Computational Physics

Problem Solving with Computers

RUBIN H. LANDAU
Professor of Physics
Oregon State University

MANUEL JOSÉ PÁEZ MEJÍ A
Professor of Physics
University of Antioquia

Contributors
Hans Kowallik and Henri Jansen

A Wiley-Interscience Publication
JOHN WILEY & SONS, INC.
New York / Chichester / Weinheim / Brisbane / Singapore / Toronto

1 Introduction

1.1 THE NATURE OF COMPUTATIONAL SCIENCE

Computational science explores models of the natural and artificial world with the aim of understanding them at depths greater than otherwise possible. This is a modern field in which computers are used to solve problems whose difficulty or complexity places them beyond analytic solution or human endurance. Sometimes the computer serves as a super–calculating machine, sometimes as a laboratory for the numerical simulation of complex systems, sometimes as a lever for our intellectual abilities, and optimally as all of the above.

The focus of a computational scientist is science. The aim of this book is to teach how to do science with computers, and, in the process, to teach some physics with computers. This is computational science but not "computer science." Computer sciences studies computing for its own intrinsic interest and develops the hardware and software tools computational scientists use. This difference is not just semantic or academic. Computational scientists are interested in computer applications in science and engineering, and their values, prejudices, tools, organizations, goals, and measures of success reflect that interest. For example, a computational scientist may view a particular approach as reliable, self-explanatory, and easy to port to sites throughout the world, while a computer scientist may view it as lengthy and inelegant; both are right, because both are viewing it from their different disciplines.

Computational science is a team sport. It draws together people from many disciplines via a commonality of technique, approach, and philosophy. A computational scientist must know a lot about many things to be successful.

But because the same tools are used for many problems in different fields, he or she is not limited to one specialty area. A study of computational science helps broaden horizons, which is a welcome exception to the stifling subspecialization found in so much of science.

Traditionally, physics divides into experimental and theoretical approaches; computational physics requires the skills of both and contributes to both. Transforming a theory into an algorithm requires significant theoretical insight, detailed physical and mathematical understanding, and mastery of the art of programming. (The sections in this book are labeled to reflect these steps.) The actual debugging, testing, and organization of scientific programs is like an experiment. The simulations of nature with programs are virtual experiments. Throughout the entire process, the synthesis of numbers into generalizations, predictions, and conclusions requires the insight and intuition common to both experimental and theoretical science. And as visualization techniques advance,[1] computational science enters into and uses psychology and art; this, too, makes good science because it reveals the beauty contained within a theoretical picture of nature and permits scientists to use extensive visual processing capabilities of their brains to "see" better their discipline.

1.1.1 How Computational Scientists Do It

A computational scientist uses computers in a number of distinct ways, with new ways not necessarily eliminating old ones.

- In the classic approach, a scientist formulated a problem and solved as much as possible analytically. Only then was the computer used to determine numerical solutions to some equations or to evaluate some hideously complicated functions. In many cases, computing was considered a minor part of the project with little, if any, discussion of technique or error.
- A computational scientist formulates and plans the solution of a problem with the computer and program libraries as active collaborators. Use is made of past analytic and numerical advances during all stages of work. And, as the need arises, new analytic and numerical studies are undertaken.
- In a different, but by now also classic scientific approach, computers play a key role from the start by *simulating* the laws of nature. In these simulations, the computer responds to input data as a natural system might to different initial conditions. Examples are the computer tracing of rays through an optical system and the numerical generation of random numbers to simulate the radioactive decay of nuclei.

[1]There are Web tutorials describing visualization and animation techniques.

- Another modern use of computers is to create *problem-solving environments*, such as *Maple, Mathematica, Macsyma,* and *Matlab*, which hide most of the details from the user and which often include symbolic manipulations as might be done analytically.
- One of the most rewarding uses of computers is *visualizing* the results of calculations with 2-D and 3-D plots or pictures, and sometimes with color shading and animation. This assists the debugging process, the development of physical and mathematical intuition, and the enjoyment of the work. Visualization is incorporated into as many of our projects as possible and especially in the Web tutorials associated with this book.
- Finally, many personal computer applications also have value in computational science. For example, a numerical *spreadsheet* is a helpful way to analyze data as well as the results of calculations, and *hypertext* and World Wide Web documents are true advances in storing various types of information that supplement, even if they do not replace, the lab notebook and research paper.

1.2 AIMS OF THIS BOOK

To emphasize our general purpose of teaching how to do science with computers, the paradigm suggested by the Undergraduate Computational Science and Engineering Project [UCES] will be followed:

Problem	Model	Method	Implementation
(Physics)	(Discrete)	(Symbolic)	(C/Fortran)
(Life science)	(Continuous)	(Numeric)	(High-performance)

↕

Assessment
(Visualization)
(Experimentation)

This is not easy to do when developing basic skills, but it will work well once projects deal with physical problems.

When the students are relieved of the burden of extensive programming, they should be able to "pass lightly" through the background material and have a personal experience with many projects. This personalization of the material acts as stimulation for further study, discussion, and exploration.

The specific aims of the projects are

- To teach the use of scientific computers in thinking creatively and solving problems in the physical sciences through direct experience.

- To advance the development and organization of thinking about physical systems in a manner compatible with advanced computational analysis.
- To use the graphic capabilities of scientific computers to study and teach the visualization of numerical solutions into highly interpretable forms.
- To instill attitudes of independence, personal communication, and organization, all of which are essential for mastery of complex systems.
- To understand physical systems at a level often encountered only in a research environment, and to use programming to deepen that understanding.
- To understand why hard work and properly functioning and powerful software and hardware do not guarantee meaningful results. As with experimental physics, there are accuracy and applicability limits that often determine when viable results are generated.
- To instill an *objected-oriented* view of problem solving.

1.3 USING THIS BOOK WITH THE DISK AND WEB

There are references throughout this book to programs and tutorials available on the floppy diskette accompanying the text and through the World Wide Web (the "Web"). These are meant as a supplement to the text, to be used at the discretion of the student and instructor.

Programming is a valuable skill for all scientists, but it is also incredibly demanding and time consuming. For this reason the diskette and two appendixes provide both C and Fortran programs as the basic implementation part for most of the **Problems**. It is suggested that the student read through the given programs and modify or rewrite them for the project at hand. Not only will this save time, but it is a valuable lesson in learning how to read someone else's code (real-world scientists seldom have the luxury of writing their own). Note, the C and Fortran programs are not direct translations of each other, and for some problems a program in only one language is provided. We provide an appendix that tabulates analogous elements in the C and Fortran languages. This should help those readers having to struggle with a foreign language.

Most of the **problems** we examine can also be worked in a problem solving environment such as *Maple, Mathematica, Matlab*, or *Mathcad*. If you use those packaged systems, you may not learn the same programming skills, your program may be less flexible, and they may be much slower; but then again, you may end up being able to spend more time understanding the science and mathematics.

Referring to as rapidly changing a resource as the Web in a textbook is somewhat risky, yet it also adds a new dimension that is just too good to pass up. References in this book to the Web are primarily to the resources maintained by the Northwest Alliance for Computational Science and Engineering and the Undergraduate Computational Science and Engineering Project:

Computational Science Web Sites	
NACSE	http://www.nacse.org
UCES	http://www.krellinst.org/UCES/

As a research project aimed at better incorporating the techniques of high-performance computing into science, these two groups have supported the conversion of some of the computational physics projects in this book into interactive Web tutorials. On the Web you will find running codes, figures, animations, sonifications, corrected code listings, and control-panel interfaces. While these are not meant to be a substitute for studying the text or for your running you own codes, they provide some stimulating examples of what can de done and of how the physics can be "seen" in differing ways. For example, not only can you see coordinate- and phase-space plots of a chaotic pendulum, but you can actually see the pendulum swing and hear the oscillations!

Additional Web resources of interest are given by a Computational Physics Resource Letter [DeV 95] and by the list of URLs (universal resource locators) on Landau's home page. Particularly recommended are the Web sites of the U.S. National Science Foundation Supercomputer Centers [NSF].

2

Computing Software Basics

In this chapter we explore basics of computing languages, number representation, and programming. Related topics dealing with hardware basics are found in Chapter 18, *Computing Hardware Basics: Memory and CPU*. We recommend that you glance through Chapter 18 now. If you find that you really have no idea what it's about, then you would benefit by studying it as soon as possible, and especially before getting involved in heavy-duty computing.

2.1 PROBLEM 1: MAKING COMPUTERS OBEY

You write your own **program**, wanting to have the computer work something out for you. Your **problem** is that you are beginning to get annoyed because the computer repeatedly **refuses** to give you the correct answers.

2.2 THEORY: COMPUTER LANGUAGES

As anthropomorphic as your view of your computer may be, it is good to keep in mind that computers always do exactly as told. This means that you must tell them exactly and everything they have to do. Of course, the programs you write may be so complicated and have so many logical paths that you may not have the endurance to figure it out in detail, but it is always possible in principle. So the real **problem** addressed in this chapter is how to give you 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 arithmetic. Hardly any computational scientist really talks to a computer in a language it can understand. When writing and running programs, we usually talk to the computer through *shells* or in *high-level languages*. Eventually these commands or programs all get translated to the basic machine language.

A *shell* (*command-line interpreter*) is a set of medium level commands or small programs, run by a computer. As illustrated in Fig. 2.1, 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, linkage editors, 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.

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 to enable 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, VMS, MVS, DOS,* and *COS*.

We will assume you are using a *compiled* high-level language like *Fortran* or *C*, in contrast to an *interpreted* one like *BASIC* or *Maple*. In a compiled language the computer translates an entire subprogram into basic machine instructions all at one time. In an interpretive language the translation is done one statement at a time. Compiled languages usually lead to more efficient programs, permit the use of vast libraries of subprograms, and tend to be portable.

When you submit a program to your computer in a high-level language, the computer uses a *compiler* to process it. The compiler is another program that

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

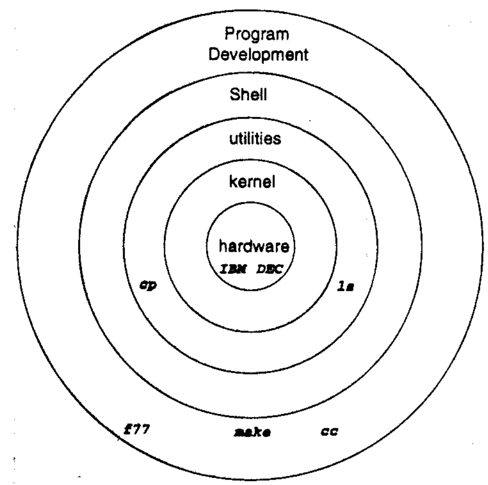

Fig. 2.1 A schematic view of a computer's kernel and shells.

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 are quite detailed and long, and the compiler may make several passes through your program to decipher your convoluted logic and to translate it into a fast code. The translated statements form an *object* code, and when *linked* together with other needed subprograms, form a *load module*. A load module is a complete set of machine language instructions that can be *loaded* into the computer's memory and read, understood, and followed by the computer.

2.3 IMPLEMENTATION: PROGRAMMING CONCEPTS

Before we discuss general programming techniques, we need to be sure that you can talk to your computer. Here is a tutorial to get you communicating. Begin by assuming that calculators have not been invented and you need a program to calculate the area of a circle. Rather than using any specific language, we will discuss how to write that program in *pseudocode* that can be converted to your favorite language later. The first program tells the computer:[2]

```
calculate area of circle                         Do this, computer!
```

[2] Comments placed in the field to the right are for you and *not* for the computer to view.

This program cannot really work because it does not tell the computer which circle to consider and what to do with the area. A better program would be

```
read radius                                    Input
calculate area of circle                    Numerics
print area                                    Output
```

The instruction *calculate area of circle* has no meaning in most computer languages, so we need to specify an *algorithm*[3] for the computer to follow:

```
read radius                                    Input
calculate area of circle                     Comment
π = 3.141593                            Set constant
area = π× radius²                      The algorithm
print area                                    Output
```

This is a better program. When we cannot think of any more embellishments, we convert this pseudocode to a language the computer can understand.

2.4 IMPLEMENTATION: FORTRAN, AREA.F

A Fortran version of our *area* code is found on the disk and in Appendix D under the name *area.f* (we usually indicate the appropriate program name in the title of Implementation sections, as you may note here). Because beginnings are so hard, we will be nice to you this time and list[4] the program here:

```
      Program area                    Tell compiler it's a main program
c                                             Space helps readability
c     area of circle, input r                     Say what's happening
      Double Precision pi, r, A                 Uppercase for clarity
c     calculate pi                                            Comment
       pi = 3.141593                                    Set value of π
c     Read r from standard input (terminal)
       Write(*,*) 'specify radius'                Appears on terminal
       Read (*, *) r                              Input from terminal
c     calculate area
       A = pi * r**2
c     Write area onto terminal screen
      Write (*,10) 'radius r =', r, ' A =', A        * for terminal
   10 Format (a20, f10.5, a15, f12.7)
       Stop 'area'                     Stop program and write 'area'
       End
```

[3]An *algorithm* is a set of rules for doing mathematics.
[4]Beware, our typeset spaces may not be perfect. In Fortran, comments usually have a c or in column 1, statement numbers must be in columns 2–5, continuation characters must be in column 6, and executable statements begin in column 7 (or higher).

Notice that the variable and program names are meaningful and similar to standard nomenclature (even with an uppercase A), there are plenty of comments, and the input and output are self-explanatory.

2.5 IMPLEMENTATION: C, AREA.C

A C version of our area program is found on the disk and in Appendix C under the name *area.c* (we usually indicate the appropriate program name in the title of Implementation sections, as you may note here). Because beginnings are so hard, we will be nice to you this time and list the program here:

```
/* Calculate area of a circle */      A comment, for reader only
                                                    A blank line
#include  <stdio.h>                   Need standard I/O routines
#define pi 3.14159265369                         Define constant
main()                         Tell compiler it's a main program
{                                                  Begin program
double r, A;                         Double-precision variables
printf("Enter the radius of a circle \n");         Request input
scanf("%lf", &r);                        Read from standard input
A = r * r * pi;                                   Calculate area
printf("radius r= %f, area A = %f\n", r, A);       Print results
}                                                    End program
```

2.6 IMPLEMENTATION: SHELLS, EDITORS, AND PROGRAMS

1. To gain some experience with your computer system, enter one of the preceding programs into a file. Then
   (a) Compile and execute it (in one command).
   (b) Check that the results are correct. Good input datum for testing is $r = 1$, because then $A = \pi$.
   (c) Try $r = 2$ and see if the area increases by a factor of 4. Then experiment (e.g., see what happens if you leave off decimal points, if you feed in blanks, if you feed in a letter, . . .).
2. The programs given here take input from and place output on the terminal screen. Revise one of these programs so that the input and output come from and are placed into two separate files.
3. Revise this program so that it uses a main program (which does the input and output) and a subroutine (which does the calculation). Check that it still runs properly.

2.7 THEORY: PROGRAM DESIGN

Now that you have warmed up on the computer, let's get back to the theory that should be behind your actions. Even with a perfect set of physical laws, a perfect algorithm, and a perfect computer, there still remains the challenge of *programming*. Programming is viewed as a written art that blends elements of science, mathematics, and computer science into a set of instructions so that the computer can accomplish a scientific goal (for example; generating the cross section for the scattering of an electron from a krypton atom). Sooner or later, a scientist who wants to do something new or different has to write his or her own programs. Computational scientists who place a high value on collaboration with other people, as well as making contributions to the development of science, write programs that

- Are simple and easy to read, making the action of each part clear and easy to analyze. (Just because it was hard for you to write that program, doesn't mean that you should make it hard for others to read.)
- Document themselves so that the programmer and others understand what the programs are doing.
- Are easy to use.
- Are easy and safe to modify for different computers or systems.
- Can be passed on to others to use and further develop.
- Give the correct answers.

The lack of program readability leads to credibility problems and the stifling of creativity. It is in the interests of the science to write clear programs even for the complicated problems encountered in modern science and engineering. Keep in mind, the program is the ultimate documentation of a computational science project, and the human and economic savings in being able to reuse someone else's work is often tremendous.

True creative artists follow their own rules. Nonetheless, here are some suggested ideas for *modular* and *top-down* programming that may help you on the road to becoming a creative programmer:

1. A *modular* approach breaks up the tasks of a program into subprograms. In general, your programs will be clearer and simpler, and easier to write, if you make them modular. While you may be able to view small programs in a single glance, the complexity of hundreds or thousands of uninterrupted lines of code boggles the mind and makes a single-glance understanding impossible.[5]

[5]This may not be good for vectorization on a supercomputer, but you can always recombine the subprograms after they are debugged and running.

   (a) Write many small subprograms, each of which accomplishes limited tasks.
   (b) Give each subunit well-defined input and output that gets passed as arguments.
   (c) Make each subprogram reasonably independent of the others. You can then test them independently and use them again and again in other programs.
   (d) Do not become overzealous about writing subroutines. If a subroutine is very small and is often called, the overhead time for the calls may be relatively expensive. In that case, the compiler will optimize better if you combine often-called and related program units into one.

2. Put off as long as possible the actual writing of your program. Concentrate instead on clarifying, understanding, and defining the problem to be solved and the logic to be used.
3. Try to choose the most reliable and simple algorithm. Speed matters, but not if you get the wrong answers.
4. Be aware that an algorithm that is best for scalar architecture may not be best for parallel architecture.
5. A program that is clear and simple will usually end up being less buggy. While the clear program may take more time to write and run, this usually saves you time in the long run. More importantly, it may help a project reach a successful completion rather than being abandoned in frustration.
6. The planning of your program should be from *top down to bottom*. This means you first outline the major tasks of the algorithm, always keeping the big picture visible.
   (a) Arrange the major tasks in the order in which they need to be accomplished. This is the most basic outline.
   (b) Plan the details of each major task, making sure to break these tasks into subtasks (which may turn out to be subprograms or groups of subprograms). This will be the next level of complexity in your outline.
   (c) Continue breaking up your tasks into smaller ones until you are at the subroutine level.
7. Keep the flow through the program *linear*, as indicated in Fig.2.2, with a minimal amount of jumping around.[6] Avoid *go to*'s and especially computed *go to*'s.

[6]This principle is modified for a parallel computer where multiple, central processors work simultaneously on one problem.