

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 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 \mathbf{A} is a matrix, then the entry of \mathbf{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 $\mathbf{I}^{(N)}$, 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 $\mathbf{u}, \mathbf{v}, \mathbf{w}$ are all vectors of length N , and α, β are scalars. The entries of \mathbf{w} are given by

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

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

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

Multiplication of a matrix by a vector. Suppose \mathbf{A} is a matrix with M rows and N columns, and \mathbf{u} is a vector of length N . If $\mathbf{v} = \mathbf{A}\mathbf{u}$, then \mathbf{v} is a vector of length M with entries given by

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

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

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

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

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

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

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

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

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

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

$$\det(\mathbf{A}) = a_{00} \det(\hat{\mathbf{A}}^{(00)}) - a_{01} \det(\hat{\mathbf{A}}^{(01)}) + a_{02} \det(\hat{\mathbf{A}}^{(02)}) - a_{03} \det(\hat{\mathbf{A}}^{(03)}) + \dots \\ + (-1)^{N-1} a_{0,N-1} \det(\hat{\mathbf{A}}^{(0,N-1)}),$$

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.

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, then \mathbf{A}^{-1} exists and is given by

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

¹This 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 .

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 vectors² \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

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

Taking the limit as $p \rightarrow \infty$, 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 $\|\mathbf{A}\|_2 = \sqrt{\lambda}$.

²A 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*.

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:

$$m_{i0} = a_{i0}/a_{00}, \quad i = 1, 2, \dots, N-1, \text{ using the assumption that } a_{00} \neq 0,$$

$$a_{ij}^{(1)} = a_{ij} - m_{i0}a_{0j}, \quad i, j = 1, 2, \dots, N-1,$$

$$b_i^{(1)} = b_i - m_{i0}b_0, \quad i = 1, 2, \dots, N-1.$$

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:

$$m_{i1} = a_{i1}^{(1)}/a_{11}^{(1)}, \quad i = 2, 3, \dots, N-1, \text{ using the assumption that } a_{11}^{(1)} \neq 0,$$

$$a_{ij}^{(2)} = a_{ij}^{(1)} - m_{i1}a_{1j}^{(1)}, \quad i, j = 2, 3, \dots, N-1,$$

$$b_i^{(2)} = b_i^{(1)} - m_{i1}b_1^{(1)}, \quad i = 2, 3, \dots, N-1.$$

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

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

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

$$x_{N-1} = b_{N-1}^{(N-1)} / a_{N-1,N-1}^{(N-1)},$$

$$x_{N-2} = \frac{1}{a_{N-2,N-2}^{(N-2)}} (b_{N-2}^{(N-2)} - a_{N-2,N-1}^{(N-2)} x_{N-1}).$$

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

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

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 \mathbf{A} as the product of a lower triangular matrix \mathbf{L} and an upper triangular matrix \mathbf{U} , that is, $\mathbf{A} = \mathbf{LU}$. Defining the matrices $\mathbf{M}_0, \mathbf{M}_1, \dots$ by

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

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

we may write

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

or, equivalently,

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

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

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

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

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

$$\mathbf{L} = \mathbf{M}_0^{-1} \mathbf{M}_1^{-1} \dots \mathbf{M}_{N-1}^{-1} \mathbf{M}_{N-2}^{-1},$$

$$\mathbf{U} = \mathbf{A}^{(N-1)},$$

we see that we have $\mathbf{A} = \mathbf{L}\mathbf{U}$ 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

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

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 $a_{nk}^{(k)}$, $n = k, k + 1, \dots, N - 1$, 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

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

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

$$e_0 = 0, \quad f_0 = b_0, \\ e_i = \frac{r_i}{q_i - p_i e_{i-1}}, \quad f_i = \frac{b_i + p_i f_{i-1}}{q_i - p_i e_{i-1}}, \quad i = 1, 2, \dots, N - 2,$$

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

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

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

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

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:  $k = 0, \mathbf{r} = \mathbf{b} - \mathbf{Ax}_k, \mathbf{p} = \mathbf{0}, \beta = 0$ 
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:    $\alpha = \frac{\mathbf{r}^\top \mathbf{r}}{\mathbf{p}^\top \mathbf{A} \mathbf{p}}$ 
8:    $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha \mathbf{p}$ 
9:    $\mathbf{r}_{\text{prev}} = \mathbf{r}$ 
10:   $\mathbf{r} = \mathbf{b} - \mathbf{Ax}_{k+1}$ 
11:   $k = k + 1$ 
12: end while
13:  $\mathbf{x} = \mathbf{x}_k$ 

```

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 $\mathbf{x}_k, k = 1, 2, \dots$ are generated until $\|\mathbf{b} - \mathbf{Ax}_k\| < \varepsilon$ 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.

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:

$$\mathbf{x}^\top \mathbf{Ax} \geq 0, \quad \text{and}$$

$$\mathbf{x}^\top \mathbf{Ax} = 0 \quad \text{only if } \mathbf{x} = \mathbf{0}.$$

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++.

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[])
4  {
5      double x = 105.0;
6      int j = 500;
7      printf("x = %f and j = %i\n", x, j);
8      printf("In scientific notation, x = %e\n", x);
9      return 0;
10 }
```

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.

```
1  double a, b, x;
2  if (a > b)
3  {
```

```
4     x = 100.0;
5     }
6     else
7     {
8         // a <= b
9         x = 0.0;
10    }
```

The ternary¹ `?:` operator has identical effect to the `if-else` statements above: the code above may be written identically as

```
1     double a, b, x;
2     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.

```
1     #include <iostream>
2
3     using namespace std;
4     int main(int argc, char* argv[])
5     {
6         string city = "Cambridge";
7         cout << city << "\n";
8         return 0;
9     }
```

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.

¹A ternary operator has three inputs.

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 dereferencing a pointer).

```
1  #include <iostream>
2
3  struct ModelParameters
4  {
5      double viscosity;
6      double density;
7      int numberOfDimensions;
8  };
9
10 int main(int argc, char* argv[])
11 {
12     ModelParameters example1;
13     example1.viscosity = 1.0e-4;
14     example1.density = 1.0;
15     ModelParameters* p_eg1 = &example1;
16     p_eg1->numberOfDimensions = 3;
17
18     std::cout << "Density is " << example1.density << "\n";
19
20     return 0;
21 }
```

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`.

```

1  #include "MatrixDet.hpp"
2  #include "MatrixEigsNorm.hpp"
3
4  class MatrixCombined: public MatrixDet,
5                        public MatrixEigsNorm
6  {
7      // Body of class
8  };

```

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:

```

1      MatrixCombined mat;
2      // use method ZeroEntries from the class MatrixDet
3      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.

```

1  #include "ComplexNumber.hpp"
2  // Override default constructor
3  // Set real and imaginary parts to zero
4  ComplexNumber::ComplexNumber()
5  {
6      mRealPart = 0.0;
7      mImaginaryPart = 0.0;
8  }

```

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.

```
1  #include "ComplexNumber.hpp"
2  // Override default constructor
3  // Initialize real and imaginary parts as zero
4  ComplexNumber::ComplexNumber() :
5      mRealPart(0.0),
6      mImaginaryPart(0.0)
7  {
8      // possibly have more code in body
9  }
```

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

```

1  #ifndef MATRIXHEADERDEF
2  #define MATRIXHEADERDEF
3  #include "Vector.hpp"
4
5  class Matrix
6  {
7  private:
8      double** mData; // entries of matrix
9      int mNumRows, mNumCols; // dimensions
10 public:
11     Matrix(const Matrix& otherMatrix);
12     Matrix(int numRows, int numCols);
13     ~Matrix();
14     int GetNumberOfRows() const;
15     int GetNumberOfColumns() const;
16     double& operator()(int i, int j); //1-based indexing
17     //overloaded assignment operator
18     Matrix& operator=(const Matrix& otherMatrix);
19     Matrix operator+() const; // unary +
20     Matrix operator-() const; // unary -
21     Matrix operator+(const Matrix& m1) const; // binary +
22     Matrix operator-(const Matrix& m1) const; // binary -
23     // scalar multiplication
24     Matrix operator*(double a) const;
25     double CalculateDeterminant() const;
26     // declare vector multiplication friendship
27     friend Vector operator*(const Matrix& m,
28                             const Vector& v);

```

```

29     friend Vector operator*(const Vector& v,
30                             const Matrix& m);
31 };
32 // prototype signatures for friend operators
33 Vector operator*(const Matrix& m, const Vector& v);
34 Vector operator*(const Vector& v, const Matrix& m);
35
36 #endif

```

Listing C.2 Matrix.cpp

```

1  #include <cmath>
2  #include <cassert>
3  #include "Matrix.hpp"
4  #include "Vector.hpp"
5
6
7  // Overwritten copy constructor
8  // Allocate memory for new matrix, and copy
9  // entries into this matrix
10 Matrix::Matrix(const Matrix& otherMatrix)
11 {
12     mNumRows = otherMatrix.mNumRows;
13     mNumCols = otherMatrix.mNumCols;
14     mData = new double* [mNumRows];
15     for (int i=0; i<mNumRows; i++)
16     {
17         mData[i] = new double [mNumCols];
18     }
19     for (int i=0; i<mNumRows; i++)
20     {
21         for (int j=0; j<mNumCols; j++)
22         {
23             mData[i][j] = otherMatrix.mData[i][j];
24         }
25     }
26 }
27
28 // Constructor for vector of a given length
29 // Allocates memory, and initialises entries
30 // to zero
31 Matrix::Matrix(int numRows, int numCols)
32 {
33     assert(numRows > 0);
34     assert(numCols > 0);
35     mNumRows = numRows;

```

```
36     mNumCols = numCols;
37     mData = new double* [mNumRows];
38     for (int i=0; i<mNumRows; i++)
39     {
40         mData[i] = new double [mNumCols];
41     }
42     for (int i=0; i<mNumRows; i++)
43     {
44         for (int j=0; j<mNumCols; j++)
45         {
46             mData[i][j] = 0.0;
47         }
48     }
49 }
50
51 // Overwritten destructor to correctly free memory
52 Matrix::~Matrix()
53 {
54     for (int i=0; i<mNumRows; i++)
55     {
56         delete[] mData[i];
57     }
58     delete[] mData;
59 }
60
61 // Method to get number of rows of matrix
62 int Matrix::GetNumberOfRows() const
63 {
64     return mNumRows;
65 }
66
67 // Method to get number of columns of matrix
68 int Matrix::GetNumberOfColumns() const
69 {
70     return mNumCols;
71 }
72
73 // Overloading the round brackets
74 // Note that this uses 'one-based' indexing,
75 // and a check on the validity of the index
76 double& Matrix::operator()(int i, int j)
77 {
78     assert(i > 0);
79     assert(i < mNumRows+1);
80     assert(j > 0);
81     assert(j < mNumCols+1);
82     return mData[i-1][j-1];
83 }
```

```
84
85 // Overloading the assignment operator
86 Matrix& Matrix::operator=(const Matrix& otherMatrix)
87 {
88     assert(mNumRows == otherMatrix.mNumRows);
89     assert(mNumCols == otherMatrix.mNumCols);
90
91     for (int i=0; i<mNumRows; i++)
92     {
93         for (int j=0; j<mNumCols; j++)
94         {
95             mData[i][j] = otherMatrix.mData[i][j];
96         }
97     }
98     return *this;
99 }
100
101 // Overloading the unary + operator
102 Matrix Matrix::operator+() const
103 {
104     Matrix mat(mNumRows, mNumCols);
105     for (int i=0; i<mNumRows; i++)
106     {
107         for (int j=0; j<mNumCols; j++)
108         {
109             mat(i+1,j+1) = mData[i][j];
110         }
111     }
112     return mat;
113 }
114
115 // Overloading the unary - operator
116 Matrix Matrix::operator-() const
117 {
118     Matrix mat(mNumRows, mNumCols);
119     for (int i=0; i<mNumRows; i++)
120     {
121         for (int j=0; j<mNumCols; j++)
122         {
123             mat(i+1,j+1) = -mData[i][j];
124         }
125     }
126     return mat;
127 }
128
129 // Overloading the binary + operator
130 Matrix Matrix::operator+(const Matrix& m1) const
131 {
132     assert(mNumRows == m1.mNumRows);
133     assert(mNumCols == m1.mNumCols);
134     Matrix mat(mNumRows, mNumCols);
```

```
135     for (int i=0; i<mNumRows; i++)
136     {
137         for (int j=0; j<mNumCols; j++)
138         {
139             mat(i+1,j+1) = mData[i][j] + m1.mData[i][j];
140         }
141     }
142     return mat;
143 }
144
145 // Overloading the binary - operator
146 Matrix Matrix::operator-(const Matrix& m1) const
147 {
148     assert(mNumRows == m1.mNumRows);
149     assert(mNumCols == m1.mNumCols);
150     Matrix mat(mNumRows, mNumCols);
151     for (int i=0; i<mNumRows; i++)
152     {
153         for (int j=0; j<mNumCols; j++)
154         {
155             mat(i+1,j+1) = mData[i][j] - m1.mData[i][j];
156         }
157     }
158     return mat;
159 }
160
161 // Overloading scalar multiplication
162 Matrix Matrix::operator*(double a) const
163 {
164     Matrix mat(mNumRows, mNumCols);
165     for (int i=0; i<mNumRows; i++)
166     {
167         for (int j=0; j<mNumCols; j++)
168         {
169             mat(i+1,j+1) = a*mData[i][j];
170         }
171     }
172     return mat;
173 }
174
175 // Overloading matrix multiplied by a vector
176 Vector operator*(const Matrix& m, const Vector& v)
177 {
178     int original_vector_size = v.GetSize();
179     assert(m.GetNumberOfColumns() == original_vector_size);
180     int new_vector_length = m.GetNumberOfRows();
181     Vector new_vector(new_vector_length);
182
183     for (int i=0; i<new_vector_length; i++)
```

```

184     {
185         for (int j=0; j<original_vector_size; j++)
186         {
187             new_vector[i] += m.mData[i][j]*v.Read(j);
188         }
189     }
190
191     return new_vector;
192 }
193
194 // Overloading vector multiplied by a matrix
195 Vector operator*(const Vector& v, const Matrix& m)
196 {
197     int original_vector_size = v.GetSize();
198     assert(m.GetNumberOfRows() == original_vector_size);
199     int new_vector_length = m.GetNumberOfColumns();
200     Vector new_vector(new_vector_length);
201
202     for (int i=0; i<new_vector_length; i++)
203     {
204         for (int j=0; j<original_vector_size; j++)
205         {
206             new_vector[i] += v.Read(j)*m.mData[j][i];
207         }
208     }
209
210     return new_vector;
211 }
212
213 // Calculate determinant of square matrix recursively
214 double Matrix::CalculateDeterminant() const
215 {
216     assert(mNumRows == mNumCols);
217     double determinant = 0.0;
218
219     if (mNumRows == 1)
220     {
221         determinant = mData[0][0];
222     }
223     else
224     {
225         // More than one entry of matrix
226         for (int i_outer=0; i_outer<mNumRows; i_outer++)
227         {
228             Matrix sub_matrix(mNumRows-1,
229                               mNumRows-1);
230             for (int i=0; i<mNumRows-1; i++)
231             {
232                 for (int j=0; j<i_outer; j++)
233                 {
234                     sub_matrix(i+1,j+1) = mData[i+1][j];
235                 }

```

```

236         for (int j=i_outer; j<mNumRows-1; j++)
237             {
238                 sub_matrix(i+1,j+1) = mData[i+1][j+1];
239             }
240     }
241     double sub_matrix_determinant =
242         sub_matrix.CalculateDeterminant();
243
244     determinant += pow(-1.0, i_outer)*
245         mData[0][i_outer]*sub_matrix_determinant;
246     }
247 }
248 return determinant;
249 }

```

Listing C.3 LinearSystem.hpp

```

1  #ifndef LINEARSYSTEMHEADERDEF
2  #define LINEARSYSTEMHEADERDEF
3  #include "Vector.hpp"
4  #include "Matrix.hpp"
5
6  class LinearSystem
7  {
8  private:
9      int mSize; // size of linear system
10     Matrix* mpA; // matrix for linear system
11     Vector* mpb; // vector for linear system
12
13     // Only allow constructor that specifies matrix and
14     // vector to be used. Copy constructor is private.
15     LinearSystem(const LinearSystem& otherLinearSystem){};
16 public:
17     LinearSystem(const Matrix& A, const Vector& b);
18
19     // destructor frees memory allocated
20     ~LinearSystem();
21
22     // Method for solving system
23     virtual Vector Solve();
24 };
25
26 #endif

```


Listing C.4 LinearSystem.cpp

```
1  #include <cmath>
2  #include <cassert>
3  #include "LinearSystem.hpp"
4  #include "Matrix.hpp"
5  #include "Vector.hpp"
6
7  // Copy matrix and vector so that original matrix and vector
8  // specified are unchanged by Gaussian elimination
9  LinearSystem::LinearSystem(const Matrix& A, const Vector& b)
10 {
11     // check matrix and vector are of compatible sizes
12     int local_size = A.GetNumberOfRows();
13     assert(A.GetNumberOfColumns() == local_size);
14     assert(b.GetSize() == local_size);
15
16     // set variables for linear system
17     mSize = local_size;
18     mpA = new Matrix(A);
19     mpb = new Vector(b);
20 }
21
22 // Destructor to free memory
23 LinearSystem::~LinearSystem()
24 {
25     delete mpA;
26     delete mpb;
27 }
28
29 // Solve linear system using Gaussian elimination
30 // This method changes the content of the matrix mpA
31 Vector LinearSystem::Solve()
32 {
33     Vector m(mSize); //See description in Appendix A
34     Vector solution(mSize);
35
36     // We introduce references to make the syntax readable
37     Matrix rA = *mpA;
38     Vector rb = *mpb;
39
40     // forward sweep of Gaussian elimination
41     for (int k=0; k<mSize-1; k++)
42     {
43         // see if pivoting is necessary
44         double max = 0.0;
45         int row = -1;
46         for (int i=k; i<mSize; i++)
```

```
47     {
48         if (fabs(rA(i+1,k+1)) > max)
49         {
50             row = i;
51         }
52     }
53     assert(row > 0);
54
55     // pivot if necessary
56     if (row != k)
57     {
58         // swap matrix rows k+1 with row+1
59         for (int i=0; i<mSize; i++)
60         {
61             double temp = rA(k+1,i+1);
62             rA(k+1,i+1) = rA(row+1,i+1);
63             rA(row+1,i+1) = temp;
64         }
65         // swap vector entries k+1 with row+1
66         double temp = rb(k+1);
67         rb(k+1) = rb(row+1);
68         rb(row+1) = temp;
69     }
70
71     // create zeros in lower part of column k
72     for (int i=k+1; i<mSize; i++)
73     {
74         m(i+1) = rA(i+1,k+1)/rA(k+1,k+1);
75         for (int j=k; j<mSize; j++)
76         {
77             rA(i+1,j+1) -= rA(k+1,j+1)*m(i+1);
78         }
79         rb(i+1) -= rb(k+1)*m(i+1);
80     }
81 }
82
83 // back substitution
84 for (int i=mSize-1; i>-1; i--)
85 {
86     solution(i+1) = rb(i+1);
87     for (int j=i+1; j<mSize; j++)
88     {
89         solution(i+1) -= rA(i+1,j+1)*solution(j+1);
90     }
91     solution(i+1) /= rA(i+1,i+1);
92 }
93
94 return solution;
95 }
```

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.5 FiniteDifferenceGrid.cpp

```

1  #include <cassert>
2  #include "FiniteDifferenceGrid.hpp"
3  #include "Node.hpp"
4
5  FiniteDifferenceGrid::FiniteDifferenceGrid(int numNodes,
6                                           double xMin, double xMax)
7  {
8      double stepsize = (xMax-xMin)/((double) (numNodes-1));
9      for (int i=0; i<numNodes; i++)
10     {
11         Node node;
12         node.coordinate = xMin+i*stepsize;
13         mNodes.push_back(node);
14     }
15     assert(mNodes.size() == numNodes);
16 }

```

Listing C.6 BvpOde.cpp

```

1  #include <iostream>
2  #include <fstream>
3  #include <cassert>
4  #include "BvpOde.hpp"
5
6  BvpOde::BvpOde(SecondOrderOde* pOde,
7                BoundaryConditions* pBcs, int numNodes)
8  {
9      mpOde = pOde;
10     mpBconds = pBcs;
11
12     mNumNodes = numNodes;
13     mpGrid = new FiniteDifferenceGrid(mNumNodes, pOde->mXmin,
14                                       pOde->mXmax);
15
16     mpSolVec = new Vector(mNumNodes);
17     mpRhsVec = new Vector(mNumNodes);
18     mpLhsMat = new Matrix(mNumNodes, mNumNodes);
19
20     mFilename = "ode_output.dat";
21     mpLinearSystem = NULL;
22 }

```

```

23 BvpOde::~BvpOde()
24 {
25     // Deletes memory allocated in constructor
26     delete mpSolVec;
27     delete mpRhsVec;
28     delete mpLhsMat;
29     delete mpGrid;
30     // Only delete if Solve has been called
31     if (mpLinearSystem)
32     {
33         delete mpLinearSystem;
34     }
35 }
36
37 void BvpOde::Solve()
38 {
39     PopulateMatrix();
40     PopulateVector();
41     ApplyBoundaryConditions();
42     mpLinearSystem = new LinearSystem(*mpLhsMat, *mpRhsVec);
43     *mpSolVec = mpLinearSystem->Solve();
44     WriteSolutionFile();
45 }
46
47 void BvpOde::PopulateMatrix()
48 {
49     for (int i=1; i<mNumNodes-1; i++)
50     {
51         // xm, x and xp are x(i-1), x(i) and x(i+1)
52         double xm = mpGrid->mNodes[i-1].coordinate;
53         double x = mpGrid->mNodes[i].coordinate;
54         double xp = mpGrid->mNodes[i+1].coordinate;
55         double alpha = 2.0/(xp-xm)/(x-xm);
56         double beta = -2.0/(xp-x)/(x-xm);
57         double gamma = 2.0/(xp-xm)/(xp-x);
58         (*mpLhsMat)(i+1,i) = (mpOde->mCoeffOfUxx)*alpha -
59             (mpOde->mCoeffOfUx)/(xp-xm);
60         (*mpLhsMat)(i+1,i+1) = (mpOde->mCoeffOfUxx)*beta +
61             mpOde->mCoeffOfU;
62         (*mpLhsMat)(i+1,i+2) = (mpOde->mCoeffOfUxx)*gamma +
63             (mpOde->mCoeffOfUx)/(xp-xm);
64     }
65 }
66
67 void BvpOde::PopulateVector()
68 {
69     for (int i=1; i<mNumNodes-1; i++)
70     {
71         double x = mpGrid->mNodes[i].coordinate;
72         (*mpRhsVec)(i+1) = mpOde->mpRhsFunc(x);
73     }

```

```

74     }
75 }
76
77 void BvpOde::ApplyBoundaryConditions()
78 {
79     bool left_bc_applied = false;
80     bool right_bc_applied = false;
81
82     if (mpBconds->mLhsBcIsDirichlet)
83     {
84         (*mpLhsMat)(1,1) = 1.0;
85         (*mpRhsVec)(1) = mpBconds->mLhsBcValue;
86         left_bc_applied = true;
87     }
88
89     if (mpBconds->mRhsBcIsDirichlet)
90     {
91         (*mpLhsMat)(mNumNodes,mNumNodes) = 1.0;
92         (*mpRhsVec)(mNumNodes) = mpBconds->mRhsBcValue;
93         right_bc_applied = true;
94     }
95
96     if (mpBconds->mLhsBcIsNeumann)
97     {
98         assert(left_bc_applied == false);
99         double h = mpGrid->mNodes[1].coordinate -
100                 mpGrid->mNodes[0].coordinate;
101         (*mpLhsMat)(1,1) = -1.0/h;
102         (*mpLhsMat)(1,2) = 1.0/h;
103         (*mpRhsVec)(1) = mpBconds->mLhsBcValue;
104         left_bc_applied = true;
105     }
106
107     if (mpBconds->mRhsBcIsNeumann)
108     {
109         assert(right_bc_applied == false);
110         double h = mpGrid->mNodes[mNumNodes-1].coordinate -
111                 mpGrid->mNodes[mNumNodes-2].coordinate;
112         (*mpLhsMat)(mNumNodes,mNumNodes-1) = -1.0/h;
113         (*mpLhsMat)(mNumNodes,mNumNodes) = 1.0/h;
114         (*mpRhsVec)(mNumNodes) = mpBconds->mRhsBcValue;
115         right_bc_applied = true;
116     }
117
118     // Check that boundary conditions have been applied
119     // on both boundaries
120     assert(left_bc_applied);
121     assert(right_bc_applied);
122 }
123
124 void BvpOde::WriteSolutionFile()

```

```

125 {
126     std::ofstream output_file(mFilename.c_str());
127     assert(output_file.is_open());
128     for (int i=0; i<mNumNodes; i++)
129     {
130         double x = mpGrid->mNodes[i].coordinate;
131         output_file << x << " " << (*mpSolVec)(i+1) << "\n";
132     }
133     output_file.flush();
134     output_file.close();
135     std::cout<<"Solution written to "<<mFilename<<"\n";
136 }

```

Listing C.7 BoundaryConditions.cpp

```

1  #include <cassert>
2  #include "BoundaryConditions.hpp"
3
4  BoundaryConditions::BoundaryConditions()
5  {
6      mLhsBcIsDirichlet = false;
7      mRhsBcIsDirichlet = false;
8      mLhsBcIsNeumann = false;
9      mRhsBcIsNeumann = false;
10 }
11
12 void BoundaryConditions::SetLhsDirichletBc(double lhsValue)
13 {
14     assert(!mLhsBcIsNeumann);
15     mLhsBcIsDirichlet = true;
16     mLhsBcValue = lhsValue;
17 }
18
19 void BoundaryConditions::SetRhsDirichletBc(double rhsValue)
20 {
21     assert(!mRhsBcIsNeumann);
22     mRhsBcIsDirichlet = true;
23     mRhsBcValue = rhsValue;
24 }
25
26 void BoundaryConditions::
27     SetLhsNeumannBc(double lhsDerivValue)
28 {
29     assert(!mLhsBcIsDirichlet);
30     mLhsBcIsNeumann = true;
31     mLhsBcValue = lhsDerivValue;
32 }

```

```
33
34 void BoundaryConditions::
35     SetRhsNeumannBc(double rhsDerivValue)
36 {
37     assert(!mRhsBcIsDirichlet);
38     mRhsBcIsNeumann = true;
39     mRhsBcValue = rhsDerivValue;
40 }
```

Further Reading¹

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, 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, Philadelphia (1997)

C++ Programming

5. Cline, M.P., Lomow, G., Girou, M.: C++ FAQs, 2nd edn. Addison–Wesley, Reading (1998)
6. Meyers, S.: Effective C++, 3rd edn. Addison–Wesley, Reading (2005)
7. Stroustrup, B.: The C++ Programming Language, 3rd edn. AT&T, New York (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. Massachusetts Institute of Technology Press, Cambridge (1999)
10. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. Massachusetts Institute of Technology Press, Cambridge (1999)

¹In 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 above 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.

Index

A

Abstraction, 2, 88, 89, 134
Access privileges, *see* Classes
Addresses of variables, 55
AND, *see* Logical operators
Arrays, 16
 irregularly sized, 61
 one-based indexing, 36
 safe dynamic allocation, 63, 147
 sending to functions, 71
 static allocation of memory, 16
 initialising, 16
 zero-based indexing, 16, 36
ASCII character variables, 17
assert, 21, 141
Assertions, 141

B

Blocks, 65
Boolean variables, 17

C

C programming language, 3
 C style dynamic memory allocation, 226
 free, 226
 malloc, 226
 C style output, 225
C++ file extensions, 5
catch, *see* Exceptions
cerr, 43, 44
cfloat, 39
cin, 20, 47
Classes, 87
 abstract classes, 124
 access privileges, 90, 92, 96, 103, 118, 120,
 151, 157
 use for validating data, 93
 class initialisers, 229
 class members, 89
 constructors, 98, 151, 156
 copy constructor, 100, 156

 customised constructor, 100, 157
 default constructor, 98, 156
 overriding, 98
 derived classes, 117, 118
 destructors, 98, 151, 156, 157
 friend keyword, 103, 111, 159
 header files, 90, 91
 including only once, 91
 members, 89
 methods, 89, 92
 multiple inheritance, 117
 pointers to, 103
 setting and accessing variables, 92
Closing the file handle, 44
cmath, 13
Coding standards, 112
Commenting code, 5, 83, 112
Compiling code, 6
 at the command line, 7
 compiler flags, 8, 95
 linking, 95
 Makefiles, 95
 multiple files, 94
 object file, 94
 template compilation, 139
Complex numbers, 105
const, 11
Constructors, *see* Classes
cout, 6, 19, 44

D

Debugging code, 22, 126, 186
Destructors, *see* Classes
Distributed memory architectures, 165, 166
Division of integers, 15
Documenting code, 83
double, 10
Dynamic allocation of memory, *see* Pointers
Dynamic casting, 124

E

else, 27, 226
 Encapsulation, 2, 88, 89
 Exception
 catch, 144
 three levels of error, 143
 throw, 144
 try, 144
 Extensibility, 2, 118, 134

F

Finite difference approximations, 195, 207
 application to boundary value problems, 197
 Floating point variables, 10, 13
 comparing two floating point variables, 39
 DBL_EPSILON, 39
 double precision, 13
 Flow of control, 25
 Flushing output, 44, 45
 for, 25, 32
 Fortran, 3, 4
 fprintf, 226
 free, *see* C programming language
 friend, *see* Classes
 fstream, 44, 47
 Function overloading, 76
 Function pointers, *see* Functions
 Function prototypes, *see* Functions
 Functions, 66
 class members, 92
 default values for function arguments, 75
 function pointers, 79
 function prototypes, 66, 78
 recursive functions, 81
 return type, 67
 returning a pointer, 69
 sending arrays to, 71
 use of arguments that are pointers, 70

G

Global variable, *see* Variables
 GNU Octave, 3
 Guard, 26

I

if, 25, 226
 Indenting code, 6, 26, 112
 Infinity, 142
 Inheritance, 2, 89, 117, 118
 abstract classes, 124
 base class, 117
 child class, 117
 derived classes, 117

parent class, 117
 private inheritance, 120
 protected inheritance, 120
 public inheritance, 118, 120

Input, 43

command line, 49
 from console, 19, 20
 from file, 47
 rewinding a file, 48
 strings, 20

Input stream variable, 47**int, 10**

Integer variables, 10, 12
 long integers, 12
 short integers, 12
 signed integers, 12
 unsigned integers, 12

Integrated development environments, 6**Interpreted languages, 3****iostream, 5****K****Keywords, 11****L**

Linear algebra, 151
 conjugate gradient method, 222
 Gaussian elimination, 217
 no pivoting, 217
 pivoting, 220
 linear systems, 217
 LU decomposition, 219, 221
 parallel, 182
 Thomas algorithm, 222
 Linear algebra algorithms, 213
 Local variable, *see* Variables
 Logical operators, 29
 AND, 29
 NOT, 29
 OR, 29

M

Makefiles, *see* Compiling code
 malloc, *see* C programming language
 Mathematical operations, 13
 absolute value, 14
 arccos, 14
 arcsin, 14
 arctan, 14
 ceiling, 14
 cosh, 14
 cosine, 14
 exponential, 13
 floor, 14

- Mathematical operations (*cont.*)
- logarithm, 14
 - raising variables to a given power, 13
 - shorthand, 14
 - sine, 14
 - sinh, 14
 - square root, 13
 - tangent, 14
 - tanh, 14
- MATLAB, 3, 4, 151
- Matrices, 151, 213
- determinant, 215
 - eigenvalues, 216
 - inverse, 215
 - norm, 216
 - operations between, 214
 - positive definite, 223
 - symmetric, 214
 - transpose, 214
- Message Passing Interface (MPI), 165, 167
- Allgatherv, 179
 - Allreduce, 178
 - Barrier, 174
 - Bcast, 178
 - broadcast, 178
 - Bsend, 173
 - collective communication, 174
 - broadcast, 174
 - combined send and receive, 174
 - halo exchange, 175
 - reduction, 174
 - communication, 171
 - blocking, 173
 - buffered, 173
 - point-to-point, 171
 - compiling, 169
 - debugging, 186
 - executing, 170
 - remote machines, 170
 - gather, 179
 - Gather, 179
 - Ibsend, 173
 - input and output, 167
 - installing MPI, 167
 - Irecv, 173
 - Isend, 173
 - process, 167
 - rank, 167
 - Recv, 171, 173
 - reduce, 178
 - running code, 169
 - scatter, 179
 - Scatter, 179
 - Send, 171
 - Sendrecv, 175
 - Ssend, 173
- Modularity, 2, 88, 89
- Modules, 82, 87, 88
- Multiple inheritance, 228
- N**
- Namespace, 227
- NOT, *see* Logical operators
- Not-a-number, 141
- Numerical libraries, 210
- PETSc, 211
- O**
- Object-orientation, 1, 87
- Object-oriented numerical libraries, 193
- ordinary differential equations, 200
 - partial differential equations, 207
- Operator overloading, 105, 106, 112, 151, 158, 159
- OR, *see* Logical operators
- Ordinary differential equations, 193, 194
- Output, 43
- appending to existing file, 46
 - formatting, 19
 - redirected to file, 43
 - scientific notation, 51
 - setting precision of, 46, 51
 - to console, 19
 - to file, 44
- Output stream variable, 44
- P**
- Partial differential equations, 193
- Pointer Aliasing, *see* Pointers
- Pointer de-reference, *see* Pointers
- Pointers, 55
- array of, 60
 - delete keyword, 57, 58, 60, 63, 102, 156, 157, 160, 226
 - dynamic allocation of memory, 58
 - new keyword, 57, 58, 60, 63, 102, 156, 157, 160, 226
 - pointer aliasing, 58, 62
 - pointer de-reference, 56
 - pointer variables, 56
 - shared pointers, 64
 - to classes, 103
- Polymorphism, 2, 118
- run-time, 122
 - static, *see* Templates
 - templates, *see* Templates
- Postcondition, 142

Precondition, 142
`printf`, 225, 226
Python, 3

R

Recursive functions, *see* Functions
Reference variables, 74
Relational operators, 29
 equal to, 29
 equality versus assignment, 37
 greater than, 29
 greater than or equal to, 29
 less than, 29
 less than or equal to, 29
 not equal to, 29
Reserved words, 11

S

Scientific notation, *see* Variables
Scope, 65
Shared memory architectures, 165
Shared pointers, *see* Pointers
`sprintf`, 226
Standard Template Library (STL), 131, 134
 abstraction, 134
 algorithm, 135
 container, 134
 deque, 134
 iterator, 135
 list, 134
 push_back, 135
 set, 137
 vector, 135
 vectors, 134
Statements, 5
`stdio.h`, 226
Strings, 18

Structures, 228
`switch`, 34

T

Templates, 131
 for polymorphism, 133
 Standard Template Library, *see* Standard
 Template Library (STL)
 validation of index to array, 132
Ternary `?:` operator, 226
Test-driven development, 146
`throw`, *see* Exceptions
`try`, *see* Exceptions
Type conversion, 15

U

Unified modelling language, 120

V

Variables, 10
 constant variables, 11
 declaring variables, 10
 global variable, 66
 initialising variables, 11
 local variable, 66
 scientific notation, 12
Vectors, 151, 213
 norm, 216
 operations between, 214
 scalar product, 215
Virtual methods, 122, 125
 pure virtual method, 125
`void`, 68

W

`while`, 25, 30