Chapter One

Introduction

1.1 NATURE OF SCIENTIFIC COMPUTING

Computational scientists solve tomorrow's problems with yesterday's computers; computer scientists seem to do it the other way around.

—anonymous

The goal of scientific computing is problem solving. The computer is needed for this, because real-world problems are often too difficult or complex for analytic or human solution, yet workable with the computer. When done right, the use of a computer does not replace our intellect, but rather leverages it by providing a super-calculating machine or a virtual laboratory so that we can do things that were heretofore impossible.

The mathematical modeling and problem-solving orientation of scientific computing places it in the discipline of *computational science*. In contrast, computer science, which studies computers for their own intrinsic interest, provides the underpinning for the development of the hardware and software tools that computational scientists use. As illustrated on the left of Figure 1.1, computational science is a *multidisciplinary* field that combines a traditional discipline, such as physics or finance, with computer science and mathematics, without ignoring the rigor of each. Books such as this, which employs materials from multiple fields, aim to be the central bridge in this figure, connecting and drawing together the three fields.

Studying a multidisciplinary field is challenging. Not only must you learn more than one discipline, you must also work with the separate languages and styles of the different disciplines. To illustrate, a computational scientist may be pleased with a particular solution because it is reliable, self-explanatory, and easy to run on different computers without modification. A computer scientist may view this same solution as lengthy, inelegant, and old-fashioned. Each may be right in the sense that they are making judgments based on the differing values of different disciplines.

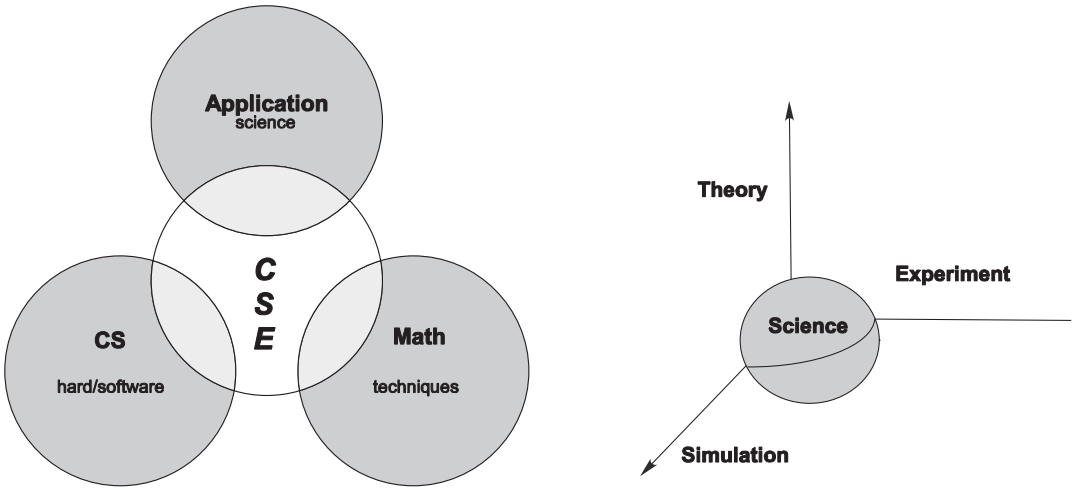

Figure 1.1 *Left:* Computational science is a multidisciplinary field that combines science with computer science and mathematics. *Right:* A new paradigm for science in which simulation plays as essential a role as does experiment and theory.

Another, possibly more fundamental, view of how computation is playing an increasingly important role in science is illustrated on the right of illustrated in Figure 1.1. This symbolizes a paradigm shift in which science's traditional foundation in theory and experiment is extended to include computer simulation. While one may argue that the future will see the CSE on the left of Figure 1.1 get absorbed into the individual disciplines, we think all would agree that the simulation on the right of Figure 1.1 will play an increasing role in science.

1.2 TALKING TO COMPUTERS

As anthropomorphic as your view of computers may be, it is good to keep in mind that a computer always does exactly as told. This means that you should not take the computer's response personally, and also that you must tell the computer exactly what you want it to do. Of course programs can be so complicated that you may not care to figure out what they will do in detail, but it is always possible in principle. Thus it follows that a basic goal of this book is to provide you with enough understanding so that you feel well enough in control, no matter how illusionary, to figure out what the computer is doing.

Before you tell the computer to obey your orders, you need to understand that life is not simple for computers. The instructions they understand are in a *basic machine language*[1] that tells the hardware to do things like move a number stored in one memory location to another location, or to do some simple, binary

[1]The "BASIC" (Beginner's All-purpose Symbolic Instruction Code) programming language should not be confused with basic machine language.

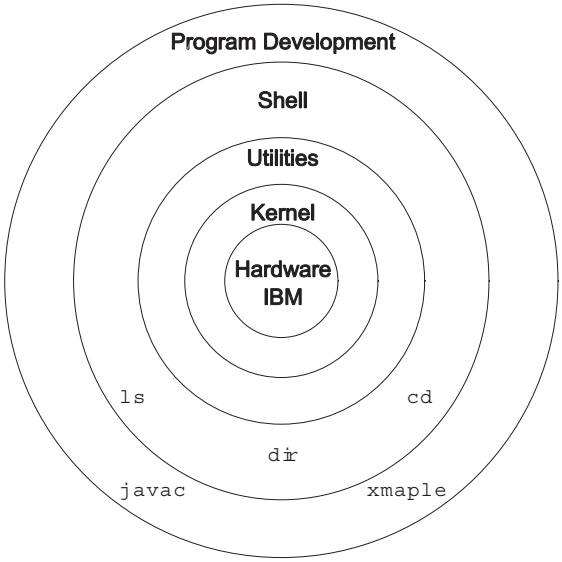

Figure 1.2 A schematic view of a computer's kernel and shells.

arithmetic. Hardly any computational scientist really talks to a computer in a language it can understand. Instead, when writing and running programs, we usually talk to the computer through a *shell* or in a *high-level language*. Eventually these commands or programs all get translated to the basic machine language.

A *shell* is a name for a command-line interpreter, that is, a place where you enter a command for the computer to obey. It is a set of medium-level commands or small programs, run by the computer. As illustrated in Figure 1.2, it is helpful to think of these shells as the outer layers of the computer's *operating system*. While every general-purpose computer has some type of shell, usually each computer has its own set of commands that constitute its shell. It is the job of the shell to run various programs, compilers, and utilities, as well as the programs of the users. There can be different types of shells on a single computer, or multiple copies of the same shell running at the same time for different users. The nucleus of the operating system is called, appropriately, the *kernel*. The user seldom interacts directly with the kernel, but the kernel interacts directly with the hardware.

The *operating system* is a group of instructions used by the computer to communicate with users and devices, to store and read data, and to execute programs. The operating system itself is a group of programs that tells the computer what to do in an elementary way. It views you, other devices, and programs as input data for it to process; in many ways it is the indispensable office manager. While all this may seem unnecessarily complicated, its purpose is to make life easier for you by letting the computer do much of the nitty-gritty work that enables you to think

higher-level thoughts and communicate with the computer in something closer to your normal, everyday language. Operating systems have names such as *Unix, OSX, DOS*, and *Windows*.

In Part 1 of this book we will use a high-level *interpreted* language, either Maple or Mathematica. In Part 2 we will use a high-level *compiled* language, either Java or Fortran90. In an interpreted language the computer translates one statement at a time into basic machine instructions. In a compiled language the computer translates an entire program unit all at once. Compiled languages usually lead to faster programs, permit the use of vast libraries of subprograms, and tend to be portable. Interpreted languages appear to be more responsive, interactive, and, consequently, more "user friendly."

When you submit a program to your computer in a compiled language, the computer uses a *compiler* to process it. The compiler is another program that treats your program as a foreign language and uses a built-in dictionary and set of rules to translate it into basic machine language. As you can imagine, the final set of instructions is quite detailed and long, especially after the compiler has made several passes through your program to translate your convoluted logic into fast code. The translated statements ultimately form an *executeable* code that runs when loaded into the computer's memory.

1.3 INSTRUCTIONAL GUIDE

Landau's Rules of Education: Much of the educational philosophy applied in this book is summarized by these three rules:

1. Most of education is learning what the words mean; the concepts are usually quite simple once you understand what you are being told.
2. Confusion is the first step to understanding.
3. Traumatic experiences tend to be the most educational ones.

This book has an attitude, and we hope you will develop one too! We enjoy computing and relish the increased creativity and productivity resulting from powerful computing tools. We believe that computing has so become part of the fabric of science that an introductory scientific computing course should be part of every lower-division university student's education. Hence we deliberately mix the languages of mathematics, science, and computer science. This mix of languages is how modern scientists think about things, and since ideas must be communicated with these same words and ideas, this is how the book is written. However, we are sensitive to the confusion multiple definitions may reap and do provide a section at the end of each chapter indicating some key words and concepts, and a glossary at the end of the book defining many technical terms and jargon.

Because we aim to give an introduction to computational science, we cover some basics of numerical analysis, information about how data are stored, and the concordant limits of computation. However, we present very little discussion of hardware, computer architecture, and operating systems. That is not to say that these are not interesting and important topics, but rather that we want to get the reader busy computing and acquiring familiarity with concrete examples. In our experience, science and engineering students need this practical experience before they can appreciate the more abstract principles of computer science.

We have aimed our presentation at first- and second-year college students. We believe that the chapters and sections not marked optional with an asterisk * provide a good introductory course for them. The inclusion of the optional chapters would raise of the level of the presentation and lead to a longer course. To maintain the logical organization of materials, we have intermixed the optional chapters with the others. However, there should be nothing in the optional chapters that is required in order to understand the chapters that follow.

It is hard to keep up with the rapid changes in computer technology and with the knowledge of how to use them. That is not such a big issue with scientific computing, because it is the basic principles of mathematics, logic, science, and computing that are important, and not the details of hardware and software. Nevertheless, students and faculty often do not enjoy computing because of the frustration and helplessness they feel when computers do not work the way they should. Although we commiserate with those feelings, one of the things we try to teach is that it is not unusual to have new things not work quite right, but that if you relax and follow a trial-and-error approach, then you usually find success working around them.

Be warned, there are some exercises in this book that give the wrong answer, that lead to error messages, or that may "break" the program (but not harm the computer). Learning to cope with the limits of computation gets easier and less traumatic after you have done it a few times. We understand, however, that many instructors may not appreciate a computer's failure that they cannot explain, and so we have assembled an *Instructor's Survival Guide*, available upon request through the Web.

It is important that students become familiar with the material in the text before they come to lab. A lecture helps, but reading and working through the materials is essential. Even though it may not be that hard to work through the problems in lab by having instructors and fellow students prompt you with the appropriate commands to enter, there is little to be gained from that. Our goal is to make this introductory experience very much a "lab" that develops an attitude of experimentation and discovery and that nurtures rewarding feelings when a project finally computes correctly.

Many of the problems we assign require numerical solution and therefore are not covered in elementary texts. These include nonlinear oscillators, the motion of projectiles with drag, and the rotation of cubes about an arbitrary axis. Notwithstanding our prediction that other elementary courses and texts will eventually be more multidisciplinary, some readers may feel that we are requiring them to understand phenomena at a level higher than in their other courses. Our hope is that the readers will recognize that they are pioneers, will be stimulated by their new-found powers, and will help modernize their other courses.

On the technical side, we have developed the materials with Maple 6–9.5, Mathematica 4.2, and the Java Development Kit (JDK or Java 2 Standard Edition, J2SE) [SunJ]. Even though studio and workbench environments are available, we prefer the pedagogical value in using a shell to issue separate commands for compilation and execution, and then having to deal with the source and class files directly. In addition, many students are able to load JDK onto their home computers and work there as well.

We have found that *WinEdt* [WinEdt] and *TextPad* work well to edit and run source code on a Windows platform, and that *Xemacs* [Gnu] with Java tools is excellent for Unix/Linux machines. In addition, *jEdit*, the Open Source programmer's editor [jEdit], is an excellent tool that, because it itself is written in Java, runs on most any platform. Visualization is very important in computational science. It is built into Maple and Mathematica and is excellent. Part of the power of Java is that it has strong graphical capabilities built right into the language, although calling them up is somewhat involved. Consequently, we have adopted the free, open source package *PtPlot* [PtPlot] as our standard approach to plotting with Java. However, *gnuplot* [Gnuplot] and *Grace* [Grace] are also recommended as stand-alone applications. For 3-D graphics, a more specialized application, we give instructions on using *gnuplot* and refer the interested reader to *OpenDx* [DX] and *VisAd* [Visad]. These too are free, powerful, and available for Unix/linux and Windows computers.

1.4 EXERCISES TO COME BACK TO

Consider the following list of problems to which a computational approach could be applied. Indicate with an "M," "J," or "E" whether the best approach would be the use of Maple, Java, or either.

1. calculate the escape velocity from Jupiter
2. write a spreadsheet (accounting) program from scratch
3. solve problems from a calculus textbook
4. prove an algebraic identity
5. determine the time required for a sky diver to reach terminal velocity
6. write a compiler for a programming language