Linear Algebra

A

This appendix summarises the linear algebra that underpins the classes of vectors and matrices developed in this book. We present little more than the algorithms used: a reader interested in a deeper understanding of this theory should consult a textbook such as one of those listed in the Further Reading section at the end of this book.

A.1 Vectors and Matrices

For the purpose of this book, a vector is a one-dimensional array and a matrix is a two-dimensional array: it is—of course—possible to work only with matrices, with vectors having either only one column or only one row. For consistency with the classes of vectors and matrices developed, we treat vectors and matrices as separate entities in this discussion.

In this Appendix, we use mathematical rather than C++ notation for vectors and matrices.We will use italics to denote a scalar. Vectors will be denoted by lower case bold font letters. Individual entries of a vector will be denoted by italics indexed by subscripts. For example, $\mathbf{v}$ represents a vector, and the entry of $\mathbf{v}$ with index i is denoted by v_i. For consistency with C++ coding, we index the vectors and matrices in this Appendix so that the indices begin from 0. We assume that all vectors are column vectors: that is, a vector $\mathbf{v}$ of length N is the vector

$$\mathbf{v} = \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{N-1} \end{pmatrix}.$$

If a row vector is required, it is denoted using the transpose superscript, that is, $\mathbf{v}^\top$. Matrices will be denoted by upper case bold font letters, with italics indexed by

J. Pitt-Francis and J. Whiteley, *Guide to Scientific Computing in C++*, Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-319-73132-2

subscripts used to denote the entries of the matrix. The first index corresponds to the row number and the second index corresponds to the column number. Using this notation, if **A** is a matrix, then the entry of **A** that appears in the row with index i and the column with index j is denoted by A_{ij}. Where required for clarity, we will separate the indices by a comma, for example $A_{i+1,j-1}$.

A square matrix of size N has both N rows and N columns. The identity matrix of size N is a square matrix, denoted by $\mathbf{I}^{(\mathbf{N})}$, with entries given by

$$I_{ij}^{(N)} = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

A.1.1 Operations Between Vectors and Matrices

Linear combinations of vectors. Suppose $\mathbf{w} = \alpha\mathbf{u} + \beta\mathbf{v}$, where **u**, **v**, **w** are all vectors of length N, and α, β are scalars. The entries of **w** are given by

$$w_i = \alpha u_i + \beta v_i, \quad i = 0, 1, \ldots, N-1.$$

Linear combinations of matrices. Suppose $\mathbf{C} = \alpha\mathbf{A} + \beta\mathbf{B}$, where **A**, **B**, **C** are all matrices with M rows and N columns, and α, β are scalars. The entries of **C** are given by

$$C_{ij} = \alpha A_{ij} + \beta B_{ij}, \qquad i = 0, 1, \ldots, M-1, \quad j = 0, 1, \ldots, N-1.$$

Multiplication of a matrix by a vector. Suppose **A** is a matrix with M rows and N columns, and **u** is a vector of length N. If **v** = **Au**, then **v** is a vector of length M with entries given by

$$v_i = \sum_{j=0}^{N-1} A_{ij} u_j, \quad i = 0, 1, \ldots, M-1.$$

Similarly, if **s** is a vector of length M and $\mathbf{t}^\top = \mathbf{s}^\top\mathbf{A}$, then **t** is a vector of length N with entries given by

$$t_j = \sum_{i=0}^{M-1} s_i A_{ij}, \quad j = 0, 1, \ldots, N-1.$$

Multiplication of a matrix by a matrix. Suppose **A** is a matrix with L rows and M columns, and **B** is a matrix with M rows and N columns. If the matrix **C** satisfies **C** = **AB**, then **C** has L rows and N columns, and has entries given by

$$C_{ij} = \sum_{k=0}^{M-1} A_{ik} B_{kj}, \quad i = 0, 1, \ldots, L-1, \quad j = 0, 1, \ldots, N-1.$$

The transpose of a matrix. Suppose $\mathbf{A}$ is a matrix with M rows and N columns. If the matrix $\mathbf{B}$ satisfies $\mathbf{B} = \mathbf{A}^\top$, then $\mathbf{B}$ has N rows and M columns with entries given by

$$B_{ij} = A_{ji}, \quad i = 0, 1, \ldots, N-1, \quad j = 0, 1, \ldots, M-1.$$

A matrix $\mathbf{A}$ is said to be symmetric if $\mathbf{A} = \mathbf{A}^\top$.

A.1.2 The Scalar Product of Two Vectors

Suppose $\mathbf{v}$ and $\mathbf{w}$ are both vectors of length N. The *scalar product* between $\mathbf{v}$ and $\mathbf{w}$, denoted by $\mathbf{v} \cdot \mathbf{w}$, is given by

$$\mathbf{v} \cdot \mathbf{w} = \sum_{i=0}^{N-1} v_i w_i. \tag{A.1}$$

A.1.3 The Determinant and the Inverse of a Matrix

The simplest way to specify the determinant of a square matrix of general size is to use recursion.[1] Suppose $\mathbf{A}$ is a square matrix of size N. The determinant of $\mathbf{A}$, denoted by $\det(\mathbf{A})$, may be written

$$\begin{aligned}\det(\mathbf{A}) = A_{00} \det(\hat{\mathbf{A}}^{(00)}) - A_{01} \det(\hat{\mathbf{A}}^{(01)}) + A_{02} \det(\hat{\mathbf{A}}^{(02)}) - A_{03} \det(\hat{\mathbf{A}}^{(03)}) + \cdots \\ +(-1)^{N-1} A_{0,N-1} \det(\hat{\mathbf{A}}^{(0,N-1)}),\end{aligned}$$

where the square matrix $\hat{\mathbf{A}}^{(ij)}$, of size $N-1$, is the matrix $\mathbf{A}$ with row i and column j removed. This definition allows us to express the determinant of a square matrix of size N as a sum of determinants of square matrices of size $N-1$. This process may be repeated recursively until the determinant is expressed as a sum of determinants of square matrices of size 1. To complete this definition, we need to define the determinant of a square matrix of size 1: under these conditions $\det(\mathbf{A}) = A_{00}$. We leave it to the reader to verify that this definition is consistent with the commonly used expressions for the determinant of matrices of sizes 2 and 3.

[1]This recursion may be mapped directly into recursive functions (discussed in Sect. 5.8) when programming. However, it is generally more efficient to hard-code commonly used determinants for small matrices such as 2×2 and 3×3.

If the determinant of a square matrix $\mathbf{A}$ of size N is nonzero, then $\mathbf{A}$ is said to be *invertible*: a unique inverse matrix—denoted by $\mathbf{A}^{-1}$—exists, and satisfies

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}^{(N)}.$$

For the square matrix $\mathbf{A}$ of size 2 given by

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix},$$

then provided the determinant, given by $ad - bc$ is nonzero, $\mathbf{A}^{-1}$ exists and is given by

$$\mathbf{A}^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}.$$

A.1.4 Eigenvalues and Eigenvectors of a Matrix

Suppose $\mathbf{A}$ is a square matrix of size N. The scalar λ is said to be an eigenvalue of $\mathbf{A}$ if

$$\det(\mathbf{A} - \lambda \mathbf{I}^{(N)}) = 0.$$

If λ is an eigenvalue of A then a family of nonzero vectors[2] $\mathbf{v}$ that satisfy $\mathbf{A}\mathbf{v} = \lambda \mathbf{v}$ exists: each $\mathbf{v}$ in this family is then said to be an *eigenvector* corresponding to the eigenvalue λ.

A.1.5 Vector and Matrix Norms

Suppose $\mathbf{v}$ is a vector of length N. The p-norm of $\mathbf{v}$, denoted by $\|\mathbf{v}\|_p$, is given by

$$\|\mathbf{v}\|_p = \left(\sum_{i=0}^{N-1} |v_i|^p \right)^{1/p}. \tag{A.2}$$

[2]A vector $\mathbf{v}$ satisfies $\mathbf{v} = \mathbf{0}$ if, and only if, all entries of this vector take the value 0: $\mathbf{v}$ is then said to be a *zero vector*. If not, $\mathbf{v}$ is said to be a *nonzero vector*.

Taking the limit as $p \to \infty$, this definition yields

$$\|\mathbf{v}\|_\infty = \max_{i=0}^{N-1} |v_i| \,.$$

Of most use is the 2-norm: this is known as the Euclidean norm, and corresponds to the length of the line that represents a vector in two or three dimensions. Using Eq. (A.1), and Eq. (A.2) with $p = 2$, we see that we may write the 2-norm as

$$\|\mathbf{v}\|_2 = \sqrt{\sum_{i=0}^{N-1} v_i^2} = \sqrt{\mathbf{v} \cdot \mathbf{v}}.$$

The p-norm of a matrix $\mathbf{A}$, denoted by $\|\mathbf{A}\|_p$, is given (in terms of the vector p-norm) by

$$\|\mathbf{A}\|_p = \max_{\mathbf{v} \neq \mathbf{0}} \frac{\|\mathbf{A}\mathbf{v}\|_p}{\|\mathbf{v}\|_p}.$$

In common with vector norms, the most commonly used norm is the 2-norm. It can be shown that the eigenvalues of the matrix $\mathbf{A}^\top\mathbf{A}$ are all real and nonnegative. Let λ be the largest of these eigenvalues. Then $\|\mathbf{A}\|_2 = \sqrt{\lambda}$.

A.2 Systems of Linear Equations

Many algorithms in scientific computing require the solution of linear systems of the form $\mathbf{Ax} = \mathbf{b}$, where: (i) $\mathbf{A}$ is a square, invertible matrix of size N; (ii) the vectors $\mathbf{x}$, $\mathbf{b}$ are both of length N; (iii) $\mathbf{A}$, $\mathbf{b}$ are known; and (iv) $\mathbf{x}$ is to be calculated. Clearly $\mathbf{x}$ satisfies $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. However, calculating $\mathbf{A}^{-1}$ is extremely computationally expensive for large N and this approach is rarely used to solve systems of linear equations. Instead a plethora of techniques are available: we list three relatively simple methods below.

A.2.1 Gaussian Elimination

Readers may remember being taught how to solve two simultaneous linear equations for unknown values of x and y at school. When using this technique, the first step is to eliminate one of the variables resulting in a single linear equation for a single variable that can easily be solved. The value of this variable is then substituted back into one of the original equations to allow the value of the other variable to be calculated. Gaussian elimination is a systematic extension of this technique when solving a system of N linear equations for N unknowns. There are two versions of Gaussian elimination: with or without pivoting. We now describe both of these versions.

A.2.1.1 Gaussian Elimination Without Pivoting

The original system of equations may be written

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} & \dots & A_{0,N-1} \\ A_{10} & A_{11} & A_{12} & \dots & A_{1,N-1} \\ A_{20} & A_{21} & A_{22} & \dots & A_{2,N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{N-1,0} & A_{N-1,1} & A_{N-1,2} & \dots & A_{N-1,N-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{N-1} \end{pmatrix}.$$

Let us first assume that $A_{00} \neq 0$. This is a very restrictive assumption: in Sect. A.2.1.3 we introduce *pivoting*, which allows us to deal with the case $A_{00} = 0$. The assumption $A_{00} \neq 0$ allows us to eliminate x_0 from all but the first equation: this is achieved by subtracting a suitable multiple of the first equation, and results in the following system:

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} & \dots & A_{0,N-1} \\ 0 & A_{11}^{(1)} & A_{12}^{(1)} & \dots & A_{1,N-1}^{(1)} \\ 0 & A_{21}^{(1)} & A_{22}^{(1)} & \dots & A_{2,N-1}^{(1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & A_{N-1,1}^{(1)} & A_{N-1,2}^{(1)} & \dots & A_{N-1,N-1}^{(1)} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1^{(1)} \\ b_2^{(1)} \\ \vdots \\ b_{N-1}^{(1)} \end{pmatrix},$$

where:

$$\begin{aligned} M_{i0} &= A_{i0}/A_{00}, \quad i = 1, 2, \dots, N-1, \text{ using the assumption that } A_{00} \neq 0, \\ A_{ij}^{(1)} &= A_{ij} - M_{i0}A_{0j}, \quad i, j = 1, 2, \dots, N-1, \\ b_i^{(1)} &= b_i - M_{i0}b_0, \quad i = 1, 2, \dots, N-1. \end{aligned}$$

Assuming now that $A_{11}^{(1)} \neq 0$, we may repeat this process to eliminate x_1 from all but the first two equations:

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} & \dots & A_{0,N-1} \\ 0 & A_{11}^{(1)} & A_{12}^{(1)} & \dots & A_{1,N-1}^{(1)} \\ 0 & 0 & A_{22}^{(2)} & \dots & A_{2,N-1}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & A_{N-1,2}^{(2)} & \dots & A_{N-1,N-1}^{(2)} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_{N-1}^{(2)} \end{pmatrix},$$

where:

$$\begin{aligned} M_{i1} &= A_{i1}^{(1)}/A_{11}^{(1)}, \quad i = 2, 3, \dots, N-1, \text{ using the assumption that } A_{11}^{(1)} \neq 0, \\ A_{ij}^{(2)} &= A_{ij}^{(1)} - M_{i1}A_{1j}^{(1)}, \quad i, j = 2, 3, \dots, N-1, \\ b_i^{(2)} &= b_i^{(1)} - M_{i1}b_1^{(1)}, \quad i = 2, 3, \dots, N-1. \end{aligned}$$

Providing that at all steps we have $A_{kk}^{(k)} \neq 0, k = 0, 1, \ldots, N-1$, we may continue in this fashion until we have generated an upper triangular matrix $\mathbf{A}^{(N-1)}$:

$$\mathbf{A}^{(N-1)}\mathbf{x} = \begin{pmatrix} A_{00} & A_{01} & A_{02} & \ldots & A_{0,N-1} \\ 0 & A_{11}^{(1)} & A_{12}^{(1)} & \ldots & A_{1,N-1}^{(1)} \\ 0 & 0 & A_{22}^{(2)} & \ldots & A_{2,N-1}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & A_{N-1,N-1}^{(N-1)} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_{N-1}^{(N-1)} \end{pmatrix} = \mathbf{b}^{(N-1)}.$$

Solving this upper triangular system is a straightforward task: we start with the last equation in this system and work our way backwards. The first two steps in this procedure are

$$\begin{aligned} x_{N-1} &= b_{N-1}^{(N-1)} / A_{N-1,N-1}^{(N-1)}, \\ x_{N-2} &= \frac{1}{A_{N-2,N-2}^{(N-2)}} \left(b_{N-2}^{(N-2)} - A_{N-2,N-1}^{(N-2)} x_{N-1} \right). \end{aligned}$$

A general formula exists for calculating $x_k, k = 0, 1, 2, \ldots, N-1$. Assuming that we have already calculated $x_{k+1}, x_{k+2}, \ldots, x_{N-1}$, we may calculate x_k by

$$x_k = \frac{1}{A_{k,k}^{(k)}} \left(b_k^{(k)} - \sum_{i=k+1}^{N-1} A_{k,i}^{(k)} x_i \right). \tag{A.3}$$

This completes the description of the Gaussian elimination algorithm without pivoting. A very important point to note is that there is no need to store all the matrices generated during this algorithm: only the most recently generated version is required, and all earlier matrices may be discarded.

A.2.1.2 LU Decomposition

The Gaussian elimination process described above may be used to factorise **A** as the product of a lower triangular matrix **L** and an upper triangular matrix **U**, that is, **A** = **LU**. Defining the matrices $\mathbf{M}_0, \mathbf{M}_1, \ldots$ by

$$\mathbf{M}_0 = \begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ -M_{10} & 1 & 0 & \ldots & 0 \\ -M_{20} & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -M_{N-1,0} & 0 & 0 & \ldots & 1 \end{pmatrix},$$

$$\mathbf{M}_1 = \begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ 0 & -M_{21} & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & -M_{N-1,1} & 0 & \ldots & 1 \end{pmatrix}, \ldots,$$

we may write

$$\mathbf{A}^{(N-1)} = \mathbf{M}_{N-1}\mathbf{M}_{N-2}\cdots\mathbf{M}_1\mathbf{M}_0\mathbf{A},$$

or, equivalently,

$$\mathbf{A} = \mathbf{M}_0^{-1}\mathbf{M}_1^{-1}\cdots\mathbf{M}_{N-2}^{-1}\mathbf{M}_{N-1}^{-1}\mathbf{A}^{(N-1)}.$$

We first note that the inverses of the matrices $\mathbf{M}_0, \mathbf{M}_1, \ldots$, are simply

$$\mathbf{M}_0^{-1} = \begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ M_{10} & 1 & 0 & \ldots & 0 \\ M_{20} & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M_{N-1,0} & 0 & 0 & \ldots & 1 \end{pmatrix},$$

$$\mathbf{M}_1^{-1} = \begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ 0 & M_{21} & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & M_{N-1,1} & 0 & \ldots & 1 \end{pmatrix}, \ldots.$$

These matrices are all lower triangular. It is trivial to prove that the product of lower triangular matrices is also lower triangular. Writing

$$\begin{aligned} \mathbf{L} &= \mathbf{M}_0^{-1}\mathbf{M}_1^{-1}\cdots\mathbf{M}_{N-2}^{-1}\mathbf{M}_{N-1}^{-1}, \\ \mathbf{U} &= \mathbf{A}^{(N-1)}, \end{aligned}$$

we see that we have $\mathbf{A} = \mathbf{LU}$ with $\mathbf{L}$ a lower triangular matrix and $\mathbf{U}$ an upper triangular matrix. An explicit representation of $\mathbf{L}$ exists: direct calculation may be used to verify that

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ M_{10} & 1 & 0 & \ldots & 0 \\ M_{20} & M_{21} & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M_{N-1,0} & M_{N-1,1} & M_{N-1,2} & \ldots & 1 \end{pmatrix}.$$

A.2.1.3 Gaussian Elimination with Pivoting

The Gaussian elimination technique described above required that $A_{kk}^{(k)} \neq 0$ at each step. Clearly this algorithm would fail for a nonsingular matrix such as

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 2 \\ 0 & 5 & 1 \end{pmatrix},$$

where

$$\mathbf{A}^{(1)} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 5 & 1 \end{pmatrix},$$

and so $A_{11}^{(1)} = 0$, violating one of the assumptions made in Sect. A.2.1.1. We can, however, proceed further: in this case we would simply interchange the last two rows of both $\mathbf{A}^{(1)}$ and $\mathbf{b}^{(1)}$. This is known as *pivoting*.

Even if $|A_{kk}^{(k)}|$ is not zero it may be advisable to use pivoting. In Eq. (A.3) we see that calculating the value of x_k requires us to divide by $A_{kk}^{(k)}$. If $|A_{kk}^{(k)}|$ is small then the division by a small number may introduce numerical errors in the calculation of x_k. To avoid both of these problems, we recommend pivoting at each step: when constructing $\mathbf{A}^{(k)}$, find the row n with the largest absolute value of $A_{nk}^{(k)}$, $n = k, k+1, \ldots, N-1$, and then interchange row k and row n. It is relatively simple to include this in our Gaussian elimination algorithm: at step k we are working with the linear system

$$\mathbf{A}^{(k)}\mathbf{x} = \mathbf{b}^{(k)}.$$

To interchange rows k and n in this system of equations, we simply multiply both sides of this equation by the matrix $\mathbf{P}^{(kn)}$:

$$\mathbf{P}^{(kn)}\mathbf{A}^{k}\mathbf{x} = \mathbf{P}^{(kn)}\mathbf{b}^{k},$$

where $\mathbf{P}^{(kn)}$ is a square matrix of size N with entries given by

$$\mathbf{P}_{ij}^{(kn)} = \begin{cases} 1, & i = j, \ \ i, j \neq k, \ \ i, j \neq n, \\ 1, & i = k, \ \ j = n, \\ 1, & i = n, \ \ j = k, \\ 0, & \text{otherwise.} \end{cases}$$

For example, if we wanted to interchange the row with index 2 and the row with index 4 in a square matrix of size 5, then the matrix $\mathbf{P}^{(24)}$ would be given by

$$\mathbf{P}^{(24)} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

The key point to note when modifying the LU-factorisation algorithm described in Sect. A.2.1.1 to take account of pivoting is that Gaussian elimination with pivoting would give exactly the same results if all the rows were interchanged first, and then Gaussian elimination with no pivoting were carried out. Denoting the product of all the matrices representing row interchanges by $\mathbf{P}$, we see that the LU-decomposition algorithm now reduces to a factorisation of the matrix $\mathbf{PA}$: that is, forming

$$\mathbf{LU} = \mathbf{PA}.$$

A.2.2 The Thomas Algorithm

The Thomas algorithm may be used for matrices with a specific structure. Suppose our matrix $\mathbf{A}$ has structure

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ -p_1 & q_1 & -r_1 & 0 & \dots & 0 & 0 & 0 \\ 0 & -p_2 & q_2 & -r_2 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & -p_{N-2} & q_{N-2} & -r_{N-2} \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 1 \end{pmatrix},$$

where the entries of $\mathbf{A}$ satisfy

$$p_i > 0, \qquad q_i > 0, \qquad r_i > 0, \qquad q_i > p_i + r_i, \qquad i = 1, 2, \dots, N-2.$$

This condition is satisfied, for example, for an implicit finite difference discretisation of the heat equation in one spatial dimension with Dirichlet boundary conditions at both ends of the spatial domain. Defining

$$\begin{aligned} &e_0 = 0, \qquad f_0 = b_0, \\ &e_i = \tfrac{r_i}{q_i - p_i e_{i-1}}, \qquad f_i = \tfrac{b_i + p_i f_{i-1}}{q_i - p_i e_{i-1}}, \qquad i = 1, 2, \dots, N-2, \end{aligned}$$

then the linear system may be solved using the explicit recurrence relation

$$\begin{aligned} x_{N-1} &= b_{N-1}, \\ x_i &= \frac{r_i}{q_i - p_i e_{i-1}} x_{i+1} + \frac{b_i + p_i f_{i-1}}{q_i - p_i e_{i-1}}, \quad i = N-2, N-3, \dots, 1, \\ x_0 &= b_0. \end{aligned}$$

A.2.3 The Conjugate Gradient Method

The matrices arising in many scientific computing applications—for example, finite element, finite difference and finite volume discretisations of partial differential equations—often have a large number of rows and columns, but very few nonzero elements in each row of the matrix. Such matrices are termed *sparse matrices*.

It is often the case that storing every element of a sparse matrix would exceed the memory limitations of a computational architecture, but storing only the nonzeros of this matrix is possible within the constraints of available memory. This poses a logistical challenge for the solution of linear systems described by this matrix: the LU-factorisation of a sparse matrix described in Sect. A.2.1.1 does not result in sparse matrices L and U, and so these matrices will suffer from the memory limitations described earlier. To circumvent this problem, iterative techniques may be used for the solution of sparse linear systems, where successive iterates of the solution of the linear system $\mathbf{x}_k$, $k = 1, 2, \ldots$ are generated until $\|\mathbf{b} - \mathbf{A}\mathbf{x}_k\| < \varepsilon$ for some user-specified tolerance ε. This branch of numerical linear algebra is a large subject in its own right and we only touch briefly upon it here, giving one algorithm for a very specific class of matrices, namely symmetric, positive definite matrices.

Algorithm 1 Conjugate gradient method for solving $\mathbf{A}\mathbf{x} = \mathbf{b}$

Require: Symmetric, positive definite matrix $\mathbf{A}$, specified vector $\mathbf{b}$, initial guess $\mathbf{x}_0$ (or set $\mathbf{x}_0 = \mathbf{0}$), tolerance ε.

```
1: $k = 0$, $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}_k$, $\mathbf{p} = \mathbf{0}$, $\beta = 0$
2: while $\|\mathbf{r}\| \geq \varepsilon$ do
3:   if $k > 0$ then
4:       $\beta = \frac{\mathbf{r}^\top \mathbf{r}}{\mathbf{r}_{\text{prev}}^\top \mathbf{r}_{\text{prev}}}$
5:   end if
6:   $\mathbf{p} = \mathbf{r} + \beta\mathbf{p}$
7:   $\alpha = \frac{\mathbf{r}^\top \mathbf{r}}{\mathbf{p}^\top A\mathbf{p}}$
8:   $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha\mathbf{p}$
9:   $\mathbf{r}_{\text{prev}} = \mathbf{r}$
10:  $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}_{k+1}$
11:  $k = k + 1$
12: end while
13: $\mathbf{x} = \mathbf{x}_k$
```

A matrix $\mathbf{A}$ is said to be *positive definite* if, and only if, for all vectors $\mathbf{x}$ of the correct size the following two conditions are met:

$$\begin{aligned} \mathbf{x}^\top \mathbf{A}\mathbf{x} &\geq 0, \quad \text{and} \\ \mathbf{x}^\top \mathbf{A}\mathbf{x} &= 0, \quad \text{only if } \mathbf{x} = \mathbf{0}. \end{aligned}$$

If a matrix $\mathbf{A}$ is positive definite and symmetric, then we may solve the linear system using the *conjugate gradient method*, given by Algorithm 1.

Other Programming Constructs You Might Meet

B

Below we briefly describe some programming constructs that other programmers may include in their C++ code. Many of these are constructs that were originally designed for the C programming language. As C++ was developed from C, much of the C language is legal C++, although the modifications developed for the C++ language are generally superior.

B.1 C Style Output

We devoted the whole of Chap. 3 to describing the C++ machinery for input and output. To explain the corresponding machinery in C would require a similar amount of space, and so we only touch upon C style output here, limiting ourselves to describing output to the console. Nevertheless, this should give the flavour of C style output commands, allowing the reader to at least recognise them should they see them.

In the code below, we show how to use C style output to print a double precision floating point variable to the screen in both normal and scientific notation, and how to print an integer to the screen. C style output requires the whole of the output to be enclosed within double quotation marks. When a variable is to be printed it is represented by `%f` for a double precision floating point variable, `%i` for an integer variable, and `%e` for a double precision floating point variable in scientific notation. Finally, the variables to be printed are included in an ordered list at the end of the statement. Note that the included file for C style printing is `<stdio.h>`—standard input and output which provides basic functionality similar to `<iostream>` in C++.

```
#include <stdio.h>

int main(int argc, char* argv[])
{
   double x = 105.0;
   int j = 500;
   printf("x = %f and j = %i\n", x, j);
   printf("In scientific notation, x = %e\n", x);
   return 0;
}
```

Other C variations on `printf` which you might meet are `fprintf` for printing to file, in which the first argument is a file pointer of type `FILE*` and `sprintf` for printing to a string.

B.2 C Style Dynamic Memory Allocation

In Sect. 4.2 we explained how the C++ keywords `new` and `delete` could be used to allocate memory dynamically for arrays, and then free the memory when it was no longer needed. C also allows this, through the use of `malloc` ("memory allocate") and `free`. As with C style output above, we only touch briefly on the use of these functions to allow the reader to recognise them should they come across them. In the code below, we declare a pointer to a double precision variable, `vector`, in line 6. In line 7, we then use the `malloc` function to allocate memory for 100 entries of the array `vector`, all of the same size as a double precision floating point variable. In lines 8–13, we use these entries in the same way as a C++ array. Finally, in line 14, we free the memory allocated to this array through the use of the C function called `free`.

```
 1  #include <iostream>
 2  #include <cstdlib>
 3
 4  int main(int argc, char* argv[])
 5  {
 6     double* vector;
 7     vector = ((double*)(malloc(100*sizeof(double))));
 8     vector[0] = 1.0;
 9     vector[90] = 3.0;
10     std::cout << "Entry of vector with index 0 = "
11               << vector[0] << "\n";
12     std::cout << "Entry of vector with index 90 = "
13               << vector[90] << "\n";
14     free(vector);
15     return 0;
16  }
```

B.3 Ternary ?: Operator

In Sect. 2.1.3 we saw that the keywords `if` and `else` could be used to execute one set of statements if a condition was met, and a different set of instructions if the condition is not met, as in the code fragment below.

```
   double a, b, x;
   if (a > b)
   {
      x = 100.0;
   }
   else
   {
      // a <= b
      x = 0.0;
   }
```

The ternary[3] ?: operator has identical effect to the `if-else` statements above: the code above may be written identically as

```
   double a, b, x;
   x = (a > b) ? 100.0 : 0.0;
```

Although the code written above is shorter than the original `if–else` statements we do not recommend it. The use of `if` and `else` makes the code much more readable, especially by anyone who is not an expert in C++ programming.

B.4 Using Namespace

You may find it tedious to have to write `std::` before `cout` and other functionality of the C++ language. There is a way around this—we may use the `using` statement once in the code as shown below.

```
#include <iostream>

using namespace std;
int main(int argc, char* argv[])
{
   string city = "Cambridge";
   cout << city << "\n";
   return 0;
}
```

[3] A ternary operator has three inputs.

At first sight, the code above may appear to make a programmer's life a little easier. Both `string` and `cout` have been used here without being preceded by the slightly clunky `std::`. This approach does, however, introduce a subtle problem. Suppose we declared a variable called "`vector`". It would then be unclear whether an instance of the word "`vector`" is referring to this variable, or the STL vector introduced in Chap. 8, which the `using` statement now allows us to refer to as `vector` rather than `std::vector`. As such, we do not recommend use of the `using` keyword.

B.5 Structures

A *structure* is a collection of variables that are combined together. Structures can be thought of as very simple classes, but without the ability to declare functions, access privileges, or any other properties of classes other than variables. An example of a structure is shown below. Note how the variables are accessed in exactly the same way as classes (using "." for a member or "`->`" to access a member by de-referencing a pointer).

```
#include <iostream>

struct ModelParameters
{
   double viscosity;
   double density;
   int numberOfDimensions;
};

int main(int argc, char* argv[])
{
   ModelParameters example1;
   example1.viscosity = 1.0e-4;
   example1.density = 1.0;
   ModelParameters* p_eg1 = &example1;
   p_eg1->numberOfDimensions = 3;

   std::cout << "Density is " << example1.density << "\n";

   return 0;
}
```

B.6 Multiple Inheritance

As mentioned in Sect. 7.1 C++, unlike many other object-oriented languages, allows *multiple inheritance* in which a derived class can be derived from multiple base classes. That is, classes may have more than one parent.

Suppose we require a class of matrices so that we can calculate the determinant of given matrices, calculate the eigenvalues of these matrices, and calculate the norm of these matrices. One colleague may have a class of matrices, `MatrixDet`, that calculates the determinant of a matrix, but doesn't have the functionality for calculating the eigenvalues or the norm of a matrix. Another colleague may have a class of matrices, `MatrixEigsNorm`, that does allow us to calculate the eigenvalues and norm of a matrix, but not the determinant. The functionality required is therefore all available, but not in the same class. It would therefore be convenient to merge the two classes to create a new class that contains all the functionality required. This is possible through *multiple inheritance*. Below we show how to perform multiple inheritance to generate a new class `MatrixCombined`.

```cpp
#include "MatrixDet.hpp"
#include "MatrixEigsNorm.hpp"

class MatrixCombined: public MatrixDet,
                      public MatrixEigsNorm
{
   // Body of class
};
```

If the class `MatrixDet` has no member with the same name as a member of the class `MatrixEigsNorm` then multiple inheritance is an ideal solution to this problem. Suppose both classes have a method called `ZeroEntries`. Provided this member is made a virtual function in both the class `MatrixDet` and the class `MatrixEigsNorm` we may prevent ambiguity through either defining a new function in the class `MatrixCombined`, or by explicitly identifying which function is to be used in the calling code. For example:

```cpp
   MatrixCombined mat;
   // use method ZeroEntries from the class MatrixDet
   mat.MatrixDet::ZeroEntries();
```

B.7 Class Initialisers

In many cases, the constructor of a class is a simple piece of code involving a list of assignments. For example, the default constructor for the `Book` class in

Sect. 6.2.7 set all the string fields to "unspecified" and the default constructor of the `ComplexNumber` class in Sect. 6.4 set the real and imaginary components to zero.

```
#include "ComplexNumber.hpp"
// Override default constructor
// Set real and imaginary parts to zero
ComplexNumber::ComplexNumber()
{
   mRealPart = 0.0;
   mImaginaryPart = 0.0;
}
```

In cases where a constructor makes assignments it is more efficient to use C++ *initialisers*. These are comma-separated lists of member variables and values which appear after the constructor's signature (and a colon) but before the main body of the constructor code. Compilers for C++ are able to optimise a list of initialised values more completely than a block of code containing assignment statements. It must be noted that some compilers insist that the initialisers are ordered exactly as they appear in the definition of the class. An example constructor for the class of complex numbers given in Sect. 6.4 that uses class initialisers is shown below.

```
#include "ComplexNumber.hpp"
// Override default constructor
// Initialize real and imaginary parts as zero
ComplexNumber::ComplexNumber() :
            mRealPart(0.0),
            mImaginaryPart(0.0)
{
   // possibly have more code in body
}
```

Solutions to Exercises C

C.1 Matrix and Linear System Classes

The code below is example solutions for the `Matrix` and `LinearSystem` classes developed in the Exercises at the end of Chap. 10.

Listing C.1 `Matrix.hpp`

```
#ifndef MATRIXHEADERDEF
#define MATRIXHEADERDEF
#include "Vector.hpp"

class Matrix
{
private:
   double** mData; // entries of matrix
   int mNumRows, mNumCols; // dimensions
public:
   Matrix(const Matrix& otherMatrix);
   Matrix(int numRows, int numCols);
   ~Matrix();
   int GetNumberOfRows() const;
   int GetNumberOfColumns() const;
   double& operator()(int i, int j); //1-based indexing
   //overloaded assignment operator
   Matrix& operator=(const Matrix& otherMatrix);
   Matrix operator+() const; // unary +
   Matrix operator-() const; // unary -
   Matrix operator+(const Matrix& m1) const; // binary +
   Matrix operator-(const Matrix& m1) const; // binary -
   // scalar multiplication
```

https://doi.org/10.1007/978-3-319-73132-2

```
   Matrix operator*(double a) const;
   double CalculateDeterminant() const;
   // declare vector multiplication friendship
   friend Vector operator*(const Matrix& m,
                           const Vector& v);
   friend Vector operator*(const Vector& v,
                           const Matrix& m);
};
// prototype signatures for friend operators
Vector operator*(const Matrix& m, const Vector& v);
Vector operator*(const Vector& v, const Matrix& m);

#endif
```

Listing C.2 Matrix.cpp

```
#include <cmath>
#include <cassert>
#include "Matrix.hpp"
#include "Vector.hpp"


// Copy constructor
// Allocate memory for new matrix, and copy
// entries into this matrix
Matrix::Matrix(const Matrix& otherMatrix)
{
   mNumRows = otherMatrix.mNumRows;
   mNumCols = otherMatrix.mNumCols;
   mData = new double* [mNumRows];
   for (int i=0; i<mNumRows; i++)
   {
      mData[i] = new double [mNumCols];
   }
   for (int i=0; i<mNumRows; i++)
   {
      for (int j=0; j<mNumCols; j++)
      {
         mData[i][j] = otherMatrix.mData[i][j];
      }
   }
}

// Constructor for vector of a given length
// Allocates memory, and initialises entries
// to zero
Matrix::Matrix(int numRows, int numCols)
{
   assert(numRows > 0);
   assert(numCols > 0);
   mNumRows = numRows;
```

```
    mNumCols = numCols;
    mData = new double* [mNumRows];
    for (int i=0; i<mNumRows; i++)
    {
       mData[i] = new double [mNumCols];
    }
    for (int i=0; i<mNumRows; i++)
    {
       for (int j=0; j<mNumCols; j++)
       {
          mData[i][j] = 0.0;
       }
    }
}

// Overwritten destructor to correctly free memory
Matrix::~Matrix()
{
    for (int i=0; i<mNumRows; i++)
    {
       delete[] mData[i];
    }
    delete[] mData;
}

// Method to get number of rows of matrix
int Matrix::GetNumberOfRows() const
{
    return mNumRows;
}

// Method to get number of columns of matrix
int Matrix::GetNumberOfColumns() const
{
    return mNumCols;
}

// Overloading the round brackets
// Note that this uses 'one-based' indexing,
// and a check on the validity of the index
double& Matrix::operator()(int i, int j)
{
    assert(i > 0);
    assert(i < mNumRows+1);
    assert(j > 0);
    assert(j < mNumCols+1);
    return mData[i-1][j-1];
}

// Overloading the assignment operator
Matrix& Matrix::operator=(const Matrix& otherMatrix)
```

```
{
   assert(mNumRows = otherMatrix.mNumRows);
   assert(mNumCols = otherMatrix.mNumCols);

   for (int i=0; i<mNumRows; i++)
   {
      for (int j=0; j<mNumCols; j++)
      {
         mData[i][j] = otherMatrix.mData[i][j];
      }
   }
   return *this;
}

// Overloading the unary + operator
Matrix Matrix::operator+() const
{
   Matrix mat(mNumRows, mNumCols);
   for (int i=0; i<mNumRows; i++)
   {
      for (int j=0; j<mNumCols; j++)
      {
         mat(i+1,j+1) = mData[i][j];
      }
   }
   return mat;
}

// Overloading the unary - operator
Matrix Matrix::operator-() const
{
   Matrix mat(mNumRows, mNumCols);
   for (int i=0; i<mNumRows; i++)
   {
      for (int j=0; j<mNumCols; j++)
      {
         mat(i+1,j+1) = -mData[i][j];
      }
   }
   return mat;
}

// Overloading the binary + operator
Matrix Matrix::operator+(const Matrix& m1) const
{
   assert(mNumRows == m1.mNumRows);
   assert(mNumCols == m1.mNumCols);
   Matrix mat(mNumRows, mNumCols);
   for (int i=0; i<mNumRows; i++)
   {
      for (int j=0; j<mNumCols; j++)
```

```
        {
           mat(i+1,j+1) = mData[i][j] + m1.mData[i][j];
        }
    }
    return mat;
}

// Overloading the binary - operator
Matrix Matrix::operator-(const Matrix& m1) const
{
    assert(mNumRows == m1.mNumRows);
    assert(mNumCols == m1.mNumCols);
    Matrix mat(mNumRows, mNumCols);
    for (int i=0; i<mNumRows; i++)
    {
        for (int j=0; j<mNumCols; j++)
        {
           mat(i+1,j+1) = mData[i][j] - m1.mData[i][j];
        }
    }
    return mat;
}

// Overloading scalar multiplication
Matrix Matrix::operator*(double a) const
{
    Matrix mat(mNumRows, mNumCols);
    for (int i=0; i<mNumRows; i++)
    {
        for (int j=0; j<mNumCols; j++)
        {
           mat(i+1,j+1) = a*mData[i][j];
        }
    }
    return mat;
}

// Overloading matrix multiplied by a vector
Vector operator*(const Matrix& m, const Vector& v)
{
    int original_vector_size = v.GetSize();
    assert(m.GetNumberOfColumns() == original_vector_size);
    int new_vector_length = m.GetNumberOfRows();
    Vector new_vector(new_vector_length);

    for (int i=0; i<new_vector_length; i++)
    {
        for (int j=0; j<original_vector_size; j++)
        {
           new_vector[i] += m.mData[i][j]*v.Read(j);
        }
```

```
   }

   return new_vector;
}

// Overloading vector multiplied by a matrix
Vector operator*(const Vector& v, const Matrix& m)
{
   int original_vector_size = v.GetSize();
   assert(m.GetNumberOfRows() == original_vector_size);
   int new_vector_length = m.GetNumberOfColumns();
   Vector new_vector(new_vector_length);

   for (int i=0; i<new_vector_length; i++)
   {
      for (int j=0; j<original_vector_size; j++)
      {
         new_vector[i] += v.Read(j)*m.mData[j][i];
      }
   }

   return new_vector;
}

// Calculate determinant of square matrix recursively
double Matrix::CalculateDeterminant() const
{
   assert(mNumRows == mNumCols);
   double determinant = 0.0;

   if (mNumRows == 1)
   {
      determinant = mData[0][0];
   }
   else
   {
      // More than one entry of matrix
      for (int i_outer=0; i_outer<mNumRows; i_outer++)
      {
         Matrix sub_matrix(mNumRows-1,
                              mNumRows-1);
         for (int i=0; i<mNumRows-1; i++)
         {
            for (int j=0; j<i_outer; j++)
            {
               sub_matrix(i+1,j+1) = mData[i+1][j];
            }
            for (int j=i_outer; j<mNumRows-1; j++)
            {
               sub_matrix(i+1,j+1) = mData[i+1][j+1];
            }
            }
            double sub_matrix_determinant =
                     sub_matrix.CalculateDeterminant();

            determinant += pow(-1.0, i_outer)*
                     mData[0][i_outer]*sub_matrix_determinant;
        }
    }
    return determinant;
}
```

Listing C.3 LinearSystem.hpp

```
#ifndef LINEARSYSTEMHEADERDEF
#define LINEARSYSTEMHEADERDEF
#include "Vector.hpp"
#include "Matrix.hpp"

class LinearSystem
{
protected://private or protected for Exercise 10.5
   int mSize; // size of linear system
   Matrix* mpA;  // matrix for linear system
   Vector* mpb;  // vector for linear system

   // Only allow constructor that specifies matrix/vector
   // to be used.  Copy constructor is private or protected.
   LinearSystem(const LinearSystem& otherLinearSystem){};
public:
   LinearSystem(const Matrix& A, const Vector& b);

   // destructor frees memory allocated
   ~LinearSystem();

   // Method for solving system
   virtual Vector Solve();
};

#endif
```

Listing C.4 LinearSystem.cpp

```cpp
#include <cmath>
#include <cassert>
#include "LinearSystem.hpp"
#include "Matrix.hpp"
#include "Vector.hpp"

// Copy matrix and vector so that original matrix and vector
// specified are unchanged by Gaussian elimination
LinearSystem::LinearSystem(const Matrix& A, const Vector& b)
{
   // check matrix and vector are of compatible sizes
   int local_size = A.GetNumberOfRows();
   assert(A.GetNumberOfColumns() == local_size);
   assert(b.GetSize() == local_size);

   // set variables for linear system
   mSize = local_size;
   mpA = new Matrix(A);
   mpb = new Vector(b);
}

// Destructor to free memory
LinearSystem::~LinearSystem()
{
   delete mpA;
   delete mpb;
}

// Solve linear system using Gaussian elimination
// This method changes the content of the matrix mpA
Vector LinearSystem::Solve()
{
   Vector m(mSize); //See description in Appendix A
   Vector solution(mSize);

   // We introduce references to make the syntax readable
   Matrix& rA = *mpA;
   Vector& rb = *mpb;

   // forward sweep of Gaussian elimination
   for (int k=0; k<mSize-1; k++)
   {
      // see if pivoting is necessary
      double max = 0.0;
      int row = -1;
      for (int i=k; i<mSize; i++)
      {
         if (fabs(rA(i+1,k+1)) > max)
         {
            row = i;
            max=fabs(rA(i+1,k+1));
         }
      }
      assert(row >= 0);

      // pivot if necessary
      if (row != k)
      {
         // swap matrix rows k+1 with row+1
         for (int i=0; i<mSize; i++)
         {
            double temp = rA(k+1,i+1);
            rA(k+1,i+1) = rA(row+1,i+1);
            rA(row+1,i+1) = temp;
         }
         // swap vector entries k+1 with row+1
         double temp = rb(k+1);
         rb(k+1) = rb(row+1);
         rb(row+1) = temp;
      }

      // create zeros in lower part of column k
      for (int i=k+1; i<mSize; i++)
      {
         m(i+1) = rA(i+1,k+1)/rA(k+1,k+1);
         for (int j=k; j<mSize; j++)
         {
            rA(i+1,j+1) -= rA(k+1,j+1)*m(i+1);
         }
         rb(i+1) -= rb(k+1)*m(i+1);
      }
   }

   // back substitution
   for (int i=mSize-1; i>-1; i--)
   {
      solution(i+1) = rb(i+1);
      for (int j=i+1; j<mSize; j++)
      {
         solution(i+1) -= rA(i+1,j+1)*solution(j+1);
      }
      solution(i+1) /= rA(i+1,i+1);
   }

   return solution;
}
```

Listing C.5 LinearSystemTestSuite.hpp

```
#include <cmath>
#include <cxxtest/TestSuite.h>
#include "Vector.hpp"
#include "Matrix.hpp"
#include "LinearSystem.hpp"

// An outline solution for Exercise 10.4
class LinearSystemTestSuite : public CxxTest::TestSuite
{
public:
   // Test constructors (using norm etc.)
   void TestDefaultConstructors(void)
   {
      Matrix squ(5,5);
      TS_ASSERT_DELTA(squ.CalculateDeterminant(), 0.0, 1e-8);
      Matrix nonsquare(7,13);
      TS_ASSERT_EQUALS(nonsquare.GetNumberOfRows(),     7);
      TS_ASSERT_EQUALS(nonsquare.GetNumberOfColumns(), 13);
      Vector vec(25);
      TS_ASSERT_DELTA(vec.CalculateNorm(), 0.0, 1.0e-8);
      TS_ASSERT_EQUALS(vec.GetSize(), 25);
      TS_ASSERT_EQUALS(length(vec), 25);
   }
   // Empty test
   void TestSomeExceptions(void)
   {
      // Our code uses assertions for error checking.
      // If you use Exception then test:
      //       Matrix a(3,3); Matrix b(4,4);
      //       TS_ASSERT_THROWS_ANYTHING(a+b);
   }
   // Test with cond(a) ~= 1e7
   void TestLargeConditionNumber(void)
   {
      Matrix a(3,3);  Vector b(3); Vector x(3);
      a(1,1) = 1;   a(1,2) = 0;   a(1,3) = 1e7;
      a(2,1) = 1;   a(2,2) =-1;   a(2,3) = 0;
      a(3,1) = 1;   a(3,2) = 0;   a(3,3) = 1;
      b(1) = 1e7+1; b(2) = 0;     b(3) = 2;
      double det = a.CalculateDeterminant();
      TS_ASSERT_DELTA(det, 1.0e7-1.0, 1e-8);
      LinearSystem ls(a, b);
      x = ls.Solve();
      for (int i=1; i<=3; i++)
      {
         TS_ASSERT_DELTA( x(i), 1.0, 1e-8);
      }
   }
   // Gaussian Elimination without pivoting would fail:
   void TestZeroPivot(void)
   {
      Matrix a(3,3);  Vector b(3); Vector x(3);
      a(1,1) = 0;   a(1,2) = 1;   a(1,3) = 1;
      a(2,1) = 1;   a(2,2) =-1;   a(2,3) = 0;
      a(3,1) = 1;   a(3,2) = 1;   a(3,3) = 1;
      b(1) = 2;     b(2) = 0;     b(3) = 3;
      TS_ASSERT_DELTA( a.CalculateDeterminant(), 1.0, 1e-8);
      LinearSystem ls(a, b);
      x = ls.Solve();
      for (int i=1; i<=3; i++)
      {
         TS_ASSERT_DELTA( x(i), 1.0, 1e-8);
      }
      TS_ASSERT_DELTA( x.CalculateNorm(1), 3.0,       1e-8);
      TS_ASSERT_DELTA( x.CalculateNorm(2), sqrt(3.0), 1e-8);
   }
};
```

C.2 ODE Solver Library

The code below is example solutions for the classes developed in the Exercises at the end of Chap. 12.

Listing C.6 FiniteDifferenceGrid.cpp

```
#include <cassert>
#include "FiniteDifferenceGrid.hpp"
#include "Node.hpp"

FiniteDifferenceGrid::FiniteDifferenceGrid(int numNodes,
                                          double xMin, double xMax)
{
   double stepsize = (xMax-xMin)/((double)(numNodes-1));
   for (int i=0; i<numNodes; i++)
   {
      Node node;
      node.coordinate = xMin+i*stepsize;
      mNodes.push_back(node);
   }
   assert(mNodes.size() == numNodes);
}
```

Listing C.7 `BvpOde.cpp`

```cpp
#include <iostream>
#include <fstream>
#include <cassert>
#include "BvpOde.hpp"

BvpOde::BvpOde(SecondOrderOde* pOde,
               BoundaryConditions* pBcs, int numNodes)
{
   mpOde = pOde;
   mpBconds = pBcs;

   mNumNodes = numNodes;
   mpGrid = new FiniteDifferenceGrid(mNumNodes, pOde->mXmin,
                        pOde->mXmax);

   mpSolVec = new Vector(mNumNodes);
   mpRhsVec = new Vector(mNumNodes);
   mpLhsMat = new Matrix(mNumNodes, mNumNodes);

   mFilename = "ode_output.dat";
   mpLinearSystem = NULL;
}

BvpOde::~BvpOde()
{
   // Deletes memory allocated in constructor
   delete mpSolVec;
   delete mpRhsVec;
   delete mpLhsMat;
   delete mpGrid;
   // Only delete if Solve has been called
   if (mpLinearSystem)
   {
      delete mpLinearSystem;
   }
}

void BvpOde::Solve()
{
   PopulateMatrix();
   PopulateVector();
   ApplyBoundaryConditions();
   mpLinearSystem = new LinearSystem(*mpLhsMat, *mpRhsVec);
   *mpSolVec = mpLinearSystem->Solve();
   WriteSolutionFile();
}

void BvpOde::PopulateMatrix()
{
   for (int i=1; i<mNumNodes-1; i++)
   {
      // xm, x and xp are  x(i-1), x(i) and x(i+1)
      double xm = mpGrid->mNodes[i-1].coordinate;
      double x = mpGrid->mNodes[i].coordinate;
      double xp = mpGrid->mNodes[i+1].coordinate;
      double alpha = 2.0/(xp-xm)/(x-xm);
      double beta = -2.0/(xp-x)/(x-xm);
      double gamma = 2.0/(xp-xm)/(xp-x);
      (*mpLhsMat)(i+1,i) = (mpOde->mCoeffOfUxx)*alpha -
                           (mpOde->mCoeffOfUx)/(xp-xm);
      (*mpLhsMat)(i+1,i+1) = (mpOde->mCoeffOfUxx)*beta +
                             mpOde->mCoeffOfU;
      (*mpLhsMat)(i+1,i+2) = (mpOde->mCoeffOfUxx)*gamma +
                             (mpOde->mCoeffOfUx)/(xp-xm);
   }
}

void BvpOde::PopulateVector()
{
   for (int i=1; i<mNumNodes-1; i++)
   {
      double x = mpGrid->mNodes[i].coordinate;
      (*mpRhsVec)(i+1) = mpOde->mpRhsFunc(x);
   }
}

void BvpOde::ApplyBoundaryConditions()
{
   bool left_bc_applied = false;
   bool right_bc_applied = false;

   if (mpBconds->mLhsBcIsDirichlet)
   {
      (*mpLhsMat)(1,1) = 1.0;
      (*mpRhsVec)(1) = mpBconds->mLhsBcValue;
      left_bc_applied = true;
   }

   if (mpBconds->mRhsBcIsDirichlet)
   {
      (*mpLhsMat)(mNumNodes,mNumNodes) = 1.0;
      (*mpRhsVec)(mNumNodes) = mpBconds->mRhsBcValue;
      right_bc_applied = true;
   }

   if (mpBconds->mLhsBcIsNeumann)
   {
      assert(left_bc_applied == false);
      double h = mpGrid->mNodes[1].coordinate -
                 mpGrid->mNodes[0].coordinate;
      (*mpLhsMat)(1,1) = -1.0/h;
      (*mpLhsMat)(1,2) = 1.0/h;
      (*mpRhsVec)(1) = mpBconds->mLhsBcValue;
      left_bc_applied = true;
   }

   if (mpBconds->mRhsBcIsNeumann)
   {
      assert(right_bc_applied == false);
      double h = mpGrid->mNodes[mNumNodes-1].coordinate -
                 mpGrid->mNodes[mNumNodes-2].coordinate;
      (*mpLhsMat)(mNumNodes,mNumNodes-1) = -1.0/h;
      (*mpLhsMat)(mNumNodes,mNumNodes) = 1.0/h;
      (*mpRhsVec)(mNumNodes) = mpBconds->mRhsBcValue;
      right_bc_applied = true;
   }

   // Check that boundary conditions have been applied
   // on both boundaries
   assert(left_bc_applied);
   assert(right_bc_applied);
}

void BvpOde::WriteSolutionFile()
{
   std::ofstream output_file(mFilename.c_str());
   assert(output_file.is_open());
   for (int i=0; i<mNumNodes; i++)
   {
      double x = mpGrid->mNodes[i].coordinate;
      output_file << x << "  " << (*mpSolVec)(i+1) << "\n";
   }
   output_file.flush();
   output_file.close();
   std::cout<<"Solution written to "<<mFilename<<"\n";
}
```

Listing C.8 BoundaryConditions.cpp

```
#include <cassert>
#include "BoundaryConditions.hpp"

BoundaryConditions::BoundaryConditions()
{
   mLhsBcIsDirichlet = false;
   mRhsBcIsDirichlet = false;
   mLhsBcIsNeumann = false;
   mRhsBcIsNeumann = false;
}
```

```

void BoundaryConditions::SetLhsDirichletBc(double lhsValue)
{
   assert(!mLhsBcIsNeumann);
   mLhsBcIsDirichlet = true;
   mLhsBcValue = lhsValue;
}

void BoundaryConditions::SetRhsDirichletBc(double rhsValue)
{
   assert(!mRhsBcIsNeumann);
   mRhsBcIsDirichlet = true;
   mRhsBcValue = rhsValue;
}

void BoundaryConditions::
            SetLhsNeumannBc(double lhsDerivValue)
{
   assert(!mLhsBcIsDirichlet);
   mLhsBcIsNeumann = true;
   mLhsBcValue = lhsDerivValue;
}

void BoundaryConditions::
            SetRhsNeumannBc(double rhsDerivValue)
{
   assert(!mRhsBcIsDirichlet);
   mRhsBcIsNeumann = true;
   mRhsBcValue = rhsDerivValue;
}
```

Listing C.9 `BvpOdeTestSuite.hpp`

```
#include <cxxtest/TestSuite.h>
#include <fstream>
#include "BvpOde.hpp"

double model_prob_1_rhs(double x){return 1.0;}
double model_prob_2_rhs(double x){return 34.0*sin(x);}

// This suite is an example solution to Exercise 12.5
class BvpOdeTestSuite : public CxxTest::TestSuite
{
private:
   void ReadIn(const char* rName, std::vector<double>& ts,
                                  std::vector<double>& vs)
   {
      std::ifstream file(rName);
      double time, value;
      while (!file.eof())
      {
         file >> time >> value;
         if (file.good())
         {
            ts.push_back(time); vs.push_back(value);
         }
      }
   }
public:
   void TestModelProblem1(void)
   {
      SecondOrderOde ode_mp1(-1.0, 0.0, 0.0,
                             model_prob_1_rhs,
                             0.0, 1.0);
      BoundaryConditions bc_mp1;
      bc_mp1.SetLhsDirichletBc(0.0);
      bc_mp1.SetRhsDirichletBc(0.0);

      BvpOde bvpode_mp1(&ode_mp1, &bc_mp1, 101);
      bvpode_mp1.SetFilename("model_problem_results1.dat");
      bvpode_mp1.Solve();
      std::vector<double> xs, us;
      ReadIn("model_problem_results1.dat", xs, us);
      TS_ASSERT_EQUALS(xs.size(), 101u);
      TS_ASSERT_EQUALS(us.size(), 101u);
      // Test solution as given in Sec. 12.1.1
      for (int i=0; i<xs.size(); i++)
      {
         TS_ASSERT_DELTA(us[i], xs[i]*(1.0-xs[i])/2.0, 1e-8);
      }
   }

   void TestModelProblem2(void)
   {
      SecondOrderOde ode_mp2(1.0, 3.0, -4.0,
                             model_prob_2_rhs,
                             0.0, M_PI);
      BoundaryConditions bc_mp2;
      bc_mp2.SetLhsNeumannBc(-5.0);
      bc_mp2.SetRhsDirichletBc(4.0);

      BvpOde bvpode_mp2(&ode_mp2, &bc_mp2, 1001);
      bvpode_mp2.SetFilename("model_problem_results2.dat");
      bvpode_mp2.Solve();
      std::vector<double> xs, us;
      ReadIn("model_problem_results2.dat", xs, us);
      TS_ASSERT_EQUALS(xs.size(), 1001u);
      TS_ASSERT_EQUALS(us.size(), 1001u);
      // Test solution as given in Sec. 12.1.1
      for (int i=0; i<xs.size(); i++)
      {
            double u = (4*exp(xs[i])+exp(-4*xs[i]))/
                          (4*exp(M_PI)+exp(-4*M_PI))
                        - 5*sin(xs[i]) - 3*cos(xs[i]);
            TS_ASSERT_DELTA(us[i], u, 2e-3); // Error ~= delta x
        }
    }
};
```

Further Reading

In earlier chapters we have touched on a few issues that are beyond the scope of this book. When discussing these issues we have directed the interested reader towards a selection of various resources: these are listed below thematically. For the "Mathematical Methods and Linear Algebra" theme, the most comprehensive reference for the basic material is that written by Kreyszig. The other references given are suitable for more advanced numerical concepts. For the "C++ Programming" theme, the website http://www.cplusplus.com provides extensive practical guidance, whilst the texts listed focus on advanced features of the language. In the "Message–Passing Interface" theme the texts give an accessible tutorial–based overview of MPI-1 and MPI-2, respectively. The differences between these two MPI standards are discussed in Sect. 11.2.

Mathematical Methods and Linear Algebra

1. Iserles, A.: A First Course in the Numerical Analysis of Differential Equations, 2nd edn. Cambridge University Press, Cambridge (2009)
2. Kreyszig, E.: Advanced Engineering Mathematics, 9th edn. Wiley, Inc., New York (2006)
3. Süli, E., Mayers, D.F.: An Introduction to Numerical Analysis. Cambridge University Press, Cambridge (2006)
4. Trefethen, L.N., Bau, D.: Numerical Linear Algebra, Society for Industrial and Applied Mathematics (1997)

C++ Programming

5. Cline, M.P., Lomow, G., Girou, M.: C++ FAQs, 2nd edn. Addison–Wesley, Boston (1998)
6. Meyers, S.: Effective C++, 3rd edn. Addison–Wesley, Boston (2005)
7. Stroustrup, B.: The C++ Programming Language, 3rd edn. AT&T (2000)
8. The Website, http://www.cplusplus.com

The Message–Passing Interface (MPI)

9. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message–Passing Interface, 2nd edn. Massachussetts Institute of Technology Press, Massachussetts (1999)
10. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message–Passing Interface. Massachussetts Institute of Technology Press, Massachussetts (1999)

Index

A
Abstraction, 2, 100, 101, 148
Access privileges, *see* Classes
Addresses of variables, 59
AND, *see* Logical operators
Arrays, 17
  irregularly sized, 65
  one-based indexing, 39
  safe dynamic allocation, 67, 179
  sending to functions, 82
  static allocation of memory, 17
    initialising, 17
  zero-based indexing, 17, 39
ASCII character variables, 18
`assert`, 22, 163
Assertions, 163

B
Black box testing, *see* Testing software
Blocks, 75
Boolean variables, 19

C
C programming language, 3
  C style dynamic memory allocation, 258
    `free`, 258
    `malloc`, 258
  C style output, 257
C++ file extensions, 5
C++11, *see* Modern C++
`catch`, *see* Exceptions
`cerr`, 47, 48
`cfloat`, 42
`cin`, 21, 51
Classes, 99
  abstract classes, 137
  access privileges, 102, 104, 109, 116, 131, 133, 184, 189
    use for validating data, 105
  class initialisers, 261
  class members, 102
  constructors, 110, 183, 188
    copy constructor, 112, 188
    customised constructor, 112, 189
    default constructor, 110, 188
    overriding, 110
  derived classes, 129, 130
  destructors, 110, 183, 188, 189
  `friend` keyword, 116, 124, 191
  header files, 102, 103
    including only once, 103
  members, 101
  methods, 101, 104
  multiple inheritance, 129
  pointers to, 115
  setting and accessing variables, 104
Closing the file handle, 48
`cmath`, 14
Coding standards, 125
Commenting code, 5, 94, 125
Compiling code, 6
  at the command line, 8
  C++11, 154
  compiler flags, 9, 107
  linking, 107
  Makefiles, 108
  multiple files, 107
  object file, 107
  template compilation, 159

Complex numbers, 117
`const`, 12
Constructors, *see* Classes
`cout`, 5, 20, 48

D
Debugging code, 24, 138, 218
Destructors, *see* Classes
Distributed memory architectures, 197, 198
Division of integers, 16
Documenting code, 94
`double`, 10
Dynamic allocation of memory, *see* Pointers
Dynamic casting, 136

E
`else`, 29, 259
Encapsulation, 2, 100, 101
Exception
  `catch`, 167
  three levels of error, 166
  `throw`, 167
  `try`, 167
Extensibility, 2, 130, 148

F
Finite difference approximations, 227, 239
  application to boundary value problems, 229
Floating point variables, 10, 13
  comparing two floating point variables, 41
  `DBL_EPSILON`, 42
  double precision, 13
Flow of control, 27
Flushing output, 49
`for`, 27, 35
Fortran, 3, 4
`fprintf`, 258
`free`, *see* C programming language
`friend`, *see* Classes
`fstream`, 48, 51
Function overloading, 87
Function pointers, *see* Functions
Function prototypes, *see* Functions
Functions, 77
  class members, 104
  default values for function arguments, 86
  function pointers, 89
  function prototypes, 77, 89
  recursive functions, 92
  return type, 77
  returning a pointer, 79
  sending arrays to, 82
  use of arguments that are pointers, 80

G
Global variable, *see* Variables
GNU Octave, 3
Guard, 28

I
`if`, 27, 28, 259
Indenting code, 6, 28, 125
Infinity, 164
Inheritance, 2, 101, 129, 130
  abstract classes, 137
  base class, 129
  child class, 129
  derived classes, 129
  parent class, 129
  private inheritance, 133
  protected inheritance, 133
  public inheritance, 131, 133
Input
  command line, 54
  from console, 20, 21
  from file, 51
  rewinding a file, 53
  strings, 22
Input stream variable, 51
`int`, 10
Integer variables, 10, 12
  long integers, 12
  short integers, 12
  signed integers, 13
  unsigned integers, 13
Integrated development environments, 7
Interpreted languages, 3
`iostream`, 5

K
Keywords, 11

L
Linear algebra, 183
  conjugate gradient method, 254
  Gaussian elimination, 249
    no pivoting, 250
    pivoting, 252
  linear systems, 249
  LU decomposition, 251, 253
  parallel, 215

Thomas algorithm, 254
Linear algebra algorithms, 245
Local variable, *see* Variables
Logical operators, 31
AND, 31
NOT, 31
OR, 31

M
Makefiles, *see* Compiling code
malloc, *see* C programming language
Mathematical operations, 14
absolute value, 14
arccos, 14
arcsin, 14
arctan, 14
ceiling, 14
cosh, 14
cosine, 14
exponential, 14
floor, 14
logarithm, 14
raising variables to a given power, 14
shorthand, 15
sine, 14
sinh, 14
square root, 14
tangent, 14
tanh, 14
MATLAB, 3, 4, 183
Matrices, 183, 245
determinant, 247
eigenvalues, 248
inverse, 247
norm, 248
operations between, 246
positive definite, 255
symmetric, 247
transpose, 247
Message Passing Interface (MPI), 199
Allgatherv, 212
Allreduce, 210
Barrier, 207
Bcast, 210
broadcast, 210
Bsend, 205
collective communication, 206
broadcast, 206
combined send and receive, 207
halo exchange, 207
reduction, 206
communication, 203
blocking, 205
buffered, 205
point-to-point, 203
compiling, 201
debugging, 218
executing, 202
remote machines, 202
Gather, 212
gather, 211
Ibsend, 206
input and output, 199
installing MPI, 199
Irecv, 206
Isend, 206
process, 200
rank, 200
Recv, 203, 206
reduce, 210
running code, 201
Scatter, 212
scatter, 211
Send, 203
Sendrecv, 208
Ssend, 205
Modern C++, 69, 71, 153
array, 155
auto, 154
compilation, 69
mapping lambda functions, 158
range-based for loops, 157
smart pointers
shared_ptr, 71
unique_ptr, 69
Modularity, 2, 100, 101
Modules, 93, 99, 100
Multiple inheritance, 261

N
Namespace, 259
NOT, *see* Logical operators
Not–a–number, 163
Numerical libraries, 243
PETSc, 243

O
Object–oriented numerical libraries
ordinary differential equations, 232
partial differential equations, 239
Object-orientation, 2, 99
Object-oriented numerical libraries, 225

Operator overloading, 118, 125, 183, 190, 191
OR, *see* Logical operators
Ordinary differential equations, 225, 226
Output
  appending to existing file, 50
  formatting, 21
  redirected to file, 47
  scientific notation, 56
  setting precision of, 51, 56
  to console, 20
  to file, 48
Output stream variable, 48

P
Partial differential equations, 225
Pointer Aliasing, *see* Pointers
Pointer de-reference, *see* Pointers
Pointers
  array of, 65
  delete keyword, 62, 63, 65, 68, 113, 188, 189, 192, 258
  dynamic allocation of memory, 62
  new keyword, 61, 63, 64, 68, 113, 188, 189, 192, 258
  pointer aliasing, 62, 66, 69, 71
  pointer de–reference, 61
  pointer variables, 60
  shared pointers, 68
  smart pointers, 69, 71
  to classes, 115
Polymorphism, 2, 130
  run–time, 134
  static, *see* Templates
  templates, *see* Templates
Postcondition, 164
Precondition, 164
printf, 257, 258
Python, 3

R
Recursive functions, *see* Functions
Reference variables, 85
Relational operators, 31, 32
  equal to, 32
  equality versus assignment, 39
  greater than, 32
  greater than or equal to, 32
  less than, 32
  less than or equal to, 32
  not equal to, 32

Reserved words, 11

S
Scientific notation, *see* Variables
Scope, 75
Shared memory architectures, 197
Shared pointers, *see* Pointers
sprintf, 258
Standard Template Library (STL), 145, 148
  abstraction, 148
  algorithm, 150
  container, 148
  deque, 148
  iterator, 149
  list, 148
  push_back, 149
  set, 151
  vector, 149
  vectors, 148
Statements, 5
stdio.h, 258
Strings, 19
Structures, 260
switch, 37

T
Templates, 145
  for polymorphism, 147
  Standard Template Library, *see* Standard Template Library (STL)
  validation of index to array, 146
Ternary ?: operator, 259
Test driven development, *see* Testing software
Testing software, 169
  black box testing, 172
  test-driven development, 177
  white box testing, 176
throw, *see* Exceptions
try, *see* Exceptions
Type conversion, 16

U
Unified modelling language, 132

V
Variables, 10
  constant variables, 12
  declaring variables, 10
  global variable, 76
  initialising variables, 11

local variable, 76
scientific notation, 12
Vectors, 183, 245
norm, 248
operations between, 246
scalar product, 247
Virtual methods, 134, 137
pure virtual method, 137
`void`, 78

W

`while`, 27, 33
White box testing, *see* Testing software