not need all of them to obtain a good approximation to the solution \( \hat{x} \). So, we want to regard the conjugate gradient method as an iterative method. This also allows us to solve systems where \( n \) is so large that the direct method would take too much time.

We denote the initial guess for \( \hat{x} \) as \( \hat{x}_0 \). We can assume without loss of generality that

\[ \hat{x}_0 = 0, \]

or consider the system

\[ \hat{A} \hat{z} = \hat{b} - \hat{A} \hat{x}_0, \]

instead.

One can show that the solution \( \hat{x} \) is also the unique minimizer of the quadratic form

\[ f(\hat{x}) = \frac{1}{2} \hat{x}^T \hat{A} \hat{x} - \hat{x}^T \hat{A} \hat{b}, \quad \hat{x} \in \mathbb{R}^n. \]

This suggests taking the first basis vector \( \hat{p}_1 \) to be the gradient of \( f \) at \( \hat{x} = \hat{x}_0 \), which equals

\[ \hat{A} \hat{x}_0 - \hat{b}, \]

and \( \hat{x}_0 = 0 \) it is equal \( -\hat{b} \). The other vectors in the basis will be conjugate to the gradient, hence the name conjugate gradient method.

Let \( \hat{r}_k \) be the residual at the \( k \)-th step:

\[ \hat{r}_k = \hat{b} - \hat{A} \hat{x}_k. \]

Note that \( \hat{r}_k \) is the negative gradient of \( f \) at \( \hat{x} = \hat{x}_k \), so the gradient descent method would be to move in the direction \( \hat{r}_k \). Here, we insist that the directions \( \hat{p}_k \) are conjugate to each other, so we take the direction closest to the gradient \( \hat{r}_k \) under the conjugacy constraint. This gives the following expression

\[ \hat{p}_{k+1} = \hat{r}_k - \frac{\hat{p}_k^T \hat{A} \hat{r}_k}{\hat{p}_k^T \hat{A} \hat{p}_k} \hat{p}_k. \]

We can also compute the residual iteratively as

\[ \hat{r}_{k+1} = \hat{b} - \hat{A} \hat{x}_{k+1}, \]

which equals

\[ \hat{b} - \hat{A} (\hat{x}_k + \alpha_k \hat{p}_k), \]

or
\[(\hat{b} - \hat{A} \hat{x}_k) - \alpha_k \hat{A} \hat{p}_k,\]

which gives

\[\hat{r}_{k+1} = \hat{r}_k - \hat{A} \hat{p}_k.\]

If we consider finding the minimum of a function \(f\) using Newton’s method, that implies a search for a zero of the gradient of a function. Near a point \(x_i\) we have to second order

\[f(\hat{x}) = f(\hat{x}_i) + (\hat{x} - \hat{x}_i) \nabla f(\hat{x}_i) + \frac{1}{2} (\hat{x} - \hat{x}_i) \hat{A} (\hat{x} - \hat{x}_i),\]

giving

\[\nabla f(\hat{x}) = \nabla f(\hat{x}_i) + \hat{A} (\hat{x} - \hat{x}_i).\]

In Newton’s method we set \(\nabla f = 0\) and we can thus compute the next iteration point

\[\hat{x} - \hat{x}_i = \hat{A}^{-1} \nabla f(\hat{x}_i).\]

Subtracting this equation from that of \(\hat{x}_{i+1}\) we have

\[\hat{x}_{i+1} - \hat{x}_i = \hat{A}^{-1} (\nabla f(\hat{x}_{i+1}) - \nabla f(\hat{x}_i)).\]

6.7 A vector and matrix class

We end this chapter by presenting a class which allows to manipulate one- and two-dimensional arrays. However, before we proceed, we would like to come with some general recommendations. Although it is useful to write your own classes, like the one included here, in general these classes may not be very efficient from a computational point of view. There are several libraries which include many interesting array features that allow us to write more compact code. The latter has the advantage that the code is lost likely easier to debug in case of errors (obviously assuming that the library is functioning correctly). Furthermore, if the proper functionalities are included, the final code may closely resemble the mathematical operations we wish to perform, increasing considerably the readability of our program. And finally, the code is in almost all cases much faster than the one we wrote!

In particular, we would like to recommend the C++ linear algebra library Armadillo, see http://arma.sourceforge.net. For those of you who are familiar with compiled programs like Matlab, the syntax is deliberately similar. Integer, floating point and complex numbers are supported, as well as a subset of trigonometric and statistics functions. Various matrix decompositions are provided through optional integration with LAPACK, or one of its high performance drop-in replacements (such as the multi-threaded MKL or ACML libraries). The selected examples included here show some examples on how to declare arrays and rearrange arrays or perform mathematical operations on say vectors or matrices. The first example here defines two random matrices of dimensionality \(10 \times 10\) and performs a matrix-matrix multiplication using the \textit{dgemm} function of the library BLAS.

Simple matrix-matrix multiplication of two random matrices

```cpp
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

int main(int argc, char** argv) {
```
6.7 A vector and matrix class

mat A = randu<mat>(10,10);
mat B = randu<mat>(10,10);
// Matrix-matrix multiplication
cout << A* B << endl;
return 0;
}

In the next example we compute the determinant of a $5 \times 5$ matrix, its inverse and perform thereafter several operations on various matrices.

Determinant and inverse of a matrix

```cpp
#include <iostream>
#include "armadillo"
using namespace arma;
using namespace std;

int main(int argc, char** argv)
{
    cout << "Armadillo version: " << arma_version::as_string() << endl;
    mat A;
    // Hard coding of the matrix
    // endr indicates "end of row"
    A << 0.16530 << 0.454037 << 0.955795 << 0.124098 << 0.047084 << endr
       << 0.688782 << 0.036549 << 0.558248 << 0.937664 << 0.866401 << endr
       << 0.348740 << 0.473886 << 0.562288 << 0.145673 << 0.491547 << endr
       << 0.148678 << 0.682258 << 0.571154 << 0.874724 << 0.444632 << endr
       << 0.245726 << 0.595218 << 0.409327 << 0.367827 << 0.385736 << endr;
    // rows = number of rows
    // cols = number of columns
    cout << "A.rows = " << A.n_rows << endl;
    cout << "A.cols = " << A.n_cols << endl;
    // Print the matrix A
    A.print("A =");
    // Computation of the determinant
    cout << "det(A) = " << det(A) << endl;
    // inverse
    cout << "inv(A) = " << endl << inv(A) << endl;
    // save to disk
    A.save("MatrixA.txt", raw_ascii);
    // Define a new matrix B which reads A from file
    mat B;
    B.load("MatrixA.txt");
    B += 5.0*A;
    B.print("The matrix B:");
    // generate the identity matrix
    mat C = eye<mat>(4,4);
    // transpose of B
    cout << "trans(B) = " << endl;
    // maximum from each column (traverse along rows)
    cout << "max(B) = " << endl;
    cout << max(B) << endl;
    // sum of all elements B
    cout << "sum(sum(B)) = " << sum(sum(B)) << endl;
    cout << "accu(B) = " << accu(B) << endl;
    // trace = sum along diagonal
    cout << "trace(B) = " << trace(B) << endl;
    // random matrix -- values are uniformly distributed in the [0,1] interval
    mat D = randu<mat>(4,4);
    D.print("Matrix D:");
    // sum of four matrices (no temporary matrices are created)
    mat E = A+B+C+D;
```
For more examples, please consult the online manual, see http://arma.sourceforge.net.

### 6.7.1 How to construct your own matrix-vector class

The rest of this section shows how one can build a matrix-vector class. We first give an example of a function which use the header file Array.h.

```cpp
#include "Array.h"
#include <iostream>
using namespace std;

int main()
{
    // Create an array with (default) nrows = 1, ncols = 1:
    Array<double> v1;

    // Redimension the array to have length n:
    int n1 = 3;
    v1.redim(n1);

    // Extract the length of the array:
    const int length = v1.getLength();

    // Create an array of specific length:
    int n2 = 5;
    Array<double> v2(n2);

    // Create an array as a copy of another one:
    Array<double> v5(v1);

    // Assign the entries in an array:
    v5(0) = 3.0;
    v5(1) = 2.5;
    v5(2) = 1.0;
    for(int i=0; i<3; i++)
    {
        cout << v5(i) << endl;
    }

    // Extract the ith component of an array:
    int i = 2;
    double value = v5(1);
    cout << "value: " << value << endl;

    // Set an array equal another one:
    Array<double> v6 = v5;
    for(int i=0; i<3; i++)
    {
        v1(i) = 1.0;
        v2(i) = 2.0;
    }

    // Create a two-dimensional array (matrix):
}
Array<

Array<double> matrix(2, 2);

// Fill the array:
matrix(0,0) = 1;
matrix(0,1) = 2;
matrix(1,0) = 3;
matrix(1,1) = 4;

// Get the entries in the array:
cout << "Matrix: " << endl;
for(int i=0; i<2; i++){
    for(int j=0; j<2; j++){
        cout << matrix(i,j) << " ";
    }
    cout << endl;
}
// Assign an entry of the matrix to a variable:
double scalar = matrix(0,0);
const double b = matrix(1,1);

Array<double> vector(2);
vector(0) = 1.0;
vector(1) = 2.0;

Array<double> v = vector;
Array<double> A = matrix;
Array<double> u(2);

// Arithmetic operations with arrays using a syntax close to the mathematical language
Array<double> w = v1 + 2.0*v2;

// Create multidimensional matrices and assign values to them:
int N = 3;
Array<double> multiD; multiD.redim(N,N,N);
for(int i=0; i<N; i++){
    for(int j=0; j<N; j++){
        for(int k=0; k<N; k++){
            cout << "multiD(i,j,k) = " << multiD(i,j,k) << endl;
        }
    }
}
multiD(1,2,3) = 4.0;
cout << "multiD(1,2,3) = " << multiD(1,2,3) << endl;

The header file follows here
```cpp
#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>
#include <sstream>
#include <iomanip>
#include <cstdlib>

using namespace std;

template<class T>
class Array{

private:
    static const int MAXDIM = 6;
    T *data ;
    int size[MAXDIM];
    int ndim;
    int length;
    int dx1, dx2, dx3, dx4, dx5;

    void allocate(int ni=0, int nj=0, int nk=0, int nl=0, int nm=0, int nn=0){
        ndim = MAXDIM;

        // Set the number of entries in each dimension.
        size[0]=ni;
        size[1]=nj;
        size[2]=nk;
        size[3]=nl;
        size[4]=nm;
        size[5]=nn;

        // Set the number of dimensions used.
        if(size[5] == 0)
            ndim--;
        if(size[4] == 0)
            ndim--;
        if(size[3] == 0)
            ndim--;
        if(size[2] == 0)
            ndim--;
        if(size[1] == 0)
            ndim--;
        if(size[0] == 0){
            ndim = 0;
            length = 0;
            data = NULL;
        }
        else{
            try{
                int i;

                // Set the length (total number of entries) of the one-dimensional array.
                length = 1;
                for(i=0; i<ndim; i++)
                    length *= size[i];

                data = new T[length];
                dx1 = size[0];
            }
        }
    }

```
dx2 = dx1 * size[1];
dx3 = dx2 * size[2];
dx4 = dx3 * size[3];
dx5 = dx4 * size[4];

} catch (std::bad_alloc &)
{
  std::cerr << "Array::allocate -- unable to allocate array of length " << length << std::endl;
  exit(1);
}

}

public:

/**
 * @brief Constructor with default arguments.
 * Creates an array with one or two-dimensions.
 * @param int nrows. Number of rows in the array.
 * @param int ncolsd. Number of columns in the array.
 **/
Array(int ni=0, int nj=0, int nk=0, int nl=0, int nm=0, int nn=0)
{
  // Allocate memory
  allocate(ni, nj, nk, nl, nm, nn);
} // end constructor

/*! Constructor */
Array(T* array, int ndim_, int size_[])
{
  ndim = ndim_;

  length = 1;
  int i;
  for(i=0; i<ndim; i++)
  {
    size[i] = size_[i]; // Copy only the ndim entries. The rest is zero by default.
    length *= size[i];
  }

  // Now when we known the length, we should not forget to allocate memory!!!!
  data = new T[length];

  // Copy the entries from array to data:
  for(i=0; i<length; i++)
  {
    data[i] = array[i];
  }
}

} // End constructor.

/*! Copy constructor */
Array(const Array<T>& array);

/*! Destructor */
~Array();
/**
 * @brief Checks the validity of the indexing.
 * @param i, an integer for indexing the rows.
 * @param j, an integer for indexing the columns.
 **/
 bool indexOk(int i, int j=0) const;

/**
 * @brief Change the dimensions of an array.
 * @param ni number of entries in the first dimension.
 * @param nj number of entries in the second dimension.
 * @param nk number of entries in the third dimension.
 * @param nl number of entries in the fourth dimension.
 * @param nm number of entries in the fifth dimension.
 * @param nn number of entries in the sixth dimension.
 **/
 bool redim(int ni, int nj=0, int nk=0, int nl=0, int nm=0, int nn=0);

/**
 * @return The total number of entries in the array, i.e., the sum of the entries in all
 * the dimensions.
 **/
 int getLength() const{return length;}

/**
 * @return The number of rows in a matrix.
 **/
 int getRows() const {return size[0];}

/**
 * @return Returns the number of columns in a matrix.
 **/
 int getColumns() const {return size[1];}

/** @brief Gives the number of entries in a dimension.
 * @param i An integer from 0 to 5 indicating the dimension we want to explore.
 * @return size[i] An integer for the number of elements in the dimension number i.
 **/
 int dimension(int i) const{return size[i];}

/**
 * The number of dimensions in the array.
 **/
 int getNDIM() const{return ndim;}

/**
 * @return A constant pointer to the array of data.
 * This function can be used to interface C++ with Python/Fortran/C.
 **/
 const T* getPtr() const;

/**
 * @return A pointer to the array of data.
 * This function can be used to interface C++ with Python/Fortran/C.
/**
 * @return A pointer to an array with information on the length of each dimension.
 */
int* getPtrSize();

/*******************************************************************************/
/* OPERATORS */
 /*******************************************************************************/

/// Assignment operator
Array<T>& operator=(const Array<T>& array);

/// Sum operator
Array<T> operator+(const Array<T>& array);

/// Substraction operator
Array<T> operator-(const Array<T>& array) const; // w=u-v;

/// Multiplication operator
//Array<T> operator*(const Array<T>& array);

/// Assignment by addition operator
Array<T>& operator+=(const Array<T>& w);

/// Assignment by substraction operator
Array<T>& operator-=(const Array<T>& w);

/// Assignment by scalar product operator
Array<T>& operator*=(double scalar);

/// Assignment by division operator
Array<T>& operator/=(double scalar);

/// Index operators
const T& operator[](int i) const;
const T& operator[](int i, int j) const;
const T& operator[](int i, int j, int k) const;
const T& operator[](int i, int j, int k, int l) const;
const T& operator[](int i, int j, int k, int l, int m) const;
const T& operator[](int i, int j, int k, int l, int m, int n) const;
T& operator[](int i);
T& operator[](int i, int j);
T& operator[](int i, int j, int k);
T& operator[](int i, int j, int k, int l);
T& operator[](int i, int j, int k, int l, int m);
T& operator[](int i, int j, int k, int l, int m, int n);
/***************************************************************/
/*                   FRIEND FUNCTIONS                    */
/***************************************************************/
//!! Unary operator +
template <class T2>
friend Array<T> operator+(const Array<T>&); // u = + v

//!! Unary operator -
template <class T2>
friend Array<T> operator-(const Array<T>&); // u = - v

/**
 * Premultiplication by a floating point number:
 * $\mathbf{u} = a \mathbf{v}$,
 * where $a$ is a scalar and $\mathbf{v}$ is a array.
 **/
template <class T2>
friend Array<T> operator*(double, const Array<T>&); // u = a*v

/**
 * Postmultiplication by a floating point number:
 * $\mathbf{u} = \mathbf{v} a$,
 * where $a$ is a scalar and $\mathbf{v}$ is a array.
 **/
template <class T2>
friend Array<T> operator*(const Array<T>&, double); // u = v*a

/**
 * Division of the entries of a array by a scalar.
 **/
template <class T2>
friend Array<T> operator/(const Array<T>&, double); // u = v/a

};

#include "Array.cpp"

// Destructor
template <class T>
inline Array<T>::~Array(){delete[] data;}

// Index operators
template <class T>
inline const T& Array<T>::operator[](int i) const {
    #if CHECKBOUNDS_ON
        indexOk(i);
    #endif
    return data[i];
}
template <class T>
inline const T& Array<T>::operator()(int i, int j) const {
    #if CHECKBOUNDS_ON
        indexOk(i, j);
    #endif

    return data[i + j*dx1];
}

template <class T>
inline const T& Array<T>::operator()(int i, int j, int k) const {
    #if CHECKBOUNDS_ON
        indexOk(i, j, k);
    #endif

    return data[i + j*dx1 + k*dx2];
}

template <class T>
inline const T& Array<T>::operator()(int i, int j, int k, int l) const {
    #if CHECKBOUNDS_ON
        indexOk(i, j, k, l);
    #endif

    return data[i + j*dx1 + k*dx2 + l*dx3];
}

template <class T>
inline const T& Array<T>::operator()(int i, int j, int k, int l, int m) const {
    #if CHECKBOUNDS_ON
        indexOk(i, j, k, l, m);
    #endif

    return data[i + j*dx1 + k*dx2 + l*dx3 + m*dx4];
}

template <class T>
inline const T& Array<T>::operator()(int i, int j, int k, int l, int m, int n) const {
    #if CHECKBOUNDS_ON
        indexOk(i, j, k, l, m, n);
    #endif

    return data[i + j*dx1 + k*dx2 + l*dx3 + m*dx4 + n*dx5];
}

template <class T>
inline T& Array<T>::operator()(int i) {
    #if CHECKBOUNDS_ON
        indexOk(i);
    #endif

    return data[i];
}

template <class T>
inline T& Array<T>::operator()(int i, int j) {
    #if CHECKBOUNDS_ON
        indexOk(i, j);
    #endif

    return data[i + j*dx1];
}
template <class T>
inline T& Array<T>::operator()(int i, int j, int k) {
  #if CHECKBOUNDS_ON
    indexOk(i, j, k);
  #endif
  return data[i + j*dx1 + k*dx2];
}

template <class T>
inline T& Array<T>::operator()(int i, int j, int k, int l) {
  #if CHECKBOUNDS_ON
    indexOk(i, j, k, l);
  #endif
  return data[i + j*dx1 + k*dx2 + l*dx3];
}

template <class T>
inline T& Array<T>::operator()(int i, int j, int k, int l, int m) {
  #if CHECKBOUNDS_ON
    indexOk(i, j, k, l, m);
  #endif
  return data[i + j*dx1 + k*dx2 + l*dx3 + m*dx4];
}

template <class T>
inline T& Array<T>::operator()(int i, int j, int k, int l, int m, int n) {
  #if CHECKBOUNDS_ON
    indexOk(i, j, k, l, m, n);
  #endif
  return data[i + j*dx1 + k*dx2 + l*dx3 + m*dx4 + n*dx5];
}

template <class T>
inline const T* Array<T>::getPtr() const {return data;}

template <class T>
inline T* Array<T>::getPtr(){return data;}

template <class T>
inline int* Array<T>::getPtrSize(){return size;}

// template <class T>
6.8 Exercises

6.1. The aim of this exercise is to write your own Gaussian elimination code.

1. Consider the linear system of equations

\[
\begin{align*}
    a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= w_1 \\
    a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= w_2 \\
    a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= w_3.
\end{align*}
\]
This can be written in matrix form as
\[ Ax = w. \]

We specialize here to the following case
\[-x_1 + x_2 - 4x_3 = 0 \]
\[ 2x_1 + 2x_2 = 1 \]
\[ 3x_1 + 3x_2 + 2x_3 = \frac{1}{2}. \]

Obtain the solution (by hand) of this system of equations by doing Gaussian elimination.

2. Write thereafter a program which implements Gaussian elimination (with pivoting) and solve the above system of linear equations. How many floating point operations are involved in the solution via Gaussian elimination without pivoting? Can you estimate the number of floating point operations with pivoting?

6.2. 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)
\[ L_{ii} = \left( A_{ii} - \sum_{k=0}^{i-1} L_{ik}^2 \right)^{1/2}. \] (6.51)

• within the loop over \( i \), introduce a new loop which goes from \( j = i + 1 \) to \( n - 1 \) and calculate
\[ L_{ji} = \frac{1}{L_{ij}} \left( A_{ij} - \sum_{k=0}^{i-1} L_{ik}L_{jk} \right). \] (6.52)

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. Write a function which performs the Cholesky decomposition. Test your program against the standard LU decomposition by using the matrix
\[ A = \begin{pmatrix} 6 & 3 & 2 \\ 3 & 2 & 1 \\ 2 & 1 & 1 \end{pmatrix} \] (6.53)

Finally, use the Cholesky method to solve
\[ 0.05x_1 + 0.07x_2 + 0.06x_3 + 0.05x_4 = 0.23 \\
0.07x_1 + 0.10x_2 + 0.08x_3 + 0.07x_4 = 0.32 \\
0.06x_1 + 0.08x_2 + 0.10x_3 + 0.09x_4 = 0.33 \\
0.05x_1 + 0.07x_2 + 0.09x_3 + 0.10x_4 = 0.31 \]

You can also use the LU codes for linear equations to check the results.

6.3. In this exercise we are going to solve the one-dimensional Poisson equation in terms of linear equations.
1. We are going to 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, 
\[ \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2} = -\frac{\rho(x,y,z)}{\varepsilon_0}, \]
whose solution we will discuss in chapter 10. The function \( \rho(x,y,z) \) is the charge density and \( \phi \) 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
\[ \frac{d^2 \phi}{dx^2} = -\frac{\rho(x)}{\varepsilon_0}, \]
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
\[ \frac{d^2 \phi}{dx^2} = \frac{I}{\varepsilon_0 A} \left( \frac{m_e}{2e} \right)^{1/2} \phi^{-1/2}. \]
The solution of the above equation which satisfies the boundary conditions is
\[ \phi = V_0 \left( \frac{x}{d} \right)^{4/3}, \]
with
\[ I = \frac{4}{9} \frac{\varepsilon_0 A}{d^2} \left( \frac{2e}{m_e} \right)^{1/2} V_0^{3/2}. \]
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 [37, 38], see Lieb for a newer and updated discussion [39]. 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 dimensionless 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

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

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

\[Av = \tilde{b},\]

where \( A \) is an \( n \times n \) tridiagonal matrix which we rewrite as

\[
A = \begin{pmatrix}
2 & -1 & 0 & \cdots & 0 \\
-1 & 2 & -1 & 0 & \cdots \\
0 & -1 & 2 & -1 & 0 & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & \cdots & -1 & 2 & -1 \\
0 & \cdots & 0 & -1 & 2 \\
\end{pmatrix}
\]

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.

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

\[
A = \begin{pmatrix}
b_1 & c_1 & 0 & \cdots & \cdots \\
b_2 & b_2 & c_2 & \cdots & \cdots \\
b_3 & b_3 & c_3 & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
b_{n-2} & b_{n-1} & c_{n-1} & \cdots & \cdots \\
b_n & b_n & & & \\
\end{pmatrix}
\begin{pmatrix}
v_1 \\
v_2 \\
v_3 \\
\cdots \\
v_{n-1} \\
v_n \\
\end{pmatrix}
= \begin{pmatrix}
\tilde{b}_1 \\
\tilde{b}_2 \\
\tilde{b}_3 \\
\cdots \\
\tilde{b}_{n-1} \\
\tilde{b}_n \\
\end{pmatrix}.
\]

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

\[a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i,\]
for \( i = 1, 2, \ldots, n \). 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 \( 10 \times 10, 100 \times 100 \) and \( 1000 \times 1000 \). 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 \( i = 1, \ldots, n \), by setting up

\[
\varepsilon_i = \log_{10} \left( \frac{|v_i - u_i|}{u_i} \right),
\]

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.

3. Compare your results with those from the LU decomposition codes for the matrix of size \( 1000 \times 1000 \). Use for example the unix function \texttt{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.

### 6.8.1 Solution

The program listed below encodes a possible solution to part b) 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 output 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 explicitly in the code. An obvious change is to define a separate function, allowing thereby for a generalization to other function \( f(x) \).

```cpp
/*
 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^2)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;
```
```cpp
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);
    }
    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,l> 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);
    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. 6.4 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. 6.5. It is therefore instructive to study the relative error, which we display in Table 6.2 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.
Fig. 6.4 Numerical solution obtained with \( n = 10 \) compared with the analytical solution.

Fig. 6.5 Numerical solution obtained with \( n = 10 \) compared with the analytical solution.

Table 6.2 \( \log_{10} \) values for the relative error and the step length \( h \) computed at \( x = 0.5 \).

| \( n \) | \( \log_{10}(h) \) | \( \varepsilon_i = \log_{10} \left( \frac{|v_i - u_i|}{u_i} \right) \) |
|---|---|---|
| 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 |
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.

6.4. Write your own code for performing the cubic spline interpolation using either Blitz++ or Armadillo. Alternatively you can use the vector-matrix class included in this text.

6.5. Write your own code for the LU decomposition using the same libraries as in the previous exercise. Find also the number of floating point operations.

6.6. Solve exercise 6.3 by writing a code which implements both the iterative Jacobi method and the Gauss-Seidel method. Study carefully the number of iterations needed to achieve the exact result.

6.7. Extend thereafter your code for the iterative Jacobi method to a parallel version and compare with the results from the previous exercise.

6.8. Write your own code for the Conjugate gradient method.

6.9. Write your own code for matrix-matrix multiplications using Strassen’s algorithm discussed in subsection 6.3.3.1 and compare the speed of your program with the matrix-matrix multiplication provided by the Armadillo library.