Chapter 1
Introduction

Soon after the publication of my C++ textbook, *A First Course in Computational Science and Object-Oriented Programming with C++*, in 2005, I conceived of including a yet more compact introduction to C++ in a survey of the entire field of scientific programming. Drawing on 20 years of experience of teaching programming at all levels in both physics and electrical engineering departments, I resolved to both summarize my previous treatment of C++ and incorporate a discussion of the Octave and Java programming languages, focusing on their conceptual foundations. Finally, I would insert many additional scientific programming examples, emphasizing short programs that illustrate key algorithms. By employing *only free software*, this would create a uniquely comprehensive treatment of the full set of steps from compiler installation to sophisticated scientific programming.

1.1 Objective

This textbook overviews modern scientific programming, including numerical analysis, object-oriented programming, scientific graphics, software engineering, numerical analysis and physical system modeling. Consequently, knowledge of the material will provide sufficient background to enable the reader to analyze and solve nearly all normally encountered scientific programming tasks.

1.2 Presentation

The text is concise, focusing on essential concepts. Examples are intentionally short and free of extraneous features. To promote retention, the book repeats key topics in cycles of gradually increasing difficulty. Further, since the process of learning computer language shares many similarities with that of acquiring a spoken language, *important code is highlighted in gray*. *Memorizing these features greatly decreases the time required to achieve proficiency in programming*.

1.3 Programming languages

Computing paradigms have evolved with time from the implementation of individual operations within a computing device to high-level structures that closely resemble interactions of physical objects with their environment. These concepts have simultaneously been realized through languages that have progressed from machine language to procedural, object-oriented and visual programming.

General-purpose procedural languages. A procedural language is composed of a structured sequence of commands, which may be further organized into modules termed functions or subroutines. A program implements a series of commands that are sequenced through logical statements. This strategy yields languages that are easily learned and applied. Especially early procedural languages such as FORTRAN, however, contain numerous unsafe constructs that invariably lead to coding errors.

Scientific procedural languages. To simplify small proof-of-principle computations, specialized scientific languages such as MATLAB® and Octave and symbolic manipulation languages such as MAPLE® or Mathematica® provide an easily learned high-level user interface to a unified built-in array of easily called and highly optimized numerical, scientific and graphical libraries. MATLAB code can be transformed into C++ through an add-on product while C++ and FORTRAN routines can be called by a MATLAB program with some effort. Additionally, MATLAB and similar programs originate from a single commercial source and therefore function nearly identically across all supported platforms (running the same version number). However, the suppression of advanced features such as classes, type-checking and user-controlled memory management can lead to structural confusion, programming errors and runtime inefficiency for larger problems. Further, the software is unavailable at many sites because of its substantial cost, although this can, however, increasingly be circumvented, through free software packages that imitate MATLAB commands. This textbook accordingly employs the most widely employed alternative, GNU Octave.

Object-oriented languages. The fundamental high-level unit in modern programming languages is an object. An object is a simplified model or *abstraction* of a particular entity. To illustrate, consider for definiteness a voltage meter. The meter has many attributes – in the extreme case the position and velocity of each atom – but only a few of these are typically of interest. These relevant attributes, which could include both user-accessible, **public**, data and behaviors, such as the voltage reading and the meter's response to depressing the power-on switch, and inaccessible, **private**, characteristics, such as the currents through individual circuit elements, compose the relevant abstraction of the object. By analogy, in a C++ program, **public** properties can be accessed throughout the code while **private** members are accessible only to functions that exist within the object itself, restricting the associated code region subject to inadvertent errors.

In an object-oriented language, objects with similar properties are described by a class. Two functions or variables with the same name that belong to different

classes are considered to be unrelated, circumventing name collisions. A class can incorporate the features of a second class by *inheriting* its non-**private** properties; the new, derived, class can then employ or redefine the properties of the original, base, class without recoding. Refinements to the original class automatically propagate to the new class.

Since additional programming syntax is required in order to create and manipulate objects and classes, object-oriented languages require more time to learn than procedural languages. However, the structure of the resulting program closely represents the physical objects that are being modeled. Accordingly, object-oriented development is advisable for large programs or programs that will be frequently revised.

The C++ language. C++, which extended the preexisting C procedural programming language, constituted the first widespread object-oriented language. Numerous scientific programming packages are at present available in C++, while FORTRAN programs can be, with some effort, accessed from C++, c.f. Appendix D of my companion textbook *A First Course in Computational Science and Object-Oriented Programming with C++*. However, the additional functionality of C++ enables manipulations that can introduce unexpected dependences among variables. To ascertain these dependences, C++ typically runs more slowly than FORTRAN, although advanced C++ language features can be employed to circumvent these difficulties, as discussed in Chapter 21 of the above reference.

Java: As a more recent object-oriented language, Java provides a far broader standard feature set than C++. Classes that e.g. handle graphics and internet communications are native to the language and in principle function identically across all Java implementations (although, in reality, version and machine dependences exist). Modern programming features such as multithreading and object serialization are additionally included. However, since the language is oriented toward the corporate market, many design choices are unfavorable for sophisticated scientific programming. For example, C and C++ contain high-level commands that enable direct access to hardware resources. These include addressing and modifying the contents of individual memory locations and precisely allocating and releasing the memory available to a program during execution. Since severe errors result if such manipulations are improperly performed, Java handles such operations automatically, at the cost of longer or unpredictable execution times. Further, extensive mathematical or scientific program libraries are ported only very slowly, if at all, to new programming languages.

1.4 Language standards

As requirements evolve, programming languages undergo periodic revision by a standards organization. For relatively new languages such as Java, revisions can be significant; further, existing language elements can become deprecated (unsupported) and eventually obsolete. In contrast, revisions to mature languages

such as C++ are relatively minor and do not affect core functionality. However, programs employing elements of a new standard will not necessarily function on older compilers.

1.5 Chapter summary

The organization of this book is as follows. After a short introduction to Octave in Chapter 2, the following chapters summarize the installation of a free C++ programming environment, computer hardware and software architecture and the basic structure and syntax of the C++ language. Chapter 7 introduces object-oriented programming in C++, with more advanced features of C++ following in Chapters 9–14. Chapters 15–17 discuss basic Java programming, with advanced Java features relegated to Chapter 18. Chapters 19–24 finally discuss applied numerical analysis in the context of numerous physical and engineering applications, including mechanics, electromagnetism, statistical mechanics, quantum mechanics and optics.

1.6 How to use this text

The reader is encouraged to follow the steps below.

(1) Skim through the text.
(2) Reread the chapter, programming and running as many sample programs in the text as possible. Attempt, if possible, to extend these programs.
(3) *Memorize the programs or program sections marked in gray in the text*. Success in programming is largely dependent on being able to recall instantly central language features.

1.7 Additional and alternative software packages

While comprehensive freeware and commercial C++ and Java numerical libraries exist, such as the GNU, CERN, IMSL and NAG libraries, such routines are typically designed for a restricted set of hardware and software platforms. Therefore, for smaller programs well-documented source code such as the programs in this book will often provide a more optimal trade-off between computational efficiency and development time.

Chapter 2
Octave programming

For small programs or rapid prototyping of ideas and methods, the commercial MATLAB® language, or its freeware alternatives, offers a practical alternative to C++ or FORTRAN. In this book, the free GNU Octave implementation is discussed from a scientific programming perspective. After becoming familiar with the central language constructs summarized below, the built-in Octave help facilities conveniently provide information on specialized, infrequent commands.

2.1 Obtaining octave

The Windows and Mac installation packages for GNU Octave are currently located at octave.sourceforge.net. Linux versions are available at the main Octave web site www.gnu.org/software/octave. When the program is installed, a variety of additional packages and the creation of a database of C++ components accessible by the editor can be selected. Unless space is an issue, these options should be chosen.

2.2 Command summary

(1) *Running Octave.* After clicking on the Octave icon, statements are entered interactively by typing into the resulting command window at the > prompt. An Octave session is terminated by typing **quit**.

(2) *System commands.* To change from the startup directory (folder) to the directory that either contains or will contain program files, type **cd X:\dir1\dir2\ ... \programDirectory**, where **X** is the partition (logical drive) containing the desired directory and **\dir1\dir2 ... \programDirectory** is an ordered sequence of the names of the directories enclosing the directory, **\programDirectory**, in which the program is located. If one or more directory names contain spaces, the entire expression containing these names must be surrounded by double apostrophes (**"**), e.g. **cd "X:\My Documents"**. Representative operating-system commands that can be issued from the Octave prompt include **mkdir directoryName**, which creates the directory **directoryName**, **rmdir directoryName**, which removes this directory,

dir or **ls**, which display the contents of a directory, **..**, which moves to one directory higher in the directory tree, **.**, which represents the current directory, **rename file1.1 file2.2**, which renames the file **file1.1** to the name **file2.2**, and **copy**, which similarly copies a file.

(3) *MS-DOS and Unix commands.* Standard DOS commands on Windows systems and Unix commands on Unix systems are issued in Octave by typing e.g. **dos 'copy file.1 file.2'** or, on a Unix system, **unix 'cp file.1 file.2'** (single or double apostrophe). In MATLAB such commands can also be preceded with **!**.

(4) *Command structure and continuation lines.* Octave commands end at a carriage return, comma or semicolon; however, only a semicolon suppresses the output of the statement from being written to the terminal. Two or more commands situated on the same line must be separated by commas or semicolons. A statement can span several lines but each line must normally be terminated by a three-period continuation character, **...**.

(5) *Creating and editing files.* The command **diary on** stores subsequent commands entered from the keyboard in a file named **diary** until **diary off** is issued. To examine this file or to create or edit an Octave program, after navigating to the directory in which the file resides, type **edit** at the command prompt, followed, where applicable, by the name of the file to be edited or created.

(6) *Comments.* Any text to the right of a comment character, **%**, constitutes a comment and is consequently ignored by the Octave interpreter. The beginning of a program should contain the date, version number, title and author. Every set of statements (a paragraph) performing a certain task should be *preceeded* by one or more blank lines followed by comment lines explaining the purpose of the program unit. When a variable is introduced, its meaning should be made clear by a comment either above the line or on the same line to the right of the statement. Such annotations insure the long-term readability of programs.

(7) *Using help*. To find and implement rarely employed commands, type first **lookfor subject** to obtain a list of all commands involving the operation 'subject'. Issuing **help commandName** (or **doc commandName**) then provides help on the command **commandName**.

(8) *Octave programs*. Octave program and function files must possess a **.m** extension; that is, to program in Octave, first type **edit** from within the Octave command window and create a file such as

```
s = 2;
s * ... % Illustrates the comment and continuation symbols
s
```

Then, when saving the file, specify **test** in the "File Name" text entry field while in the "Save File as Type" drop-down text box select MATrix LABoratory (.m). This automatically appends the correct **.m** extension to the file name. If the file is saved in a directory, e.g. **X:\testDirectory**, then, at the Octave prompt, type **cd** followed by the directory (including, if necessary, the partition name, e.g. **cd X:\testDirectory**)

where **test.m** resides, press enter and then type **test**. The program **test** can also be called from within another **.m** file within the same directory.

(9) *Variable-naming conventions.* For clarity, variable names should start with a small letter, while subsequent words in the name should be capitalized, e.g. **numberOfPoints**. However, in Octave a name can represent an array of any size and number of dimensions, which can result in subtle errors. To prevent this, quantities with a row dimension can be indicated by a trailing R, those with a column dimension as C and a matrix with a trailing RC, e.g, **systemMatrixRC**. If matrices with different dimensions are present, the row and column sizes can be further specified as in **systemMatrixR4C8**. *Since Octave is not a typechecked language, the above conventions can still lead to severe and difficult-to-locate errors for some compound words such as wavefunction, which can be treated as one word in certain places and as two words (waveFunction) in others. Typing a single character incorrectly generates similar problems. These errors, however, can be immediately identified by typing* **who** *at the command line, which displays a list of all the currently defined variables. Any spelling error will then be evidenced by a variable name seemingly appearing twice in the list*.

(10) *Formatting conventions.* Every binary operator (+, − etc.) should be surrounded by spaces, but not unary operators as in $3 + -4.0$. Indentation should be employed for every set of statements that are under the logical control of a control statement such as **for**, **if**, **while**. Commas, semicolons, parentheses and braces should, where appropriate, be followed by spaces.

(11) *Program input*. To prompt the user from within a **.m** file to enter a single variable or array **x** from the keyboard, employ **x = input('user prompt ')**. A variable **y** that can later be employed in a logical control statement to branch into different program units is conveniently entered with **y = menu('Select the method', 'Method A', 'Method B', 'Method C');** which assigns the value 1, 2 or 3 to **y** according to the user selection.

(12) *Output formatting.* The **more** command pages subsequent output. To write out subsequent floating-point output with 16 digits of precision, type in **format long e**, to revert to the default 5 digits, type **format short e** or, equivalently, **format compact**.

(13) *Built-in constants and functions*. Important predefined scalar quantities are **e**, **pi**, **i** and **j**, both of which represent the unit complex number, and **eps**, the smallest number which when added to 1 gives a number different from 1 (machine epsilon). However, a major problem arises if these variables are redefined, for example, the command **i = [1, 3]** overwrites the intrinsic definition of **i**, which is not reinstated until a further command **clear i** is issued. Note that **i** and **j** are frequently employed as loop variables, *so that all loop variables should instead be labeled, for example,* ***loop, innerLoop, outerLoop***.

(14) *Complex numbers*. A complex number is introduced as **c = 2.0 + 4.0i** and then manipulated with functions such as **real( )**, **imag( )**, **conj( )**, **norm( )**. Complex numbers are e.g. multiplied, divided and exponentiated in standard fashion either

by real or by other complex numbers. Functions such as **cos( )**, **sin( )**, **sinh( )** yield complex results when applied to complex quantities.

(15) *Loading arrays.* A variable name can represent a scalar or an array of any dimension. A row vector is introduced as

```
vR = [1 2 3 4];
```

or

```
vR = [1, 2, 3, 4];
```

while a column vector is entered, even from the keyboard, as

```
vC = [1
2
3
4];
```

or, equivalently (since a semicolon is largely equivalent to a carriage return),

```
vC = [1; 2; 3; 4];
```

With the transpose operator **.'** the above column vector can also be entered as

```
vC = [1 2 3 4].';
```

A matrix

$$mRC = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

can therefore be entered in any of the following ways:

```
mRC = [1 2; 3 4];
mRC = [1 2
3 4];
vR = [1 3 2 4]; mRC = reshape( vR, 2, 2 );
vC = [1
3
2
4]; mRC = reshape( vC, 2, 2 );
```

(The order in which a vector is reshaped indicates that matrix elements with successive values of the leftmost, column, index are stored next to each other in memory.) The matrix element $(\mathrm{mRC})_{12}$ is subsequently accessed by **mRC(1, 2)**. Since scalars and arrays are manipulated identically, arrays with multiple dimensions are constructed from component vectors or from subarrays in the same manner as from scalar quantities, e.g. **vR4 = [[1 2] [1 2]]** yields **[1 2 1 2];**, while **mBlockRC = [mRC mRC; mRC mRC];** is a matrix of twice the dimension of **mRC**. While a vector or matrix expands dynamically as new elements are added as in **vR = [1 2]; vR(3) = 3;** this is computationally inefficient and memory should instead be preallocated through a statement such as **vR = zeros(1, n)**, which creates a row vector of **n** zero elements.

(16) *Size and length*. Octave maintains a record of the size of an array to prevent element access outside this range. Hence **mRC(1, 3)** yields an error if mRC is a 2×2 matrix. The command **size(mRC)** for an $M \times N$ matrix returns the array $[M, N]$. For a single-dimension array, **length(vR)** returns the length of the array **vR**. However, when applied to a two-dimensionsal array **length(mRC)** returns the maximum value of M and N, possibly leading to unexpected errors.

(17) *Matrix operations*. The $n \times n$ identity matrix is represented by **eye(n)** or **eye(n, n)**, while the $n \times n$ matrix with all unity elements is denoted **ones(n)** or **ones(n, n)**. If **s = 2** and **mRC = [1 2; 3 4]** as in item (15) above, then

$$\mathrm{s} + \mathrm{mRC} = \mathrm{s} * \mathrm{ones}\,(2,2) + \mathrm{mRC} \Rightarrow \begin{pmatrix} 3 & 4 \\ 5 & 6 \end{pmatrix}$$

while

$$\mathrm{s} * \mathrm{eye}\,(2,\,2) + \mathrm{mRC} \Rightarrow \begin{pmatrix} 3 & 2 \\ 3 & 6 \end{pmatrix}$$

Very often, errors arise because of failure to differentiate between these. Multiplication similarly possesses different meanings depending on variable type. Multiplying or dividing a matrix **mRC** by a scalar **s** multiplies (divides) all elements of **mRC** by **s** while **mRC * mRC** symbolizes normal matrix multiplication and

$$\mathrm{mRC}.* \mathrm{mRC} \Rightarrow \begin{pmatrix} 1 & 4 \\ 9 & 16 \end{pmatrix}$$

implements component-by-component multiplication. Similarly **mRC^2** is **mRC * mRC**, while **mRC.^2** instead squares the individual elements of **mRC**. The dot operator functions analogously for other arithmetic operations such as **mRC ./ nRC**, which yields a matrix whose (i, j)th element is simply $(\mathrm{mRC})_{ij}/(\mathrm{nRC})_{ij}$. Standard functions such as **cos(mRC)** operate on the individual elements of **mRC**, here returning the matrix formed by taking the cosine of each element. *Two easily confused operations are array (matrix) transpose without complex conjugation*, **.'**, *and transpose with complex conjugation*, **'**. Note that the simpler syntax is applied to the Hermitian conjugate operation, since this yields the standard norm of complex (as well as real) arrays. For a vector **vR = [1, 2]** with elements $(\mathrm{vR})_{ij}$, the dot or inner product without complex conjugation is given by **vR * vR.'**, while the (Kronecker) outer product **vR.' * vR** yields the 2×2 matrix with (i, j)th element $(\mathrm{vR})_i\ (\mathrm{vR})_j$, namely **[1 2; 2 4]**.

(18) *Matrix functions*. A few functions ending in the letter m such as the matrix exponential **expm** act on matrix arguments and are defined (although not implemented) through power-series expansions such as

```
expm( aRC ) = eye( size( aRC ) ) + aRC + aRC^2/2!
+ aRC^3/3! + ...
```

The determinant, trace, inverse, logarithm and square root of **aRC** are similarly given by **det(aRC)**, **trace(aRC)**, **inv(aRC)**, **logm(aRC)** and **sqrtm(aRC)**,

respectively. The LU decomposition of aRC is expressed as **[lRC, uRC] = lu(aRC);**. The eigenvalues, arranged in ascending order, and the corresponding eigenvectors of a matrix **aRC** are placed, respectively, in the columns of the matrix **mVecRC** and the diagonal elements of **mValRC** through the call **[mVecRC, mValRC] = eig(aRC);**. Simply calling **eig(aRC)** returns a vector containing the eigenvalues in ascending order.

(19) *Solving linear equation systems*. The quotient of two matrices in Octave written as **mRC / nRC** denotes **mRC * inv(nRC)**, while **mRC \ nRC** instead represents **inv(mRC) * nRC**. Accordingly, the linear equation system **xR * mRC = yR** is solved by **xR = yR / mRC**, while **xC =mRC \ yC** if **mRC * xC = yC**.

(20) *Sparse matrices.* Operations on matrices with few non-zero elements are accelerated if the matrices are implemented as sparse matrices. The simplest procedure converts a full matrix **aRC** to a sparse matrix by **bRCsp = sparse(aRC)** (which is reversed with **aRC = full(bRCsp)**). Subsequent operations such as * and / are performed with sparse routines if all operands or arguments are sparse (except for an identity or zero matrix). **spy(mRCsp)** displays the locations of the non-zero elements of **mRCsp** while **speye(n)** is an $n \times n$ sparse identity matrix.

(21) *Random-number generation.* To compare the different versions of a program that incorporates a random-number generator, the random sequence should be the same in all versions. This is accomplished by introducing the statement **rand('state', 0)** at the beginning of each program. Uniformly distributed random numbers between 0 and 1 are generated singly with **rand** or as a multidimensional array with e.g. **rand(m, n)**. Typing **lookfor rand** displays information about functions for other random distributions.

(22) *Saving variables*. A variable **v** is stored in the text Octave file **filename** through **save filename v** and then recovered through **load filename**. This file can then be inspected or edited with any text editor. All variables present in the workspace are simultaneously saved and loaded in through the commands **save filename** and **load filename**. [In MATLAB the **save** command instead writes to a binary file **filename.mat** and will only save a variable **v** to an ASCII file **vdata** if the command **save vascii v –ascii** is instead employed. The variable is in this case recovered from the **.mat** file with **load vascii; v = vascii;** where only the first statement is employed if **v** is employed for the file name in place of **vascii**. However, retrieval from MATLAB binary files that store more than one variable presents difficulties since the original variable names are not stored as in **.mat** files.]

(23) *String manipulation.* A string such as **s = 'test'** is stored as an array ['t' 'e' 's' 't'] of characters so that **s(1)** returns **t** and **['a ', s]** yields the new string **'a test'**. Integers and floating-point numbers are translated into strings through the functions **int2str( )** and **num2str( )**, respectively; the reverse operation is performed by **str2int( )** and **str2num( )**. An Octave expression, expressed as a string, is sent to the command processor by writing **eval( )**; as an example, to display the files present in the directory, the commands **s = 'dir'; eval(s);** can be employed.