

Chapter 1

Introduction

1.1 The need for computers in science

Over the last few decades, computers have become part of everyday life. Once the domain of science and business, today almost every home has a personal computer (PC), and children grow up learning expressions like “hardware,” “software,” and “IRQ.” However, teaching computational techniques to undergraduates is just starting to become part of the science curriculum. Computational skills are essential to prepare students both for graduate school and for today’s work environment.

Physics is a corner-stone of every technological field. When you have a solid understanding of physics, and the computational know-how to calculate solutions to complex problems, success is sure to follow you in the high-tech environment of the twenty-first century.

1.2 What is computational physics?

Computational physics provides a means to solve complex numerical problems. In itself it will not give any insight into a problem (after all, a computer is only as intelligent as its user), but it will enable you to attack problems which otherwise might not be solvable. Recall your first physics course. A typical introductory physics problem is to calculate the motion of a cannon ball in two dimensions. This problem is always treated without air resistance. One of the difficulties of physics is that the moment one goes away from such an idealized system, the task rapidly becomes rather complicated. If we want to calculate the solution with real-world elements (e.g., drag), things become rather difficult. A way out of this mess is to use the methods of computational physics to solve this linear differential equation.

One important aspect of computational physics is modeling large complex systems. For example, if you are a stock broker, how will you predict stock market performance? Or if you are a meteorologist, how would you try to predict changes in climate? You would solve these problems by employing Monte Carlo techniques. This technique is simply impossible without computers and, as just noted, has applications which reach far beyond physics.

Another class of physics problems are phenomena which are represented by nonlinear differential equations, like the chaotic pendulum. Again, computational physics and its numerical methods are a perfect tool to study such systems. If these systems were purely confined to physics, one might argue that this does not deserve an extended treatment in an undergraduate course. However, there is an increasing list of fields which use these equations; for example, meteorology, epidemiology, neurology and astronomy to name just a few.

An advantage of computational physics is that one can start with a simple problem which is easily solvable analytically. The analytical solution illustrates the underlying physics and allows one the possibility to compare the computer program with the analytical solution. Once a program has been written which can handle the case with the typical physicist's approximation, then you add more and more complex real-world factors.

With this short introduction, we hope that we have sparked your interest in learning computational physics. Before we get to the heart of it, however, we want to tell you what computer operating system and language we will be using.

1.3 Linux and C++

Linux

You may be accustomed to the Microsoft Windows or Apple MAC operating systems. In science and in companies with large computing needs, however, UNIX is the most widely used operating system platform. Linux is a UNIX-type operating system originally developed by Linus Torwald which runs on PCs. Today hundreds of people around the world continue to work on this system and either provide software updates or write new software. We use Linux as the operating system of choice for this text book because:

- Linux is widely available at no cost;
- Linux runs on almost all available computers;
- it has long-term stability not achieved by any other PC operating system;
- Linux distributions include a lot of free software, i.e., PASCAL, FORTRAN, C, C++.

In today's trend to use networked clusters of workstations for large computational tasks, knowledge of UNIX/Linux will provide you with an additional, highly marketable skill.

C++

In science, historically the most widely used programming language was FORTRAN, a fact reflected in all the mathematical and statistical libraries still in use the world over (e.g., SLATEC, LAPACK, CERNLIB). One disadvantage of FORTRAN has always been that it was strongly decoupled from the hardware. If you wanted to write a program which would interact directly with one of the peripherals, you would have to write code in assembly language. This meant that not only had you to learn a new language, but your program was now really platform dependent.

With the emergence in the late 1970s of C [2] and UNIX, which is written in C, all of a sudden a high level language was available which could do both. C allowed you to write scientific programs and hardware drivers at the same time, without having to use low level processor dependent languages. In the mid 1980s Stroustrup [3] invented C++, which extended C's capabilities immensely. Today C and C++ are the most widely used high level languages.

Having "grown up" in a FORTRAN environment ourselves, we still consider this to be the best language for numerical tasks (we can hear a collective groan in the C/C++ community). Despite this, we decided to "bite the bullet" and switch to C++ for the course work.

The GNU C/C++ compiler is an excellent tool and quite versatile. Compared to the **Windows** C++ compilers (e.g., Visual C++ [Microsoft] or Borland C/C++), the user interface is primitive. While the Windows compiler-packages have an extensive graphical user interface (GUI) for editing and compiling, the GNU compiler still requires you first to use a text editor and then to collect all the necessary routines to compile and link. One "disadvantage" to the Windows compiler packages is that many of them automatically perform a number of tasks necessary to building a program. You might be wondering how that could be a disadvantage. We have noticed that when students have used such packages, they often have a poor understanding of concepts like linking, debuggers, and so on. In addition, if a student switches from one Windows compiler package to another, s/he must learn a new environment. Therefore, this text will use/refer to the GNU C/C++ compiler; however the programs can be easily transported to the afore mentioned proprietary systems.

Having extolled the virtues of C/C++, we must mention here that some of the sample programs in this book reflect our roots in FORTRAN. There are many functions and subroutines available for scientific tasks which have been written in FORTRAN and have been extensively tested and used. It would be foolish to ignore these programs or attempt to rewrite them in C/C++. It is much less time consuming to call these libraries from C++ programs than it is to write your own version. In Appendix C we describe how FORTRAN libraries and subroutines can be called from C++.

Chapter 2

Basics

Before we start we need to introduce a few concepts of computers and the interaction between you, the user, and the machine. This will help you decide when to write a program for solving a physics or science problem and when it is much easier or faster to use a piece of paper and a pocket calculator. In thinking about computers, remember there is a distinction between hardware and software. Software is divided into the operating system and your particular application, like a spreadsheet, word-processor or a high level language. In this book we will spend most of the time in dealing with issues relevant to physics and the algorithms used to solve problems. However, in order to make this as productive as possible, we will start off with a short description of the hardware and then some discussion of the operating system.

2.1 Basic computer hardware

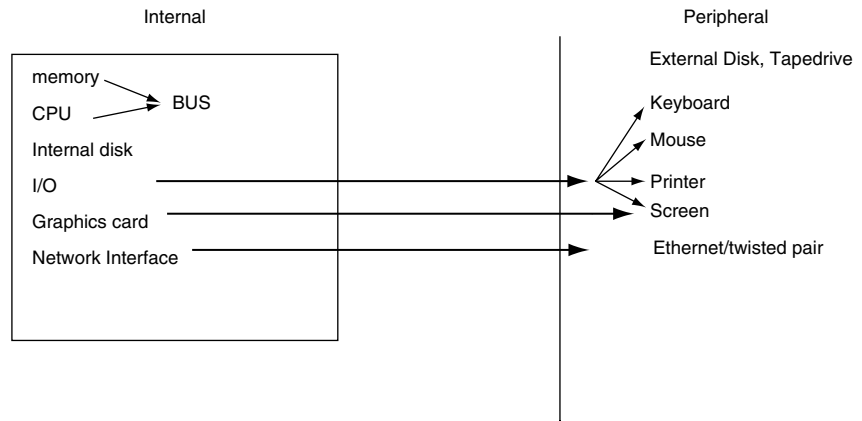
Apart from huge parallel supercomputers, all workstations you can buy today are organized in a similar way (Figure 2.1).

The heart of the computer is the **CPU** (Central Processing Unit) controlling everything in your workstation. Any disk I/O (Input/Output) or computational task is handled by the CPU. The speed at which this chip can execute an instruction is measured in Hz (cycles per second) at several GHz. The CPU needs places to store data and instructions. There are typically four levels of memory available: level I cache, level II cache, **RAM** (Random Access Memory) and **swap space**, the last being on the hard disk. The main distinction between the different memory types is the speed.

The cache memory is located on the CPU chip.

This CPU chip has two small memory areas (one for data and one for instructions), called the **level I caches** (see below for further discussion),

Figure 2.1 Schematic layout of a workstation.



which are accessed at the full processor speed. The second cache, level II, acts as a fast storage for code or variables needed by the code. However, if the program is too large to fit into the second cache, the CPU will put some of the code into the RAM. The communication between the CPU and the RAM is handled by the **Bus**, which runs at a lower speed than the CPU clock. It is immediately clear that this poses a first bottleneck compared to the speed the CPU would be able to handle. As we will discuss later, careful programming can help in speeding up the execution of a program by reducing the number of times the CPU has to read and write to RAM. If this additional memory is too small, a much more severe restriction will come into play, namely **virtual memory** or **swap space**. Virtual memory is an area on the disk where the CPU can temporarily store code which does not fit into the main memory, calling it in when it is needed. However, the communication speed between the CPU and the virtual memory is now given by the speed at which a disk can do I/O operations.

The internal disk in Figure 2.1 is used for storing the operating system and any application or code you want to keep. The disk size is measured in gigabytes (**GB**) and 18–20 GB disks are standard today for a workstation. Disk prices are getting cheaper all the time thus reducing the need for code which is optimized for size. However the danger also is that people clutter their hard disk and never clean it up.

Another important part of your system is the Input/Output (**I/O**) system, which handles all the physical interaction between you and the computer as well as the communication between the computer and the external peripherals (e.g., printers). The I/O system responds to your keyboard or mouse but will

also handle print requests and send them to the printer, or communicate with an external or internal tape or hard drive.

The last piece of hardware we want to describe briefly is the network interface card, which establishes communication between different computers over a network connection. Many home users connect to other computers through a modem, which runs over telephone lines. Recently cable companies have started to offer the use of their cable lines for network traffic, allowing the use of faster cable modems. The telephone modem is currently limited to a maximum speed of 56 kB/s, which will be even slower if the line has a lot of interferences. In our environment we are using an Ethernet network, which runs at 100 MB/s.

2.2 Software

Operating system

In getting your box to do something useful, you need a way to communicate with your CPU. The software responsible for doing this is the **operating system** or OS. The operating system serves as the interface between the CPU and the peripherals. It also lets you interact with the system. It used to be that different OSs were tied to different hardware, for example VMS was Digital Equipment's (now Hewlett-Packard) operating system for their VAX computers, Apple's OS was running on Motorola chips and Windows 3.1 or DOS was only running on Intel chips. This of course led to software designers concentrating on specific platforms, therefore seriously hampering the distribution of code to different machines. This has changed and today we have several OSs which can run on different platforms.

One of the first operating systems to address this problem was UNIX, developed at the AT&T Labs. Still a proprietary system, it at least enabled different computer manufacturers to get a variant of UNIX running on their machines, thus making people more independent in their choices of computers. Various blends of UNIX appeared as shown in the following table:

Ultrix	Digital Equipment Corporation (now HP)
HP True64 Unix (formerly OSF)	HP
Solaris/SunOS	SUN
AIX	IBM
HP-Unix	Hewlett-Packard

These systems are still proprietary and you cannot run HP-Unix on a Sun machine and vice versa, even though for you as a user they look very similar. In the 1980s Linus Torwald started a project for his own amusement called Linux, which was a completely free UNIX system covered under the GNU license. Since then many people have contributed to Linux and it is now a mature and stable system. In the scientific community Linux is the fastest growing operating system, due to its stability and low cost and its ability to run on almost all computer platforms. You can either download Linux over the internet from one of the sites listed in the appendix, or you can buy one of the variants from vendors like Red Hat or SuSE. The differences in these distributions are in ease of installation, graphical interfaces and support for different peripherals. However the **kernel**, the heart of the operating system, is the same for all.

Unlike other operating systems, when you get Linux, you also get the complete source code. Usually, you do not change anything in the code for Linux, unless either you are very knowledgeable (but read the license information first) or you want to get into trouble really fast.

Two other advantages of Linux are that there is a lot of free application software and it is a very stable system. The machines in our cluster run for months without crashing or needing to be rebooted.

Applications and languages

This is the part the average user is most familiar with. Most users buy applications out of the box, consisting of business software like spreadsheets, word processors and databases. For us the most important issue is the programming language. These are the languages you can use to instruct your computer to do certain tasks and are usually referred to as high level languages in contrast to assembly language.

We usually distinguish high level languages in the following way: interpreted languages like **Basic**, **Perl**, **awk** and compiled ones like **FORTRAN**, **FORTRAN90**, **C** and **C++**. The distinction, however, is not clear cut; there are C-interpreters and compiled versions of Perl. The interpreted languages execute every line the moment it is terminated by a carriage return and then wait for the next line. In a way this is similar to your pocket calculator, where you do one operation after the next. This is a very handy way of doing some calculations but it will pose some serious restrictions, especially when you try to solve more complex problems or you want to use libraries or

functions which have been previously written. Another disadvantage is the slow running of the program and the lack of optimization.

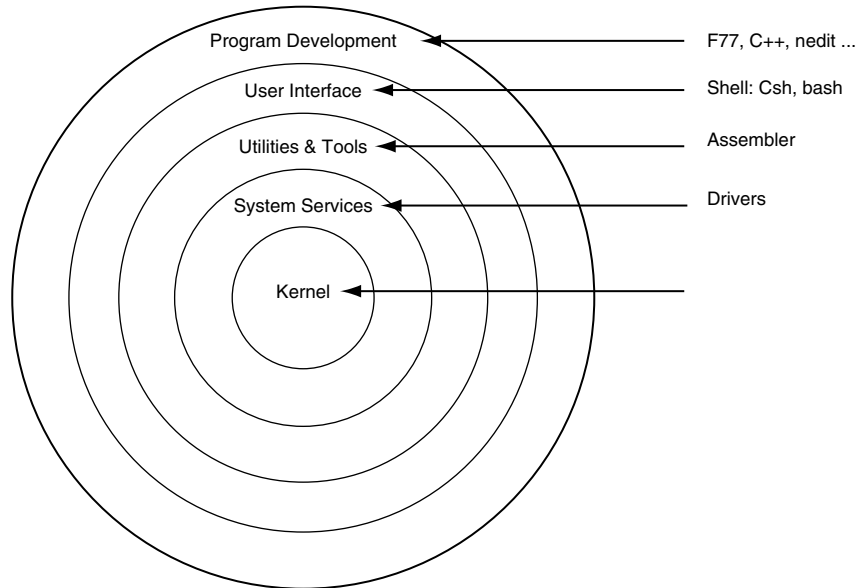
In a compiled language you first write your complete code (hopefully without error), gather all the necessary functions (which could be in libraries) and then have the computer translate your entire program into machine language. Today's compilers will not only translate your code but will also try to optimize the program for speed or memory usage. Another advantage of the compiler is that it can check for consistency throughout the program, and try to catch some errors you introduced. This is similar to your sophisticated word processor, which can catch spelling and even some grammar mistakes. However, as the spell checker cannot distinguish between two or too (both would be fine) or check whether what you have written makes sense, so the compiler will not be able to ensure consistency in your program. We will discuss these issues further below, when we give our guidelines for good programming practice.

After you have run your different routines through the compiler, the last step is the linker or loader. This step will tie together all the different parts, reserve space in memory for variables, and bind any needed library to your program. On Linux the last step is usually executed automatically when you invoke the compiler. The languages most used in scientific computing (especially in physics) are FORTRAN and C/C++. Traditionally FORTRAN was the language of choice, and still today there is a wealth of programs readily available in FORTRAN libraries (e.g. **CERN library**, **SLATEC**, **LAPACK**). During the last decade, C/C++ has become more and more important in physics, so that this book focuses on C++ (sigh!) and moves away from FORTRAN. We are still convinced that FORTRAN is the better language for physics, but in order to get the newer FORTRAN90/95 compiler, one has to buy a commercial package, while the C and C++ compilers are available at no cost for Linux.

2.3 How does it work?

In Figure 2.2 we have outlined how the different layers on a UNIX workstation can be grouped logically. The innermost part is the kernel, which controls the hardware, where the services are the part of the system which interacts directly with the kernel. Assembly language code will interact with this system level. The utilities layer contains programs like `rm` (remove) or `cp` (copy) and the compilers. The user interface and program development

Figure 2.2 Schematic layout of a workstation.



areas are where you will be working most. You can choose the particular shell you prefer, which then will be your interface to the lower levels. Even though you will be in an X-Window environment, you still have to use command line input and write scripts, which will automate your tasks. This is done in your chosen shell, and some of the commands will be different for different shells. In the outermost shell you will have your applications, like compiled programs.