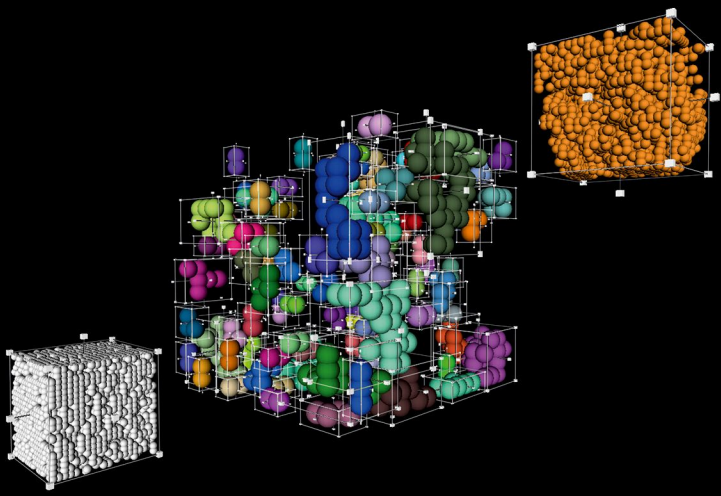
OXFORD



APPLIED COMPUTATIONAL PHYSICS

JOSEPH F. BOUDREAU | ERIC S. SWANSON

with contributions from Riccardo Maria Bianchi

APPLIED COMPUTATIONAL PHYSICS

Applied Computational Physics

Joseph F. Boudreau and Eric S. Swanson

with contributions from Riccardo Maria Bianchi

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Great Clarendon Street, Oxford, OX2 6DP,
United Kingdom

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide. Oxford is a registered trade mark of
Oxford University Press in the UK and in certain other countries

First Edition published in 2018
Impression: 1

Published in the United States of America by Oxford University Press
198 Madison Avenue, New York, NY 10016, United States of America

British Library Cataloguing in Publication Data
Data available

Library of Congress Control Number: 2017946193

ISBN 978–0–19–870863–6 (hbk.)
ISBN 978–0–19–870864–3 (pbk.)

DOI 10.1093/oso/9780198708636.001.0001

Printed and bound by
CPI Group (UK) Ltd, Croydon, CR0 4YY

For Pascale.
For Lou, Gordy, Suzy, Kris, Maura, Vin.
For Gordon V.

For Erin, Megan, Liam, and Drew.
For Max and Gudrun.

reor ut specular

Preface

New graduate students often experience something like shock when they are asked to solve real-world problems for the first time. These problems can be only rarely solved with pen and paper and the use of computational techniques becomes mandatory. The role of computation in any scientific endeavor is growing, and presents an increasing set of challenges. Numerical algorithms play a central role in theoretical prediction and in the analysis of experimental data. In addition, we see an increasing number of less numerical tasks taking on major importance in the life of young scientists. For example, how do you blend together two computing languages or split a computation between multiple computers? How does one design program libraries of numerical or scientific code for thousands of users? How is functionality added to a million-line reconstruction program? How can complicated datasets be visualized? What goes into a monitoring program that runs in a control room? These tasks are not particularly numerical or even scientific, but they are nonetheless important in the professional lives of scientists. From data acquisition systems to solving quantum field theory or presenting information, students face an intimidating computational environment with many languages and operating systems, multiple users with conflicting goals and methods, and complex code solving subtle and complicated problems.

Unfortunately, the typical student is marginally prepared for the challenges faced in modern computational ecosystems. Most students have had some exposure to a programming language such as C, C++, Java, or Fortran. In their first contact with "real" code, they may well be exposed to a proliferation of legacy software that in some cases is better used as a counterexample of good modern coding practices. Under these circumstances the usual solution is to learn on the fly. In a bygone era when the computing environment was simple this learning process was perfectly satisfactory, but today undirected learning leads to many false starts and some training has become indispensable. The search for help can be difficult because the nearby senior physicist probably grew up in an era preceding the explosive development of languages, paradigms, and computational hardware.

This book aims to fill some of the holes by introducing students to modern computational environments, object-oriented computing, and algorithmic techniques. We will rely on 'canned' code where reasonable. However, canned code is, by definition, incapable of solving research problems. It can at best solve portions of problems. At worst, it can lead the student researcher to false or incomplete conclusions. It is therefore imperative that the student understands what underlies his code. Thus an explanation of the numerical issues involved in common computational tasks will be presented.

Sometimes the numerical methods and applications will be quite technical; for this reason we regard this book as appropriate for newly graduated students. Our examples

will be drawn primarily from experimental and theoretical physics. Nevertheless, the book is also useful for students in chemistry, biology, atmospheric science, engineering, or any field in which complex analytical problems must be solved.

This text is meant for advanced (or graduate) students in the sciences and engineering. The physics ranges from advanced undergraduate topics in classical and quantum mechanics, to very advanced subject matter such as quantum field theory, renormalization, and scaling. The concepts of object oriented computing are introduced early in the text and steadily expanded as one progresses through the chapters. The methods of parallel computation are also introduced early and are applied in examples throughout. Since both the physics and the coding techniques can be replete with jargon, we attempt to be practical and provide many examples. We have not made any effort to prune away any discussion of fairly pedestrian material on the pretext that it is not advanced enough for a sophisticated audience. Our criterion is that the topics we discuss be *useful*, not that they be graduate-level, particularly since some topics are interdisciplinary in nature.

The numerical algorithms we consider are those applied in the major domain areas of physics. Classical problems involving a finite number of degrees of freedom are most often reduced to a coupled set of first-order differential equations. Those involving an infinite number of degrees of freedom require techniques used to solve partial differential equations. The study of quantum mechanical systems involves random processes, hence the temporal evolution of the system is handled though simulation of the underlying randomness. The computation of physical processes thus can be generally categorized according to the number of degrees of freedom and the stochastic or deterministic nature of the system. More complicated situations can mix these. For example, to follow a charged particle through a magnetic field in the presence of multiple scattering involves both deterministic and stochastic processes.

The flip side of simulation is data modeling. This is the procedure by which a mathematical description of data, often along with the values of physically interesting parameters, is obtained. Data modeling is an activity that consumes much of the time and creativity of experimental physicists working with datasets, large or small. While many treatises exist on the statistical analysis of data, the goal here is to explore, in somewhat greater detail than is usually found, the computational aspects of this field.

This text is neither a treatise on numerical analysis nor a guide to programming, but rather strives to develop practical skills in numerical and non-numerical methods applied to real world problems. Because of the emphasis on practical skills, students should expect to write programs and to refine and develop their programming techniques along the way. We assume a basic knowledge of C++ (the part of the language taken directly from C, minus the anachronisms), and treat the newer features in dedicated chapters. We do not, however, give a complete lesson on the syntax and symantics of any language, so we advise the reader who has not mastered C++ to learn it in parallel using any one of a number of sources. Our emphasis can then fall on using the language *effectively* for problems arising in physics.

A key ingredient to effective programming nowadays is mastery of object-oriented programming techniques—we strive to develop that mastery within the context of greatest interest to the target audience, namely physics. As noted in *Numerical Recipes*

(Press 2007), object-oriented programming has been "recognized as the almost unique successful paradigm for creating complex software". As a result, object-oriented techniques are widespread in the sciences and their use is growing. The physicist appreciates object oriented programming because his day-to-day life is filled with a rich menagerie of interesting objects, not just integers and real numbers, but also vectors, four-vectors, spinors, matrices, rotation group elements, Euclidean group elements, functions of one variable, functions of more than one variable, differential operators, etc. One usually gets more from a programming paradigm that allows user-defined datatypes to fill in gaps left at the language level. Encapsulation and polymorphism can be effectively used to build up a more functional set of mathematical primitives for a physicist—and also to build up an important set of not-so-mathematical objects such as are found in other nonscientific code.

Many books are devoted to object oriented analysis and design, and while some of these treatises are perfect for their target audience, a typical scientist or engineer most likely gets tired of examples featuring the payroll department and looks for a discussion of object oriented programming that "speaks his language". Accordingly, we include three chapters on object oriented programming in C++: Encapsulation, Polymorphism, and Templates. Other chapters of the book provide excellent examples of object-oriented techniques applied to various practical problems.

A companion web site has been established for this text at:

- `http://www.oup.co.uk/companion/acp`

The site includes example code (`EXAMPLES` area), skeletons which students can use as a starting point for certain exercises appearing in the text (`SKELETONS` area), and raw data for other exercises (`DATA` area). In addition the site provides user's guides, reference manuals, and source code for software libraries provided free of charge and used within this text. This software is licensed under the GNU Lesser General Public License. In referencing examples, skeletons, data, etc., we generally omit the full URL and refer simply to the directory, e.g. `EXAMPLES`, `SKELETONS`, `DATA`, etc.

Course organization

It is our experience that most of the material in this text can be covered in two terms of teaching. There are three main strands of emphasis: computational, numerical, and physical, and these are woven together, so the reader will find that emphasis alternates, though the book begins with more computational topics, then becomes more numerical, and finally more physical.

Computational topics include: Building programs (Chap 1), Encapsulation (Chap 2), Some useful classes (Chap 3), How to write a class (Chap 6), Parallel computing (Chap 9), Graphics for physicists (Chap 10), Polymorphism (Chap 12), and Templates, the standard library, and modern C++ (Chap 17).

Numerical topics include Interpolation and extrapolation (Chap 4), Numerical quadrature (Chap 5), Monte Carlo methods (Chap 6), Ordinary differential equations (Chap 11).

Topics related to applications in experimental and theoretical physics are Percolation and universality (Chap 8), Nonlinear dynamics and chaos (Chap 14), Rotations and Lorentz transformations (Chap 14), Simulation (Chap 15), Data modeling (Chap 16), Many body dynamics (Chap 18), Continuum dynamics (Chap 19), Classical spin systems (Chap 20), Quantum mechanics (Chap 21 and 23), Quantum spin systems (Chap 22), and Quantum field theory (Chap 24).

Finally, there are nearly 400 exercises of widely varying difficulty in the text. To assist students and instructors in selecting problems, we have labelled those exercises that are meant to be worked out without the aid of a computer as *theoretical* [T]; exercises which are more open-ended and require significant effort are elevated to the status of a *project* and labelled with a [P].

Acknowledgments

We are grateful to many people for encouragement and for lending their expertise. Among these are Jiří Čížek, Rob Coulson, Paul Geiger, Jeff Greensite, Ken Jordan, Colin Morningstar, James Mueller, Kevin Sapp, Søren Toxvaerd, and Andrew Zentner. J. Boudreau wishes to thank Petar Maksimovic and Mark Fishler for inspiration and help with the development of Function Objects (Chapter 3), Lynn Garren for support of the original package which appeared in the CLHEP class library, and Thomas Kittelmann and Vakho Tsulaia for their collaboration over the years. Preliminary versions of this text, and particularly the exercises, have been inflicted on our graduate students over the years–their help has been instrumental in effecting many important revisions. We thank them wholeheartedly.

Contents

1

Building programs in a Linux environment

The goal of this book is to develop computational skills and apply them to problems in physics and the physical sciences. This gives us a certain license to try to teach any (legal) computational skill that we believe will be useful to you sooner or later in your career. The skill set you'll need includes scientific computing and not-specifically-scientific computing. For example, applying statistical techniques in data analysis or solving the Schrödinger equation on the computer are distinctly scientific computing tasks, whereas learning how to work collaboratively with a code management system is a not-specifically-scientific task. But you will use both as you progress in your career and so we will aim to teach you a little of both. We will go back and forth to some extent between scientific computing topics and general computing topics, so that the more generic skill set becomes useful in writing programs of a scientific nature, and the scientific programs provide opportunities to apply and reinforce the full skill set of numerical and not-so-numerical techniques.

Applied Computational Physics. Joseph F. Boudreau and Eric S. Swanson, Oxford University Press (2018).

DOI:10.1093/oso/9780198708636.001.0001

Like mathematics, computing often appears to be a collection of tricks, with the well-known tricks elevated to the status of techniques. Deciding which tricks and techniques to teach is a difficult question, and a book like this has no traditional road-map. Our selection criterion is *usefulness*. Many of the topics are concerned with the simulation, classification and modeling of experimental data. Others (like integration or computational linear algebra) provide a basis for some of the later topics in simulation and modeling. Later, application to classical and quantum mechanical problems will be discussed. Like mathematics, computation is an art, and as practitioners we will pass on our own approach. If you learn to play a saxophone from John Coltrane, you will be absorbing John Coltrane's style, but also, hopefully, developing your own style along the way. So it is with the art of computation.

Writing executable programs and toolkits is of course central to this enterprise; since we are not about to describe computational techniques in wholly abstract terms, we have to be specific about which language(s) we are proposing to use. Our coding examples are usually expressed in the modern C++ language, or occasionally in the older, simpler computing language "C". We will sometimes also employ "pseudocode", which is a generic code-like description of any algorithmic process. Like almost any choice in computation, the focus of C++ is not totally obvious or universally acclaimed, but rather involves certain pros and cons–a debate that we will not lead you through here. The motivation for our choice is:

- In physics our lives consist of manipulating objects which are more abstract than scalar data types, including vectors, spinors, matrices, group elements, etc. While calculations involving these objects can be done in many computer languages, our lives will be vastly simpler if our computer languages support the objects of day-to-day life. No language is vast enough to support these at the language level, but languages supporting the object-oriented paradigm do allow you to add user-defined objects to the set of built-in data types. C++ also allows us to define basic operations on these data types, and maintains the speed of a compiled language.
- Most of a typical operating system is written in C and you will find it very easy to integrate specifically scientific software together with a vast body of generic software, particularly lower-level system calls.

Few people learn how to write software by writing programs from the bottom up. The "software stack" of even a very simple program can already involve toolkits that have taken a generation or two of computer scientists and physicists to develop. It is very common to make big scientific contributions by working on a small part of a huge program. Making modifications to an existing program, or filling in a piece of an incomplete program, can be a valuable learning experience. Some infrastructure for building programs is generally required. At a very minimum, a computing platform, operating system, and suite of compilers is needed. More complicated projects may even require a sophisticated set of tools to coordinate the development, distribution, and build of a software system. As more and more software is packaged and distributed for re-use,

the build of computer programs becomes more challenging. In this chapter we introduce the basic ideas related to the building of software with some very simple examples.

Our reference operating systems are the popular Ubuntu linux (now at version 17.04) and macOS (version 10.12.6). These are both variants of unix, an operating system written in C and dating back to the 1970s; and we will refer to them generically as such. The commands required for writing, building, and executing programs as well as tailoring the environment will be expressed as required by the bash shell on a Ubuntu linux machine. Because of its low cost and portability, the linux operating system is widely used in scientific computing. Not only can it be used on personal computers (laptops, desktops, and now even smart phones), but it can also be found running in the machine rooms of large computer centers on thousands of high-density rack mount computers. The latest version of Ubuntu linux can always be installed on a PC after downloading from the website `www.ubuntu.com`. The installation, management and customization of operating systems are not trivial skills, but they are also extremely useful.

Your first task is to get a laptop or a PC, and equip it with a basic linux operating system. We recommend that you install and maintain the operating system yourself. It is possible to dual-boot desktop and/or laptop computers, preserving the original operating system (e.g. Windows) which then coexists with linux. A Macintosh computer, which runs macOS, will also do for this book, since it runs an operating system similar to linux.

We assume a working knowledge of C++ basics–the part of C++ which is essentially just the C programming language, but minus anachronisms such as `malloc`, `free`, `printf`, `scanf`. In this text we develop in a few chapters that which is necessary to go beyond the ground level and understand classes, inheritance, polymorphism, and templates. For those who need to brush up on the basics, a good, short, but somewhat anachronistic introduction is the famous text of Kernighan and Ritchie (1988). The first few chapters of Capper (1994) or Bronson (2013) also cover the basics and provide a more modern introduction to the same subject. Another good source is the tutorial section of the online reference `www.cplusplus.com`. While our presentation of the C++ language will be far less formal than other common treatments of this powerful computing language, the physics applications will be more interesting and appropriate for the physical sciences, and you will "learn by doing", though it may be a good idea to refer to the above references occasionally if you prefer a more formal treatment of the language.

1.1 The editor, the compiler, and the make system

You write a program with an editor. There are a large number of these available on linux, and in principle any one will do. The *emacs* text editor (provided by the GNU

Table 1.1 *List of programs (center column) commonly used to compile major computer languages (left column).*

Language	Compiler under linux, OS X	Provided by
Fortran	gfortran	GNU Project
C	cc gcc	GNU Project
C++	c++ g++	GNU Project
Java	javac	Oracle

project; homepage `www.gnu.org/s/emacs/`) has features such as syntax highlighting, which can be very helpful in writing code, since it can recognize and draw to your attention syntax errors so that you can recognize them before they are reported by the compiler. The *gedit* text editor (provided by the GNOME project, homepage `projects.gnome.org/gedit/`) also has some of these features, and is perhaps more intuitive though less powerful. A much more basic editor called *vi* is generally pre-installed on even the most basic linux distributions. This is the editor of choice for gnarled veterans. On the other end of the spectrum, interactive development environments such as *Eclipse* (provided by the Eclipse foundation, homepage `www.eclipse.org`) embed powerful editors in a suite of tools for code development in C++ as well as other languages. There can be a steep learning curve with Interactive Development Environments (IDEs) such as Eclipse, but the effort can be worthwhile.

A single file of instructions is called a ***compilation unit***. A ***compiler*** turns these into ***object code***, and a ***linker*** puts different pieces of object code together into an executable program. Each computing language (Fortran, C, C++, Java) has its own compiler. Since C++ is a superset of C we can and will use the C++ compiler everywhere. Under linux, g++ and c++ are the same program. A table of common compilers is given in Table 1.1.

We look at a simple program which is a single compilation unit. It has a routine called main and like other functions takes arguments and returns an integer value (more on that, later). We call the file containing these lines `foo.cpp`. ".cpp" is the most common extension for c++ code. The program illustrates the important features of the main program unit, particularly how to write new commands under a unix operating system.

```
int main (int argc, char ** argv) {
  return 0;
}
```

Here are three ways to build an executable program from this source:

1. Compile to object code and then link to make an executable program.

```
$c++ -c foo.cpp -o foo.o
$c++ foo.o -o foo
```

2. Compile/link at the same time to make an executable program in one step.

```
$c++ foo.cpp -o foo
```

3. Use make

```
$make foo
```

The compilation step transforms human-readable C++ into machine instructions that the CPU can understand, and is called object code. The link step links together object code from various sources into an executable program. Which sources? The example above may give you the impression that there is only one, called `foo.cpp` but that is not true. Your program also contains pieces from the ***C standard library*** `libc.so` as well as others.

Even when the compilation and link is performed in one single command, there are still two phases to the process, and thus two points of failure. If you get an error message, try to figure out whether the error message is a compile error or a link error. Link errors do not occur at specific instructions, but constitute a failure to assemble the final program from the various pieces of object code, and usually in this case a piece of the program, or the object code containing the piece has been omitted, is missing, or cannot be located.

Once you've built the program you can see which run time libraries have been linked by issuing the command:

```
$ldd foo
```

which will generate the output

```
linux-vdso.so.1 =>  (0x00007fffc77fe000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f19f6313000)
/lib64/ld-linux-x86-64.so.2 (0x00007)
```

In addition to these libraries that are included automatically in the link of any C++ program, additional libraries can be linked by mentioning them explicitly in the list of arguments to the compiler as we will soon see, using the `-l` and `-L` flags. In general programs will include program libraries containing useful functions and class libraries containing useful classes (extensions to the basic data types of the language).

Our first example (`foo.cpp`) is extremely simple and not at all a typical project, which these days can consist of many thousands of compilation units. Managing the development and build of large software infrastructure becomes a complicated job. Usually the `make` system (provided by the GNU project, homepage `http://www.gnu.org/software/make/`) is used to build projects containing multiple compilation units. A pedagogical guide can be found in Mecklenburg (2004). The third way of building the program `foo` illustrates the basic principle of the system: `make` knows that a program (`foo`) can be built from its sources (`foo.cpp`) with `g++`, the C++ compiler.

It applies a set of rules to build the target from its prerequisites. One of those rules says that the `g++` command can be used to build an executable with name `foo` from a source code file named `foo.cpp`.

The make system can be extensively customized (and usually is) whenever `make` is used in the context of a large software project. The customization is achieved through ***Makefiles***, which are files usually named `makefile`, `Makefile`, or `GNUmakefile` that are placed in directories that contain the source code. In some cases these makefiles are written by hand and in others they may be generated automatically by another tool. We will describe this further as the need arises. Powerful as `make` is, it is often not sufficient on its own to organize the build of large or complicated projects, so additionally, a number of code management systems are available to coordinate and manage the distributed development of software *on top of make*.

1.1.1 Troubleshooting mysterious problems

On a few occasions you are likely to find that properly written code does not compile, link, or execute *because of the configuration of the platform and not the code itself*. Header files (with `.h` or extensions, discussed below) are normally installed in a system directory such as `/usr/include` or `/usr/local/include`; they can also be installed in other locations but then the qualifier

```
-I/path/to/include/area
```

must be added to the command line during the compile step. If the header files are not installed there then obviously the compile step will fail. Libraries, discussed in Section 1.8, are specified during the link step with the `-l` flag, and their search path is specified using the `-L` flag. These libraries must exist, they must be located during the link step, they must actually contain the symbols that they are supposed to provide, and they must be compatible. These symbols are all of the local and global variables known to the compiled code, as well as all the known structures, classes, free subroutines and member functions.

Computing hardware and operating systems exist in both 32 bit and 64 bit architectures, and object code which has been compiled for a 64 bit machine will generally not run on a 32 bit machine. Normally this type of object code would not be built or installed on the wrong architecture, but computers are machines and machines can go wrong. You might find a program or object library on a cross-mounted disk drive shared by machines having different architectures. Even the execution step can fail if bits of object code collected in shared libraries (files with the `.so` extension, discussed below) do not exist, cannot be located, or are incompatible. Incompatibilities can sometimes be caused by linking together object code produced by different compilers, or even the same compiler with different options. If these problems arise, the best approach is to be systematic in investigating and determining the cause.

To that end, it's useful to know about a few utilities in unix to help debug mysterious problems with "perfectly good" code. Table 1.2 summarizes a few of them to help you

Table 1.2 *Table of utilities for examining executable files. This is useful for investigating "mysterious problems" as described in the text.*

Linux command	OS X equivalent	Purpose
ldd	otool -L	check shared libraries required by a program. Prints the location of these libraries and flags any missing libraries.
file	file	classifies files. This command can yield useful information about how object files, and libraries, were compiled.
nm	nm	lists symbols in the object files, archives, shared libraries, and executables (variables and subroutines). Output can be very long!
c++filt	c++filt	Decodes "mangled" C++ symbols (for example, from nm or link error messages) and prints them in human-readable form.

see what you have just built; the unix manual pages for these commands give more information.

1.2 A quick tour of input and output

The first program you will generally write, like the famous Hello, World example in Kernighan and Ritchie (1988), simply echoes a few words to the terminal. In Fortran, input/output (IO) is handled at the language level; in C it is handled at the level of standard C functions, and in C++ it is handled through objects. Three important objects you must learn about are `std::cout` (the standard output), `std::cerr` (the standard error output), and `std::cin` (the standard input). In fact, since C++ is a superset of C, the C standard library routines (`printf` and `scanf` for the cognoscenti) can also be used but they are considered obsolete and should be avoided because they are unable to handle user-defined data types.

Basic usage of the `std::cin`, `std::cout`, and `std::cerr` objects is extremely easy. You will need to include the `iostream` header file at the top of any compilation unit that uses them:

```
#include <iostream>
```

Consider the following line:

```
std::cout << "Hello , World" << std::endl;
```

The "<<" characters in the above line constitute an operator, the left shift operator, which is defined in the C++ standard library for the `std::cout` object. The operator can stream bits of text, `int`s, `float`s, and `double`s, and even user-defined data types (if the designer allows it) to the standard output, i.e. the terminal. You should try this on your own. This is all we will say about `std::cout` for the moment. About `std::cerr`, we will say only that under unix operating systems (i.e. linux, Mac OS, and other variants), it is convenient sometimes to have two streams, both of which will normally end up printing to the terminal, because it is possible to redirect each stream separately[1] to a file or a unix pipe, for example. So, `std::cerr` functions just like `std::cout` except that normally program output is sent to `std::cout` while informational status, warning, and error messages are sent to `std::cerr`.

Your program can read input from the terminal using the `std::cin` class. This class can read in bits of text but also read the values of `int`, `float`, `double` (among others) from text strings from standard input–normally you think of typing these in using your actual fingers, but under unix you can also tell a program to take its "standard input" from a file, like this:

```
$ foo < file.txt
```

We use `std::cin` as follows in our programs:

```
#include <iostream>
...
int i, float f;
std::cin >> i >> f;
```

Input has one unique subtlety which we need to discuss: it can fail for several reasons. One reason is that the input may not be convertible to the right data type. For example, the word "particle" cannot be interpreted as an integer. Another reason is that the input may have reached its end, if the input were a file, or if the typing were terminated by C^D (Control+D). So we normally test that input has worked with the following incantation:

```
int i;
if (std::cin >>i) { // success!
  ...
}
else {              // failure!
  ...
}
```

[1] for more information, see the unix manual page on bash, particularly the section REDIRECTION.

1.3 Where to find information on the C++ standard library

The objects `std::cout`, `std::cerr`, and `std::cin` are all part of the C++ standard library, which contains a very large set of functions and objects, many of which will be extremely useful. The most usable documentation that we have encountered so far is an online reference, `www.cplusplus.com`, particularly the section called "library reference".

The C++ standard library is beyond the scope of our treatment and we will not attempt to explain more than the essentials, just what you'll need to follow this text and complete the exercises. We use it throughout the text, and discuss certain aspects in more detail in Chapters 6, 12, and 17. When you need detailed information, the ultimate authority on the C++ language and the C++ run time library is the ISO C++ standard. An updated standard was published in August 2011 by the International Standards Organization (ISO) and the International Electrotechnical Commission (IEC) as report number ISO/IEC 14882:2011. While this document is authoritative, it has little pedagogical value and is really intended for experts.

The best advice we can give to students confronted with such a large and complex set of standard software tools is to read the documentation as you go along, and try to learn, along the way, some of the tricks that will make you more efficient in your work. In this text we will explain bits and pieces of the C++ standard library as we go along, without any intent or commitment to treat the subject comprehensively.

1.4 Command line arguments and return values

An executable program is, generally speaking, a command like any other command in unix. Like any command, you can pass in ***command-line arguments***. These are available through the variable `char **argv`, an array of character strings, whose length is given by `int argc`. Let's modify our program `foo` now so that it simply echoes the command line arguments:

```
#include <iostream>                    // Include headers for
                                          basic i/o
int main (int argc, char ** argv) {    //
  for (int i=0;i<argc; i++) {          // Loop over command
                                          line args
   std::cout << argv[i] << " ";        // Print each argument
                                          to screen.
  }                                    //
  std::cout << std::endl;              // End-of-line.
  return 0;                            // Program successful
}
```

If you build this program and execute it with a few command-line arguments, it will behave as follows:

```
$./foo A B C D
./foo A B C D
```

Notice that the zeroth argument is the command name itself. The `$` symbol in the preceding example is the command prompt, echoed by the shell unless the user has configured his system otherwise.

If you are new to unix, you may not know that the operating system looks for commands in a standard set of directories. An ordered list of directories is held in the environment variable `PATH`. To see what directories are in your path, you can type

```
$echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:
/usr/games
```

For that matter, since you have now written a program very much like echo, you can use it to discover your path as well:

```
$./foo $PATH
./foo /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
/sbin:/bin
```

You can create your own directories where you can create important programs that you have written:

```
$mkdir ~/bin
$cp ./foo ~/bin
$export PATH=~/bin/:$PATH
```

This latter command can be added to the `.bashrc` file in your home directory, so that it is executed every time you log in. Then the program `foo` will appear as a built-in command, just like any other unix command, which, in fact, are built for the most part in exactly the same way. The source code for every command in Linux is free and publicly available–and most are written in the C programming language. For example, the `echo` command, which echoes the command line in a way very similar to `foo`, is compiled from source code which can be seen in `http://git.savannah.gnu.org/cgit/coreutils.git/tree/src/echo.c`; this web page points to a software repository which also contains other linux commands that may be familiar to you. You now have a partial glimpse of how a unix operating system, particularly many of the commands, are constructed.

Now back to our program, `foo`. The main routine of our program returns an integer value, zero in our example. This is the return status of the program. It's value can be

accessed after the program has executed by the "special" parameter "?". Its value is decoded by prepending a "$", as for any environment variable (try it! type `echo $?`). It is typical to signal successful completion of a command by returning 0, and one or more failure modes with a nonzero value.

1.5 Obtaining numerical constants from input strings

One way of inputting numerical constants to a program is to read them from standard input. This is convenient in some circumstances but not in others, since from the point of view of the user of the program (who is also often the developer) the most useful way to communicate input to the program may be via the command line. However, as we have seen, command line arguments are made available as an array of character strings. How do we extract numbers from these strings?

Part of the C++ standard library is a class called `std::istringstream`. To use this object, you initialize it with a character string and then extract the numerical data through the right shift operator, `>>`. As with the object `std::cin`, which is closely related to `std::istringstream` objects, you can test for success. Here is an example of how to parse an integer from the second item on the command line:

```
#include <sstream>
...
int main (int argc, char **argv) {
   ...
   int anInteger;
   std::istringstream stream(argv[1])
   if (stream >> anInteger) {  // Success!
   ...
   }
   else {                      // Failure!
   ...
   }
}
```

1.6 Resolving shared libraries at run time

Note that most programs are not complete without a set of shared libraries that are required by the program. Those shared libraries are to be found, normally, in standard locations. By default the area /usr/lib is searched for these libraries first, then /lib. By defining an environment variable called LD_LIBRARY_PATH, which is an ordered list of colon-separated directories, you can specify other libraries to search. If a shared

library cannot be found, the system will report an error when you attempt to execute a program:

```
$myprogram
myprogram: error while loading shared libraries:
libMyLibrary.so.1: cannot open shared object file: No such file
or directory
```

Missing libraries will also be reported by the `ldd` utility, described in section 1.1.

1.7 Compiling programs from multiple units

In textbooks and exercises, programs consist of a few lines of code typed into a single source code file. In the real world of physical sciences, computations this simple are rare. In the context of large scientific experiments, often the software infrastructure is a behemoth consisting of multiple millions of lines of code split across tens of thousands of individual files, representing a major financial investment on the part of national governments. This is the ecosystem in which students in the physical sciences, particularly physics and astrophysics, may find themselves trying to operate. It should be quite obvious that organizing and managing software infrastructure involves breaking it down into more manageably sized units. In this and the following sections we will see how to compile a program from multiple compilation units, and then how to archive the units and coordinate the build of the program.

While the necessity of splitting up source code arises from very complicated systems, we will illustrate the basic idea by splitting up a small one. Our program, called `iterate`, is designed to make a certain number of calls to the system routine `sleep`. It takes two command line parameters: the number of iterations, and the duration of each iteration:

```
$ iterate 6 4
0 sleeping for 4 seconds
1 sleeping for 4 seconds
2 sleeping for 4 seconds
3 sleeping for 4 seconds
4 sleeping for 4 seconds
5 sleeping for 4 seconds
$
```

Here is the program, all in one single file, which we call `iterate.cpp`

```
#include <iostream> // for std::cout, & cetera
#include <sstream>  // for istringstream
#include <cstdlib>  // for exit
//
```

```
// Define a data structure
//
struct Control {
  int iterations;
  int seconds;
};
//
// Parse the command line:
//
Control *initialize (int argc, char ** argv) {
  Control *control=NULL;
  if (argc!=3) {
   std::cerr << "Usage: " << argv[0] << " iterations seconds"
        << std::endl; exit(0);
  }
  else {
    control =new Control;
    control->iterations=0;
    control->seconds=0;
    {
      std::istringstream stream(argv[1]);
      if (!(stream >> control->iterations)) return control;
    }
    {
     std::istringstream stream(argv[2]);
     if (!(stream >> control->seconds)) return control;
    }
  }
  return control;
}
//
// finalize:
//
void finalize(Control *control) {
   delete control;
}
//
// execute:
//
void execute(Control *control) {
  if (control) {
    for (int i=0;i<control->iterations;i++) {
      sleep(control->seconds);
```

```
      std::cout << i << " sleeping for " << control->seconds
          << " seconds" << std::endl;
    }
  }
}
//
//
//
int main(int argc, char ** argv) {
  Control *control = initialize(argc, argv);
  execute(control);
  finalize(control);
  return 0;
}
```

The simplest way to build this is to type `make iterate`, which will in turn execute the following command in a child process:

```
$g++ iterate.cpp -o iterate
```

To illustrate the typical way of organizing a project with multiple compilation units, we will separate `iterate.cpp` into several pieces. The main program declares and defines three functions (in addition to `main`): `initialize`, `execute`, and `finalize`. It also declares a single data structure, called `Control`. The functions are defined where they are declared, which must be before they are used. Hence `main` is the last function to be defined, rather than the first.

In C++, as in C, every function and data structure (or class) must be declared before it is used, once, and only once; this amounts to specifying the interface to the function and/or class. The first step to breaking up this program is to separate the declarations from the definitions, putting them in a header file that can be included by each compilation unit. We therefore now create the header file `iterate.h`, and put it in the same directory as the source code:

```
#ifndef _ITERATE_H_
#define _ITERATE_H_

// Data structure controlling the iteration loop:
struct Control {
  int iterations;
  int seconds;
};

// Initialize.  Parse the command line:
Control *initialize(int argc, char ** argv);
```

```
// Execute the iteration loop:
void execute(Control *control);

// Finalize.  Clean up the memory:
void finalize(Control *control);

#endif
```

The preprocessor directives (those including `#ifndef`, `#define`, `#endif`) form an ***include guard***, guaranteeing that the declarations occur only once, even if the same header file is included twice, for example, directly, and indirectly.

We can now split our program into four compilation units, which we will call `iterate.cpp` (containing only `main`), `initialize.cpp`, `finalize.cpp`, and `execute.cpp`. Each one of these should include the header file `iterate.h` at the top, along with any other header files that it needs. For example the file `execute.cpp` looks now like this:

```
#include "iterate.h"
#include <iostream>
void execute(Control *control) {
  if (control) {
    for (int i=0;i<control->iterations;i++) {
      sleep(control->seconds);
      std::cout << i
                << " sleeping for "
                << control->seconds
                << " seconds"
                << std::endl;
    }
  }
}
```

The functions `initialize` and `finalize` have been moved into their own files, in a similar way; and the program `iterate.cpp` now becomes very simple:

```
#include "iterate.h"
int main(int argc, char ** argv) {
  Control *control = initialize(argc, argv);
  execute(control);
  finalize(control);
  return 0;
}
```

At this stage our directory contains four source code files–one of them containing the function `main`, which is required in any program. We can build the program `iterate`

in a number of different ways from these sources. The first option that we have is to compile and link all of the source code at the same time:

```
$g++ iterate.cpp initialize.cpp execute.cpp finalize.cpp \
   -o iterate
```

This is fine except it has the drawback that if you change one part of this "project" you need to recompile all of its files. This is not too high a price for the program `iterate`, but could be very high in a real-world project. The preferred option is to compile each unit to object code separately, and then link them together at the end. The `g++` command can be used for both stages of this procedure (here, the `-c` flag indicates that `g++` should compile to object code and not bother to attempt a link):

```
$g++ -c -o iterate.o iterate.cpp
$g++ -c -o initialize.o initialize.cpp
$g++ -c -o execute.o execute.cpp
$g++ -c -o finalize.o finalize.cpp
$g++ iterate.o initialize.o execute.o finalize.o -o iterate
```

A mixed approach can also be taken:

```
$g++ -c -o iterate.o iterate.cpp
$g++ -c -o initialize.o initialize.cpp
$g++ iterate.o initialize.o execute.cpp finalize.cpp -o iterate
```

Stepping back and looking at what we have done, we notice that the modularity of the program is greatly improved; the protocol for each routine is clear from looking at the header file (where additional documentation can also be collected), and developers can work individually on one piece of the program at a time. On the other hand, while the program development itself is simplified, the price is additional complexity on the side of *building* the program. In the real world the management of very many compilation units is an extremely complicated task often carried out by teams of full-time people. Fortunately, a number of standard tools have been developed to better manage the build of a program. In the following section we will discuss several of them that are almost always at the core of any project: libraries of object code, the make system, and a source code management system.

1.8 Libraries and library tools

Having seen how to break up the source code into more manageable units, we now address the issue of how to keep the compiled code together in order to ease the process of compilation. Files containing object code (with the `.o` extension) are often collected together into ***libraries***, so that the ***client*** code (`iterate.cpp` in the above

example) can link with one library rather than many different object files. In the world of professional programming the library (and associated header files and documentation) is the implementation of what is called the API, or ***application programming interface***, which represents a toolkit for the development of applications, vulgarly referred to as "apps" amongst the rabble.

There are two kinds of libraries, called static (files ending with `.a`) and shared (files ending with `.so` on linux systems or with `.dylib` on the mac). The main difference is that at link time, the object code in static libraries becomes part of the executable file; i.e., it is copied in. Programs linked to a shared library contain only pointers to bits of executable code which are not copied into the executable; and therefore the shared library file must be resolved at run time, before the program can be executed (see Section 1.6). Working with static libraries is usually simpler though it does create larger executables.

The linux command `ar` manipulates a static library. To create the static library `libIterate.a`, give the command:

```
$ar rc libIterate.a initialize.o execute.o finalize.o
```

The modifier `rc` stands for "replace" and "create": the library (or archive) is created if it doesn't exist, and if it does exist then any object files within the library are replaced by those given on the command line. The contents of the library can be examined with `ar t`:

```
$ar t libIterate.a
initialize.o
execute.o
finalize.o
```

Also, the contents of the library can be manipulated; for example individual pieces of object code may be extracted (`ar x`) or deleted (`ar d`). More information can be obtained from the `ar` manual page.

Now that you've built the library, you naturally will wonder how to link it to other code. The `g++` command has two important compiler flags that can be used, `-L` and `-l`. The first one, `-L`, specifies a directory to be searched for the library. The working directory is not on that path by default, so if your library lives in your working directory you should add `-L.` or `-L\`pwd\`` to the `g++` command. The second one, -l, gives the name of the library to link, but with an odd rule: to link to `libIterate.a`, you should write `-lIterate`. In other words, transform "lib" into `-l` and drop the extension `.a`. Both flags can be repeated on the command line and specify a search list: if the library is not found in the first directory, the second is searched; likewise if the symbols are not found in the first library on the command line, the second is searched. Therefore the order of `-L` and `-l` arguments is important. In our example, you will use the command:

```
$g++ iterate.o  -L. -lIterate -o iterate
```

To make a shared library, you can use the versatile g++ command again, giving the list of object files on the command line in addition to the `-shared` flag, and specifying a `lib` prefix and an `.so` extension for the output file, like this:

```
$g++ -shared initialize.o execute.o finalize.o -o libIterate.so
```

Some compilers also require the `-fPIC` qualifier during the compilation of object code files like `initialize.o` in order to produce "relocatable" code required in a shared library[2]. The only way to examine the contents of a shared library file is with the `nm` command, which dumps the symbol table. It is usually best to pipe this through `c++filt`, so that subroutine names are demangled into a readable form. You can link to the shared library in the same way as you link to a static library, and by default `libIterate.so` (for example) will be take precedence over `libIterate.a` if both files are found. Typically, when you then run the program, the shared object library will need to be resolved as well, as we have discussed in Section 1.6.

The software on a linux system is built from packages and these packages include programs, APIs, and sometimes both programs and APIs. One good (and useful) example is the ***gnu scientific library***, or `gsl`, which can be installed (on a Ubuntu linux system) by typing:

```
$sudo apt-get install libgsl0-dev
```

This installs the libraries `/usr/lib/libgsl.a`, `/usr/lib/libgsl.so`, as well as the headers `/usr/include/gsl/*.h`, and a manual page that can be referenced by typing

```
$man gsl
```

1.9 More on Makefile

The example we have been developing over the last few sections now constitutes a mini-project with a number of steps in the build process, namely (if we are going to use a static library):

- build the object files `initialize.o`, `execute.o`, `finalize.o`
- build the archive `libIterate.a`
- build the executable `iterate`

Extrapolating our experience to a large project, we can see that while building each constituent requires knowledge of a few compiler flags at the very least, building the entire

[2] This is particularly the case with the gnu compiler g++.

project interactively quickly becomes prohibitive. One way to automate this is to script the build (in `bash`, `csh`, or a similar scripting language); however with large projects one prefers to skip compiling code that has not changed since the last compilation. The `make` system was invented to build components of a large project from prerequisites. It can detect when a component is stale (because its prerequisites have changed) and rebuild it. It does this from rules, some of them built-in, while others can be specified to the system. The typical way to extend the set of built-in rules is to write a `Makefile` which lives in the same directory as the source code. We demonstrate this by an example. First, notice that the `make` system already knows that a `.cpp` file is prerequisite to an `.o` file with the same base name; typing make `initialize.o` is sufficient for the system to execute

```
g++    -c -o initialize.o initialize.cpp
```

when `initialize.cpp` is found in the working directory and has been modified more recently than `initialize.o`. Ditto for `execute.o` and `finalize.o`. `Makefile` does not know (yet) that these object files are prerequisite to the archive `libIterate.a`. Thus, we create a file named `Makefile` and add the following lines:

```
libIterate.a:initialize.o execute.o finalize.o
```

This says that `libIterate.a` is a target and that it depends on the three listed object files. Since it is the first target in the `Makefile` (and the only one for the moment), typing `make` will automatically build the three prerequisite files, from built-in rules. Following this line in the `Makefile`, one can specify which actions to take to build the target. Such lines must begin with a tab. The make system is very sensitive to this and using spaces will cause `make` to fail. We revise the `Makefile` so that it looks like this:

```
libIterate.a:initialize.o execute.o finalize.o
        ar rc libIterate.a initialize.o execute.o finalize.o
```

This can also be written a little differently, since the syntax of `Makefile` allows you to use the expression `$@`, which means "the target"; `$?` which means "the list of prerequisites"; and "$<", which means "the first prerequisite". Thus one can write:

```
libIterate.a:initialize.o execute.o finalize.o
        ar rc $@ $?
```

Macros can be defined in a `Makefile` too, so one can write this as:

```
OFILES=initialize.o execute.o finalize.o

libIterate.a:$(OFILES)
        ar rc $@ $?
```

Additional targets can be added to this list which now only includes `libIterate.a`. A common one to add is `clean` which removes all compiled code (.o files, static and/or shared libraries, the executable) and sometimes the backup files (ending in ~) left by the emacs editor. Such a line would look like

```
clean:
        rm -f *.a *.o iterate *~
```

(obviously you must be extremely careful!) and we can clean our directory now by typing `make clean`, which will rid the directory of everything but source code. Now we still have not told the make system how to build the final executable. We insert (at the top, after macro definition but before any other target) a line that determines how the executable `iterate` will be built; because this line will be the first target in the `Makefile`, `iterate` will be the *default target*. It will be built merely by typing `make` from that directory. The final version of `Makefile` looks like this:

```
OFILES=initialize.o execute.o finalize.o

iterate:iterate.o libIterate.a
        g++ -o $@ $< -L. -lIterate

libIterate.a:$(OFILES)
        ar rc $@ $?

clean:
        rm -f *.a *.o iterate
```

you can build the whole project by typing `make`, or pieces of it:

```
$make execute.o
$make libIterate.a
```

for example.

This probably looks like a nice tool, but even `Makefile` has its limits and many code management systems have been built on top of `Makefile` when things get even more complicated. We will not discuss these in this book, but be prepared for a shock when you enter the world of million-line software projects.

1.10 The subversion source code management system (SVN)

The last important tool that we will discuss in this introduction is the ***Subversion system***, or `SVN`, which is very widely used to manage the process of distributed software development. This is an example of a source code management system. Other examples

include the ***Concurrent Version System*** (`CVS`), which is now practically obsolete and ***git*** which is more recent and rapidly gaining popularity. In some ways these systems resemble a dropbox service, which is perhaps more familiar to a general audience. While a source code management system provides a central synchronized repository for code, it contains special features that enable tracking source code development and even restoring an entire project to a specific version and/or date. Some of the exercises in this book will require you to ***reposit*** the assignment in `SVN`; and your instructor will provide access to an `SVN` ***repository*** on a remote machine. Besides the ability to track changes and restore the code to some previous state, `SVN` allows for multiple developers to work on the same piece of code. Alternatively, the same individual can work on a project from multiple locations. This, for example, allows you to move potentially valuable pieces of work (including, but not limited to code, perhaps your thesis!) off of your laptop computer and onto a central site. Like many of the tools discussed in this chapter, `SVN` has a lot of powerful functionality and we will describe just enough of it so that you can get started using it. We assume that for your first experience with the system, somebody else (i.e. your instructor) will set up the `SVN` repository for you, set up the necessary access mechanisms, and grant you the necessary access rights.

1.10.1 The SVN repository

A repository is an abstract concept that implies a central place where code resides. How central? Repositories can be set up and administered by unprivileged users as files on a single laptop computer, if desired; more privileged administrators can configure servers where contributions from users around the country or around the world are centralized; and now commercial "cloud computing" services operate servers that can be accessed by users at little or no expense.

In the first case the `SVN` repository is a single file that the user creates:

```
$svnadmin create /path/to/repositoryfile
```

This creates a repository in a file called `/path/to/repositoryfile` that can now be filled, but *never* by acting directly on the repository file, which is henceforth only to be modified using `SVN` commands; i.e., those with the format `svn command options`. One useful command, `svn ls`, simply lists the contents of the repository, which, if you have just created it as explained above, is empty. The syntax of the command is

```
$svn ls file:///path/to/repositoryfile
```

If you wish to execute this from a remote machine (assuming that both local and remote machines can communicate via the secure shell, `ssh`), you can issue the following command from the remote machine:

```
$svn ls svn+ssh://user@host.domain/path/to/repositoryfile
```

where `user@host.domain` specified the username, hostname, and domain name of the local machine. Other types of servers, using other protocols, may have other prefixes such as `http://`, `https://`, or `svn://`.

For the following examples we assume that you have access to an `SVN` repository because either

- You have created a repository, as described above.
- You have requested and received a repository through a web-hosting service such as Sourceforge (`www.sourceforge.com`) or Cloudforge (`www.cloudforge.com`).
- Your instructor has set up a repository via one of the above methods, and given you access and instructions.

Web-hosting services have interesting benefits, first because they require no effort from the users to maintain and administer, second because they often add additional useful tools such as web-based code browsing. Whatever solution is adopted will will refer to the repository in the following as `protocol://repository`.

1.10.2 Importing a project into SVN

A project can be imported to `SVN` as follows. First we assume that the original source code lives in a directory called `/path/to/PROJECT`. To import that code into `SVN` issue the following command:

```
$svn import /path/to/PROJECT protocol://repository/PROJECT
```

This creates a project within the repository, and copies all of the files in /path/to/PROJECT into the repository. The project then grows only when additional directories are added to it, additional files are added to directories, or files are modified.

Each change to the repository is logged, and all the logged information is accessible via the command `svn log`. If you execute the `svn import` command as it appears above, an editor window will pop up and prompt you for a log message. Alternately, short log messages can be added by adding the option `-m "here is my log message"` to the command line.

1.10.3 The basic idea

We now have both a repository and a project residing in that repository. With this initialization out of the way, we describe the basic `SVN` operations for source code management.

The main idea behind `SVN` is that the repository holds a master copy of the project, while developers check out a local copy to their machine and make modifications. When they are satisfied, they check their modifications back into the master copy (repository). This is called a ***copy-modify-merge*** model. The simplest scenario is illustrated in Figure 1.1. The command

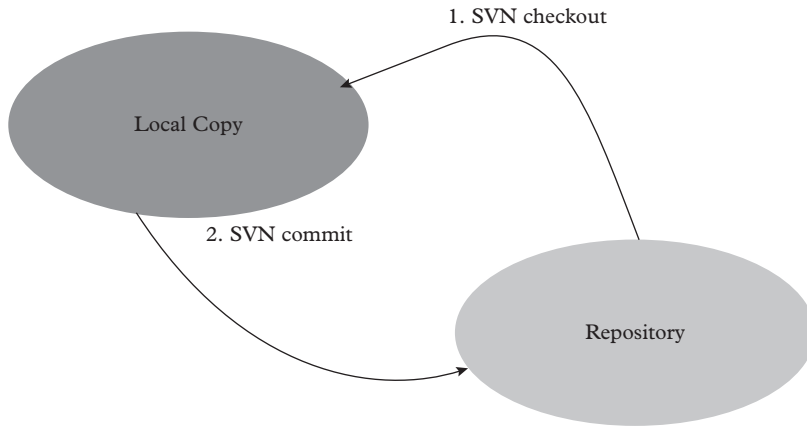


Figure 1.1 *Sketch of the simplest* `SVN` *transaction; first a checkout, then a commit following modification to the local copy.*

```
$svn checkout protocol://repository/PROJECT.
```

checks out the package (step 1 in Figure 1.1), and the command

```
$svn commit -m "message"
```

puts the modifications back into the repository (and adds the message "message" to the log). This is all that is needed as long as the file content of the project does not change. If files need to be added to the project then this is done with `svn add file1 [file2] [file3] ...`; the command schedules files for addition but does not add them until `svn commit` is issued; directories are added in the same way. `svn remove file1 [file2] [file3] ...` can be issued to remove files from the repository (after the files are removed from the local copy).

With multiple local copies in play (e.g., with multiple developers) the situation is more complicated because the master copy can change while one of the developers is making modifications to his or her local copy. Figure 1.2 shows a diagram of a typical transaction.

1. User 1 checks out the package to his/her local copy.
2. User 2 checks out the package to his/her local copy.
3. User 2 commits a modification to the repository. At this point user 1's local copy is out of date. *User 1 will not be able to commit until he refreshes his own copy with the changes that have gone into the repository.*
4. User 1 refreshes by typing `svn update`.
5. Then user 1 checks his changes into the repository with `svn commit`, after resolving any conflicts.

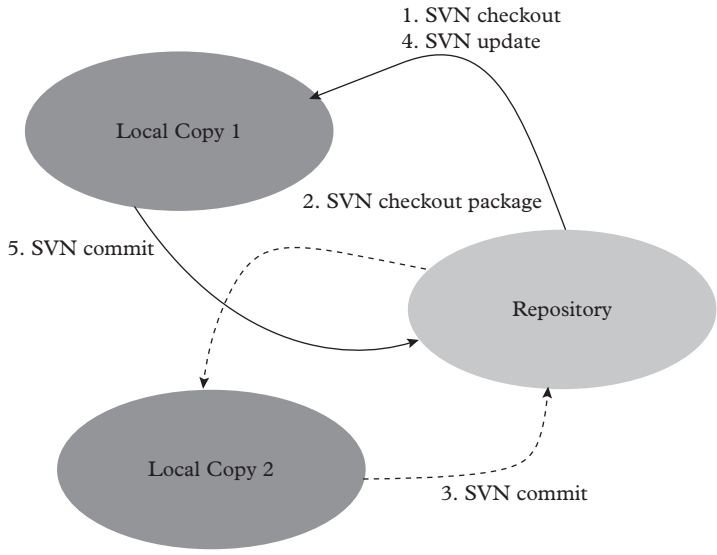


Figure 1.2 *A diagram of a more complicated set of* `SVN` *transactions arising from two users simultaneously developing within the same package. See text for details.*

Conflicts can arise when two users simultaneously work on the same package, but usually `SVN` is smart enough to resolve those without user intervention. `SVN` will resolve situations in which two users work on two different files in the same project. It will also resolve conflicts when two users have altered different parts of the same file. The only situation that normally requires user intervention is when two users have changed the same part of the same file; this can occur when two people jump in to fix the same bug. Conflicts are clearly indicated in the text and generally do not cause much trouble. We refer the reader to Collins-Sussman *et al.* (2004) for a detailed description of what to do when these cases arise.

The entire set of operations is carried out with very few `SVN` commands: `checkout`, `commit`, `update`, `add`, `remove`, `status`, and `log`; which suffices for most needs. The `svn checkout` and `update` commands have options that let users checkout, or update to, a tagged version, or a version that was current on a specific date and time. The `SVN log` command allows one to view the whole revision history of a file, including all of the comment messages that were given as arguments to the commit command. Finally, `svn status` displays the status of the project as a whole and/or individual file, depending on the parameters to the command.

It is important to never introduce executable programs, libraries, object code, backup copies, or large binary data files into the repository. These will make the administration of the repository difficult. Besides, these files need not be stored in the repository since they are typically regenerated from source. Putting `Makefiles` in the repository to aid in the build *is* generally a good idea.

After this brief introduction, we leave the reader to explore the full set of `SVN` operations using the references and/or the exercises.

1.11 Style guide: advice for beginners

We close with an offering of advice for beginners that has been gleaned over 80 combined years of programming experience. Presumably these good practices will be set by protocols if you are working in a large group. If you are not, read on.

dive in. Learning a computer language should be like learning French or German. You don't need to have mastered all the nuances of the plus-que-parfait tense to order a burger in Paris. Look at examples, learn some basics, and get coding!

adopt a convention. Adopting a convention for naming your files, libraries, variables, objects, and classes is important. Established conventions do exist, but we will not venture to advocate one over another since this is a matter of personal choice (or a choice that has been imposed by management). Peruse the Wikipedia article on ***Hungarian notation*** to get an idea of the options, and controversy, in adopting conventions.

use descriptive names. Calling an index `i` is traditional, but it is much better to give it a descriptive name such as `particleNumber`.

document your code. You will not remember the inner workings of your code after a few months. Describe its purpose, method, important variables, compiler directives, and usage somewhere (often at the top of the main file). If you are not using a repository, record revision history.

write clear code. You might think this is obvious, but it is quite easy for code to become unclear and one must be vigilant. Sooner or later you will find some clever compact way to do something and add it to your code. But you will not remember the trick the next time you look at your program and will not feel so clever. If you are going to be clever at least document it.

At a more general level, clear code is the result of clearly understanding the computational issue. Of course, as scientific programmers it is your duty to understand your problem to the best of your ability.

Lastly, if your code is destined to be used by others you should also consider the typical user's conceptualization of the code. In particular, you want the user interface to match the user's expectations. A familiar version of this problem is when a user attempts to change an aspect of text in a word processing program and all sorts of unexpected and frustrating code behavior ensues.

trap errors. A part of structured programming is trapping and dealing with errors when they are generated. This is implemented, for example, in C++ or java with the `try` and `catch` statements. Unfortunately, no such functionality exists in C or Fortran. Do what you can.

avoid magic numbers. Strings of digits should not appear in your code. You may recognize 0.197327 as $\hbar c$ in the appropriate units but the next person to work on your code may not. Better to place a statement such as `static float hbarc = 0.197327` somewhere. While we are on this, why would you ever write `double pi = 3.1415926` when you could write `double pi = 4.0*atan(1.0)`, or even better include the header `<cmath>`, which defines many mathematical constants, including `M_PI`?

allow for parameter input. Students tend to code parameter values directly into their programs. In our experience, programs are never run once and it is therefore important to allow for flexible parameter input. This is typically accomplished in one of three ways: (i) query the user for console input, (ii) via the command line, (iii) via an input file. Option (i) is simple; option (ii) permits easy scripting; and option (iii) is useful when many parameters are required but is more difficult to manage. One approach is to query the user and echo input data into a file. This file can then be modified and redirected as input in subsequent runs: `./foo < input.txt`. A more sophisticated approach might use a graphical user interface or a database.

specify default parameters. It is easy to spend days or weeks determining parameter values that optimize sought behavior. Once these are found they should be documented with the project, preferably as defaults in the code itself. We guarantee that you will not remember the preferred temporal grid spacing in your electromagnetic field computation a few months from now.

document the output. Most scientific code produces output files, sometimes dozens of them. These files should contain header information that specifies the program that created them, parameter values that were used, and the meaning (and units) of the output.

learn a debugger. Simple programs can be debugged with judicious print statements. However more complicated issues will require the use of a debugger, such as `gdb`. The GNU collaboration has created a free graphical front end for a variety of debuggers called *DataDisplayDebugger* that we happily recommend. Integrated Development Environments (such as Eclipse) often have debugging capability as well. It is worth the effort to learn to use these.

squash kludge. As you develop code you will often find yourself adding options to deal with an ever-increasing number of cases. The result is a kludgy mess with multiple layers of `if` or `case` statements. At this stage it is better to redesign the program or, more simply, to clone variant code.

develop systematically. Plan the general code structure before writing anything. Develop code in small logical chunks and debug as you go. Rushing ahead is a guarantee of future headache.

test extensively. If you are writing scientific code, you are doing science, and your code must be reliable. Check your results against known limits and special cases with analytical solutions. If you must strike out on your own, try to make the change from known territory as minimal as possible (for example, substituting a complicated potential for a square well potential in a quantum mechanics problem). Debugging Monte Carlo code is extremely difficult! You need analytic limits and simple cases worked out for comparison.

understand code parameters. Students often confuse *physical* parameters and *algorithmic* parameters. The former define the physics problem that you want to solve. The latter specify the method in which you solve the problem–in principle your answer should be independent of algorithmic parameters. It is up to you to confirm this! In practice this means extrapolating to infinity or zero. You can always double or halve an algorithmic parameter and confirm stability of your answer. Even better is to plot how your answer changes with algorithmic parameters, and better yet is to have a theoretical understanding of the functional form of that dependence so that reliable extrapolations can be made.

Similar advice with additional detail is contained in Wilson (2014).

1.12 Exercises

1. Write and compile an empty program. Name it `exp.cpp`, and put in in a directory called `CH1/EX1`. Compile it, and use the four utilities in Table 1.2 to examine the output file. Add to your empty program a call to the math library function `exp`, which requires you to include the header file:

   ```
   #include <cmath>
   ```

   Run the `nm` and the `ldd` utilities on the program and describe carefully what happens before and after the call to `exp` is added. Try this with both a constant and a variable argument to `exp`. Pipe the output of `nm` through `c++filt` and explain how the output is different. (This is called "name mangling" and "demangling").
2. Now write a variant of this program in a directory called `CH1/EX2`. Add to your program a single command line argument. Your program should now accept a single command line argument and echo the exponential of that argument, like this:

   ```
   $exp 5
   148.413
   ```

   Examine the program with `ldd` and `nm`. Pipe the output of `nm` through `c++filt`.

3. In a directory called `CH1/EX3`, replace the call to your own version of the `exp` function. Use a Taylor series expansion[3]. Check again with `ldd`, `nm` and `nm | c++filt`. Tabulate x vs. $\exp(x)$ for $x = \{-10, -8, -6....6, 8, 10\}$. By switching between your version of `exp`, and the one in the math library, determine how `nm` tells you whether the `exp` routine is internal or external to the executable program file.
4. In unix the wildcard character is the star or asterisk, "`*`". In a directory called `CH1/EX4`, write the program `foo` described in Section 1.4. Go into your home directory and execute `/path/to/foo *`. Explain the output.
5. Write a program called `plus` (in a directory called `CH1/EX5`) which expects two integers on the command line, adds the two integers and prints out the answer. The expected behavior of this program is:

```
$plus 3 7
3+7=10
```

6. In the previous example, break the code up into three compilation units:

   a) `main.cpp` – main program
   b) `parse.cpp` – parses the command line
   c) `add.cpp` – adds the two input values
   d) `print.cpp` – prints the result

   Put these four files together a directory called `CH1/EX6`, and add a `Makefile` which builds:

   a) a library called `libPlus.a`
   b) an executable called `plus`

   Make sure that the `Makefile` contains a "clean" target to remove all object code and executables generated by the compiler during the make procedure.
7. Clone the directory `CH1/EX6` into a directory `CH1/EX7`; then modify your `Makefile` so that the shared library `libPlus.so` is used, rather than a static library. Check and report the size of the final executable in both circumstances.
8. Write `Makefiles` in the directories (`CH1/EX1-CH1/EX7`) to build all of the targets in the previous exercises. Make sure each executable has a "clean" target.
9. Your instructor will provide access to a `SVN` repository. Add the source code and `Makefiles` for `CH1/EX1-CH1/EX7` to the repository.

   a) make sure that you do not reposit any object code.
   b) make sure that you can check out and build each executable from scratch.

[3] In Chapter 4 we will learn a better way to implement an exponential function

BIBLIOGRAPHY

Bronson, G. J. (2013). *C++ for scientists and engineers*. Cengage Learning.

Capper, D. M. (1994). *The C++ programming language for scientists, engineers, and mathematicians*. Springer-Verlag.

Collins-Sussman, B., Brian W. Fitzpatrick, and C. Michael Pilato (2004). *Version control with subversion*. O'Reilly Media; also available freely as http://svnbook.red-bean.com

Kernighan, B. W. and D. M. Ritchie (1988). *The C Programming language*. 2nd ed. Prentice-Hall.

Mecklenburg, R. (2004). *Managing projects with GNU make*. O'Reilly Media; also available as www.oreilly.com/openbook/make3/book

Wilson, G. *et al.* (2014). *Best Practices for Scientific Computing*. PLOS Biol **12(1)**: e1001745.

2

Encapsulation and the C++ class

2.1 Introduction

In the previous chapter we concentrated on how computer codes are written, compiled, archived, linked, and executed; how executable systems interact with the linux operating system; how source code is managed; and how build procedures can be automated. In this chapter we will talk more about the code itself, assuming that the reader has familiarity with the basics of C++, namely things like operations, control structures, functions, arrays, and pointers.

While datatypes are likely also familiar to the reader, we discuss integer, floating point, and complex datatypes in some detail for two reasons. The first is that in numerical calculations, one needs to understand the limitations in accuracy arising from the representation of numbers on a computer and the way in which they are acted upon by the CPU during an arithmetic calculation.

The second reason is more subtle. A new feature of C++ (relative to its predecessors C and Fortran), and probably the most important building-block of modern code, is the user-defined datatype or ***class***. Some of these datatypes, like those you are already using for input and output, are part of the C++ standard library, while others may be imported from other sources and ultimately written by you.

Applied Computational Physics. Joseph F. Boudreau and Eric S. Swanson, Oxford University Press (2018).

DOI:10.1093/oso/9780198708636.001.0001

If the notion of a class is unfamiliar, we will not give a full or accurate definition here, but for our numerically literate target audience it is productive to *initially* think of a class as a new type of numerical datatype, like a floating point number or an integer. For example, in C++ complex numbers are not built into the language; they are represented by their own class in the C++ standard library. Vectors, four-vectors, matrices, quaternions, octonions, and other useful entities can be imported from a variety of sources.

When a skilled programmer implements a class in C++, his or her implementation shares some of the same features that make the built-in datatypes convenient to use. Most notably, there is a clean separation between the representation of the datatype and its interface; i.e., the set of operations to which it responds. This separation is called ***encapsulation***. We expect to find this feature in classes as well. Along with inheritance and polymorphism, to be tackled later, encapsulation is one of the key components in the object-oriented style of programming.

2.2 The representation of numbers

A number, floating point or otherwise, is represented by a array of bits held on a magnetic storage device or a silicon flip-flop circuit. Encoding this information is fairly simple in the case of integers but can be quite complicated in the case of floating point numbers. Encapsulation is achieved by hiding all of this internal structure in several datatypes:

- `int`
- `unsigned int`
- `float`
- `double`,

which respond to four basic operations +,-,/,*, as well as a few others. The encapsulation of the internal representation of these numbers is so effective that many programmers seldom consider it. For numerical work it is, however, important to know about the internal structure because the accuracy of numerical computations depends on it. We therefore describe the manner in which numbers are represented and the attendant consequences for computation.

Numbers are stored in arrays of ***bits***, which are comprised of 0's and 1's. A collection of eight bits is called a ***byte***. All built-in datatypes require at least one byte of storage. The number of bytes used to store a datatype is called the size of the datatype, and it varies from platform to platform. The size can can be determined with the `sizeof` operator in C(++):

```
int sizeofA=sizeof(a);
```

A short program to print out the size of the major built-in datatypes is shown below:

```
#include <iostream>
int main(int argc, char **argv) {
  std::cout << sizeof(bool) << std::endl;
  std::cout << sizeof(char) << std::endl;
  std::cout << sizeof(int) << std::endl;
  std::cout << sizeof(unsigned int) << std::endl;
  std::cout << sizeof(long int) << std::endl;
  std::cout << sizeof(long unsigned int) << std::endl;
  std::cout << sizeof(short int) << std::endl;
  std::cout << sizeof(short unsigned int) << std::endl;
  std::cout << sizeof(float) << std::endl;
  std::cout << sizeof(double) << std::endl;
}
```

You should copy this into an editor and compile and run it on your machine.

2.2.1 Integer datatypes

An `unsigned int` has 4 bytes or 32 bits. It uses these to express a number in base-2 according to the expression:

$$x = a_{31}2^{31} + a_{30}2^{30} + \ldots + a_0 2^0$$

where a_i is either 1 or 0. This is called ***fixed-point notation***. Other unsigned integer datatypes are `char`, with a size of one byte, and `long unsigned int`, which is 32 bits on some machines and 64 bits on more modern ones. In an `int` variable the upper bit is reserved for the sign:

$$x = (-1)^{a_{31}}(a_{30}2^{30} + \ldots + a_0 2^0)$$

A consequence is that there is a maximum and minimum integer that can be represented for every datatype; between those limits integers are stored exactly. Integer operations + (addition), – (subtraction) and * (multiplication) are also exact as long as one does not overflow or underflow the range, but the operation / (division) discards any remainder to obtain an integer result. The range of integer datatypes can be obtained using the `std::numeric_limits` class:

```
#include <iostream>
#include <iomanip>
#include <limits>
int main( int argc, char ** argv) {
```

```
  std::cout << std::numeric_limits<bool>::min() << std::endl;
  std::cout <<  std::numeric_limits<bool>::max() << std
     ::endl;
  std::cout << (int) std::numeric_limits<unsigned char>
     ::min() << std::endl;
  std::cout << (int) std::numeric_limits<unsigned char>
     ::max() << std::endl;
  std::cout << std::numeric_limits<int>::min() << std::endl;
  std::cout << std::numeric_limits<int>::max() << std::endl;
  std::cout << std::numeric_limits<unsigned int>::min()
     << std::endl;
  std::cout << std::numeric_limits<unsigned int>::max()
     << std::endl;
  std::cout << std::numeric_limits<long int>::min() << std::
     endl;
  std::cout << std::numeric_limits<long int>::max() << std::
     endl;
  std::cout << std::numeric_limits<long unsigned int>::min()
     << std::endl;
  std::cout << std::numeric_limits<long unsigned int>::max()
     << std::endl;
}
```

Good documentation on this class can be obtained from the online reference `www.cplusplus.com`. Running this code reveals that a 32 bit `int` has an allowed range of −2147483648 to 2147483647. For an `unsigned int` the range is [0,4294967295]. A `char` has a range of [0,255]. These ranges are not big; for larger numbers one needs to resort to floating-point datatypes.

2.2.2 Floating point datatypes

A fixed-length datatype can store a finite number of values, while (even within a finite range of values) the number of rational numbers is infinite, and the number of real numbers is even larger. Thus only certain floating point numbers can be represented exactly, and the result of floating point operations involves some ***roundoff error***.

The typical error in a floating point calculation is called the ***machine precision***; for `float` variables this is between 6 and 7 decimal places, and for `double` variables it is between 15 and 16 decimal places. The class `numeric_limits` reveals that the range of `floats` is between $1.17549 \cdot 10^{-38}$ and $3.40282 \cdot 10^{38}$, while that of `double` variables is from $2.22507 \cdot 10^{-308}$ to $1.79769 \cdot 10^{+308}$.

A modern CPU does not require more time to carry out floating point operations on `double` variables than on `floats`, so `double` will be our preferred floating point datatype (unless memory or disk space forces us to use a more compact representation).

Table 2.1 *Bit allocation for float and double datatypes.*

Type	Sign	Exponent	Fraction
`double`	1 Bit	11 Bits	52 Bits
`float`	1 Bit	8 Bits	23 Bits

The IEEE-754 standard format for a floating point number is the following:

$$x = (-1)^s 1.f \times 2^{e-b}$$

where:

- s is the sign bit.
- 1 is always implied. It is called the "ghost bit"
- $1.f$ is the mantissa, with f being the fraction. For `doubles`, it is stored in 52 bits (23 for `float`)
- e is the exponent. For `doubles`, it is stored in 11 bits (8 for `float`)
- b is the bias. For `doubles`, it has a fixed value 1023 (127 for `float`).

Note that zero can have the sign bit set or unset, so the numbers $+0$ and -0 are distinct, though they give the same results in all floating point operations. Table 2.1 shows bit allocation for `float` and `double` datatypes.

The exponent of a floating point number is stored exactly, so the precision of the number is the precision of the full mantissa.

In scientific notation the numbers 3.0×10^4 and 30×10^3 are the same. Since a fixed number of bits are allocated for the mantissa it is advantageous to have the mantissa as left-shifted as possible. In this form the binary representation is $1.x \times 2^n$. Since the binary digit to the left of the decimal point is always 1 it is not stored. The resulting number is referred to as ***normalized***.

2.2.3 Special floating point numbers

The result of a numerical operation can be undefined if, for example, a division by zero is made. Even if the result of a calculation is defined mathematically, it can still result in an underflow or overflow due to the limited range of floating point variables. For `doubles` the exponent is stored in eleven bits. The normalized numbers discussed above have an exponent between 0 and 2046 (in binary 11111111110). Underflows, overflows, and other the results of undefined operations are represented by an exponent field with all eleven bits set (i.e., decimal 2047).

Infinity of either sign is represented by a fraction of zero, with the sign determined by the sign bit. When the fraction is nonzero the number represents an undefined quantity,

called ***NaN*** ("not a number"). This could result, for example, from dividing by zero or taking the square root of a negative number. The value of the fraction (called the ***payload***) contains extra information that can be decoded by clever programmers.

If the first bit of a fraction is nonzero the invalid number is called a ***signalling NaN***. When such variables arise, many floating point routines will raise a signal and exit with an error. If the first bit is not set, the invalid number is called a ***quiet NaN***. Computations are expected to fail silently when they encounter such a number. These numbers can be obtained from the `numeric_limits` class:

```
double qNaN=double nan=std::numeric_limits<double>
   ::quiet_NaN();
double sNaN=double nan=std::numeric_limits<double>
   ::signaling_NaN();
double inf =double nan=std::numeric_limits<double>
   ::infinity();
```

The smallest floating point numbers are represented with an exponent $e = 0$. In this case a smaller bias is taken (1022 for `double`, instead of 1023; for `float` 126 instead of 127) and the ghost bit is taken but ***subnormal numbers***. Subnormal numbers lie between true underflow and the result of `std::numeric_limits<double>::min()` which in fact returns the smallest *normalized* number. They lack the precision of normal numbers. A routine called `std::fpclassify` is available as part of the runtime library:

```
#include <cmath>
int std::fpclassify(double x);
```

which classifies floating point numbers as zero, `NaN`, normal, or subnormal. Like all library and system calls, the routine is documented in the linux manual pages and you can read the documentation by typing

```
man fpclassify
```

A related routine called

```
  int std::isfinite(double x);
```

returns 1 if the number is normal or subnormal and 0 if it is infinite or `NaN`. This is useful when debugging code. For example, the following code illustrates how `isfinite` can be used to track down a divide-by-zero error.

```
#include <sstream>
#include <stdexcept>
...
```

```
double z = y/x;
if (!std::isfinite(z)) {
  std::ostringstream stream;
  stream << "x,y,z=:" << x << "," << y << "," << z << std::
     endl;
  throw std::runtime_error(stream.str());
}
```

This example ***throws an exception***, which is a good way for C++ to signal that something has gone wrong; calling routines can ***catch*** the exception and take appropriate actions. In an interactive debugger, one can not only print out information about the condition that led to an infinite or undefined value of z, but also set a breakpoint at the print statement and examine other variables at that point in the program's execution in order to determine why the abnormal condition has occurred.

Routines called `std::isnormal`, `std::isnan`, and `std::isinf` also exist and are defined in the `<cmath>` header file. Their functionality is pretty obvious; details can be found in the manual pages.

2.2.4 Floating point arithmetic on the computer

When two floating point numbers are added in the CPU, the mantissa of the smaller of the two numbers is right-shifted (divided by two), an operation that is compensated by incrementing the exponent by one unit, until the exponents of the two numbers are equal. As a result the lower-order bits are shifted away and precision is lost. Once this is done, the CPU takes the sum of the two mantissas and uses the common exponent to form the result, which it then normalizes. The summation is carried out in an ***arithmetic logic unit*** that mostly consists of simple gates capable of carrying out Boolean operations on the bits of the numbers.

A simple example illustrates the loss of precision in base−10 arithmetic.

- We are going to add 1.276×10^6 to 2.852×10^8.
- The smaller number is expressed as 0.012×10^8.
- The sum is 2.864×10^8. Notice that the last two digits in the first operand have been lost.

The same type of rounding error takes place in base-2 arithmetic with floating point operations in the CPU and one has to worry about the cumulative effect of these rounding errors. Notice that the entire mantissa is shifted away when the number of digits in the mantissa is less than the difference in the exponents of two summands. In this case the sum is equal to the larger summand.The largest number for which this occurs is the machine precision mentioned before. The numerical value of the machine precision depends on the data type and can be accessed via the `numeric_limits` class:

```
std::numeric_limits<float>::epsilon()
```

Example: In the Large Hadron Collider (LHC), experiments energy deposits of hundreds of GeV can be placed in the heavy materials of a device called a calorimeter. The deposit consists of very many smaller ionizations called hits, where the hits can consist of deposits down to about one KeV. The full deposit and the hits that contribute to this deposit therefore range over about eight orders of magnitude. When summing all of the smaller hits into one deposit using `floats`, one eventually reaches a point where including additional low energy hits produces no difference, and this energy is lost to the calculation. The simplest resolution is to employ higher precision data types such as `double`. An algorithmic alternative is to sum small numbers in a hierarchical fashion.

A similar loss of precision occurs with subtraction. Subtraction is performed in very much the same way as addition: the two numbers are expressed with a common exponent (shifting the mantissa if needed) and then the mantissas are subtracted using logic gates in the CPU. When two numbers of similar size are subtracted, the result will be expressed in only a few bits. When the result is normalized, those bits will be shifted left as much as possible and the trailing bits will be filled with zeros. The resulting roundoff error can be significant.

Sometimes the problem can be avoided with a little cleverness. For example, consider the stretch factor $\gamma = 1/\sqrt{1-\beta^2}$ for a relativistic particle, where $\beta = v/c$, v is the particle's speed, and c is the speed of light. Since relativistic particles have $\beta \approx 1$, there is a good chance that a loss of precision will make the denominator of this expression zero. One can instead express $1-\beta^2 = (1+\beta)(1-\beta)$. Setting $\epsilon = 1-\beta$ gives $\gamma = 1/\sqrt{(2-\epsilon)\epsilon}$, which will generally be calculated to higher precision.

Machines carry out approximate computations on approximate representations of real numbers. Nevertheless, one must admire the skill with which the developers of the IEEE-754 standard, and our compiler set, have kept these details so effectively hidden from our sight, thereby freeing us to frame the problems of our day-to day life in terms of numbers, rather than bits. We will attempt to achieve the same functionality, via encapsulation, when defining our own datatypes.

2.3 Encapsulation: an analogy

In discussing the notion of encapsulation, we find it germane to draw an analogy to a development in electrical engineering that parallels developments in software engineering. We are referring to the advent of integrated circuitry, which has promoted the proliferation of highly complex circuitry such as sophisticated Central Processing Units (CPUs) with up to 10 billion transistors on a single chip.

Figure 2.1 shows a popular operational amplifier called the LM741. On the right is a circuit diagram, which gives one view of what is inside the neat little package.

Users of the LM741 need to be knowledgeable about the eight pins in the package, which, counterclockwise from the upper left, are the offset null used to cancel voltages that appear as the result of leakage currents, the negative and positive input, the negative supply voltage, an unused pin, the output, the positive supply voltage, and a second offset null. While sometimes a serious designer needs to know about the internal circuitry of this

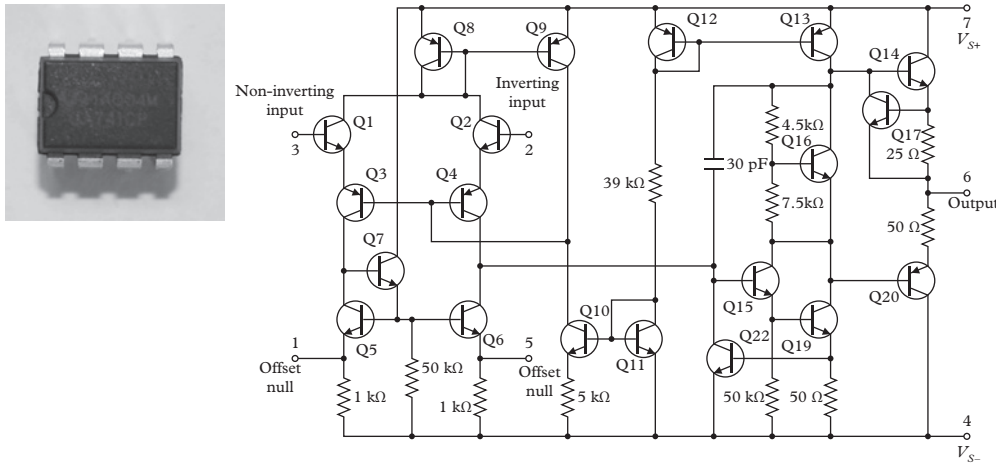


Figure 2.1 *Picture (left) and circuit diagram (right) of the LM741 operational amplifier (adapted from svg file by Daniel Braun from https://en.wikipedia.org/wiki/Operational_amplifier. Creative commons BY SA 3.0).*

device, most of the time he or she can forget those details when building more complex devices, or at least relegate their consideration to the later stages of design.

The LM741 implements a clean separation between what is exposed to the outside world and the messy internal hardware. The choice to hide the details of the circuit is a matter of convenience and practicality. The choice of which parts of the circuit to expose on the chip's interface is a matter for the chip designer, who must develop a clear idea of what is to be achieved with the chip. In this case, the LM741 is an amplifier. Should it contain headphones then? No, because it is not an iPod.

In carrying out computational work we use (and sometimes write) classes for mathematics and physics that help us with these computations. Such classes can represent vectors or matrices, for example. And these classes should, like the integrated circuit, have a trustworthy mechanism on the inside which is not exposed, and a clean, simple, and logical interface on the outside. This is what we call encapsulation. While the LM741 is encapsulated in a plastic case, encapsulation in object-oriented programming is conceptual and is applied to user-defined datatypes, namely the aforementioned classes.

2.4 Complex numbers

Our next example of a numerical datatype is `std::complex`. Unlike `float`, `double`, and the integer datatypes, `std::complex` is a C++ class; actually it is a ***class template***; roughly speaking this is essentially a recipe for generating a class for complex numbers once the internal representation of real and imaginary numbers (`float` or `double`)

is fixed. The choice is in your hands. The following incantation will create a complex variable:

```
#include <complex>
std::complex<double> I(0,1);
```

This creates a variable `I` and assigns it the value $i = \sqrt{-1}$, where i is evaluated to double precision. The statement is intended to closely parallel the declaration of an integer variable `i`:

```
int i=1;
```

Here, `std` is a ***namespace***, which is a mechanism to avoid clashes between the names we give to constants, variables, and datatypes. If you don't want to type the full namespace-qualified form of the declaration, you can also add the line `using namespace std;` prior to the declaration, usually at the top of a compilation unit or near the beginning of a particular function.

We introduce some vocabulary which applies to classes. The *declaration* (found in `<complex>`) of a class does not create any new variables, known as *objects*, but only specifies how those variables are created and what they can do, i.e. their *interface*. New objects are created typically with a statement that looks like the declaration of any other variable. For a class representing a vector the statement might look like

```
Vector m(0,0,0)
```

—think:

```
int i(0);
```

The creation of an object actually results in a function call, to the so-called *constructor*. `Vector` is the *class*, and `m` is the *object*. A class is a category of objects which all behave according to a common interface. Each object is said to be an ***instance*** of the class, and the creation of objects is also called ***instantiation***.

Classes typically contain ***bound functions***, which are invoked like this:

```
object.function(arguments);
```

rather than the free subroutine calls that you are used to, i.e.,

```
function(arguments);
```

When we invoke a function bound to an object we are said to be ***sending a message*** to the object.

The class template `std::complex` is a built-in part of the C++ standard library; we say that it is parameterized on the data type used to represent the real and imaginary parts of the complex number. To instantiate:

```
std::complex<double> U(1,0);
std::complex<double> I(0,1);
```

where the datatype `double` is called the template parameter, the values used in the constructor call are the real and imaginary parts of the complex number. The first variable declared above (`U`) is the real number 1, while the second variable `I` is the imaginary number $i = \sqrt{-1}$. One way to simplify the declaration of variables such as these is to use a ***type definition*** to establish a ***typedef-name***:

```
typedef std::complex<double> Complex;
```

then, one can use the more convenient typedef-name (`Complex`) instead of the parameterized class (`std::complex<double>`). A number of functions are declared to carry out elementary operations on complex numbers; these are defined in the Table 2.2. The operations in this table can be used to add, subtract, multiply and divide any combination

Table 2.2 *Summary of operations defined on complex numbers.*

Operation	Function	Example
+	operator +()	`w=x+y`
-	operator -()	`w=x-y`
*	operator *()	`w=x*y`
/	operator /()	`w=x/y`
z^* (complex conjugation)	conj()	`w=conj(z)`
$Re(z)$	real()	`w=z.real()`
$Im(z)$	imag()	`w=z.imag()`
$\|z\|$	abs()	`double x=abs(z)`
$\|z\|^2$	norm()	`double x=norm(z)`
arg(z)	arg()	`double phi=arg(z)`
addition assignment	operator +=	`w+=z`
subtraction assignment	operator -=	`w-=z`
multiplication assignment	operator *=	`w*=z`
division assignment	operator /=	`w/=z`

of floating point numbers and complex numbers. A number of important mathematical functions (`cos`, `cosh`, `exp`, `log`, `pow`, `sin`, `sinh`, `sqrt`, `tan`, and `tanh`) have also been overloaded to work with complex arguments.

Complex numbers find widespread use in physics. We will illustrate how to work with the `std::complex` class with a simple problem in planar geometry. Complex numbers can be used to represent a point in a plane, if we consider the number $u = a + ib$ to be a representation of the vector $a\hat{x} + b\hat{y}$. Note that the magnitude of the complex number $|u| = \sqrt{a^2 + b^2}$ is the length of the vector. Also, if $v = c + id$, then $uv^* = (ac + bd) + i(bc - ad)$, so that the real part of this number is the dot product of the two vectors and the imaginary part is the magnitude of the cross product, $\vec{v} \times \vec{u}$. To translate a point, we merely have to add a constant (complex) number $t = t_x + it_y$ to the point. Rotations can be carried out by multiplying by $\exp(i\phi)$, where ϕ is the angle of rotation; the sense of the rotation is counterclockwise for $\phi > 0$.

One of the earliest ways in which the value of the constant π was obtained was to compute the perimeter of a polygon inscribed within a circle. Starting with a regular hexagon, one obtains the approximate value of 3.0. One can then consider a regular inscribed polygon of 12 sides, 24 sides, 48 sides, and so forth. In 265 AD the Chinese mathematician Liu Hui found an approximation for π based on a 3,072 sided polygon, and in 480 AD Zu Chongzhi found a more accurate value based on a 12,288 sided polygon. We will compute these approximations to π by starting with a 6-sided polygon centered at the origin, oriented so that one of the vertices is at $\hat{x}$ and the other is at $\frac{1}{2}\hat{x} + \frac{\sqrt{3}}{2}\hat{y}$. The two points will be represented by complex numbers $x_0 = 1$ and $x_1 = \frac{1}{2} + \frac{\sqrt{3}}{2}i$. The length of the segment joining the two vectors is $|x_1 - x_0|$, so the total perimeter of the inscribed polygon is six times this length, which is exactly equal to 3. From this first approximation we proceed to more accurate approximations with the following steps.

- Find the midpoint of the line joining the two vertices. The complex number representing that point is the average of x_0 and x_1.
- This point lies in the middle of a segment of the polygon, but not on the circle. We can put it on the circle by normalizing it to unit length, i.e. transforming

$$\frac{x_0 + x_1}{2} \quad \rightarrow \quad \frac{x_0 + x_1}{2} / \left| \frac{x_0 + x_1}{2} \right| = \frac{x_1 + x_0}{|x_1 + x_0|}$$

- At the next step this value becomes the new x_1, and we double the number of sides. The perimeter of the inscribed polygon is again taken.

As the number of sides is increased, the perimeter converges to twice the value of π. The program shown below, called `threePi`, can be found in `CH2/EXAMPLES/THREEPI`. It obtains the value of π in three ways. First, it just prints the value of the macro `M_PI`, defined in the header file `cmath`. This illustrates the usual way of obtaining the value of π in a program—it's superior, of course, to typing in the value. The second way is the

inscribed polygon algorithm, which is essentially an illustration of how to use the class `std::complex`. The last way uses Euler's famous expression $e^{i\pi} = -1$. Taking the logarithm of both sides gives a number whose real part is zero and whose imaginary part is π. This is included as an illustration of overloaded functions that operate on complex numbers in the C++ standard library. Here is the code example:

```
#include <complex>
#include <iostream>
#include <cmath>
#include <iomanip>
#include <limits>
//
// threePi is a small program to obtain pi in three
// different ways:
//
typedef std::complex<double> Complex;
main () {

  // 1. Just print out the macro:
  std::cout << std::setprecision(16) << M_PI << std::endl;

  // 2. From the area of inner inscribed polygons.  Start
  //    with a hexagon and subdivide the arc into smaller
  //    pieces. Liu Hui(n=3072, 265 AD); Zu Chongzhi
        (n=12288, 480 AD)

  Complex x0=1.0;
  Complex x1(1.0/2.,sqrt(3.0)/2.0);
  unsigned int nsides=6;

  while (nsides < std::numeric_limits<int>::max()) {
    double lside = abs(x1-x0);
    double approx=nsides*lside/2.0;
    std::cout << "Sides "
              << nsides
              << "; approx="
              << std::setprecision(16)
              << approx
              << std::endl;

    x1=(x1+x0)/abs(x1+x0);
    nsides *=2;
  }
```

```
  // 3. Take the logarithm of the imaginary number -1:
  std::cout << std::setprecision(16)
            << imag(log(Complex(-1,0)))
            << std::endl;
}
```

The notion of encapsulation applies to classes in C++. In the case of complex numbers, one can see a clean separation between the internal structure of the number and the protocol summarized in Table 2.2 that it obeys. In the next section we describe some of the mechanisms for achieving this; for achieving, in other words, encapsulation.

2.5 Classes as user defined datatypes

C++ classes are major building blocks of modern computer codes, and we shall introduce many examples in the next chapter. What is a class though, exactly? Historically, data structures, which were mere agglomerations of related data, were the precursors to classes. For example, the four components of a quaternion can be agglomerated into a single entity called a ***structure***. A quaternion is like a four-component complex number which is useful for representing rotations in *three* dimensions, among other things. In C and C++ the structure may be declared as:

```
struct Quaternion{
  double a;
  double b;
  double c;
  double d;
};
```

The structure can be passed to functions, returned from functions, and placed into arrays. Since structures are agglomerations of data, they can also agglomerate other structures to build hierarchies of structured data.

Classes add bound functions, a.k.a. ***methods***, to structures. For example, a class called `Quaternion` may return the magnitude, $\sqrt{a^2 + b^2 + c^2 + d^2}$:

```
class Quaternion{

  public:

  // return the magnitude
  double abs() { return sqrt(a*a+b*b+c*c+d*d);}

  private:
```

```
    double a;
    double b;
    double c;
    double d;

  };
```

In furnishing this example, we had to slip in two important keywords in order for our `Quaternion` class to make sense. These are `public` and `private`, and they enforce encapsulation. These keywords are like switches, switching back and forth between two access modes, public and private. When data or a member functions are public, it means that ***client*** code can invoke them. In contrast, clients may not access private data or member functions, and the compiler generates an error whenever such access is attempted. This locks away certain parts of the class which the designer does not intend for general purpose use, thereby *enforcing* the separation between interface and implementation.

In the above example, the quaternion's components a, b, c, and d are private so, they can be neither written nor read. At the very least the clients of this class need to initialize them. This is done in a constructor, as shown in this more complete example (which also provides ***accessors*** to the data members):

```
  class Quaternion{

    public:

    // Constructor
    Quaternion (double a, double b, double c, double d):
    a(a),b(b),c(c),d(d){}

    // return the magnitude
    double abs() { return sqrt(a*a+b*b+c*c+d*d);}

    // return the components
    double getA() const {return a;}
    double getB() const {return b;}
    double getC() const {return c;}
    double getD() const {return d;}

    private:

    double a;
    double b;
```

```
    double c;
    double d;

  };
```

The syntax of the constructor may strike you as funny. We discuss it in more detail in Chapter 6. Finally, for the sake of accuracy, we mention one fact about structures and classes. While structures were originally simple agglomerations with data without methods, C++ allows developers to add methods to structures, too, making them technically nearly the same as classes. Like classes, the keywords `public` and `private` also function within structure definitions to control access. There is in fact only one difference between structures and classes: the default access, which is `public` for structures and `private` for classes. In our work we generally use structures where simple agglomeration without member functions is required.

2.6 Style guide: defining constants and conversion factors in one place

In the physical sciences we encounter several types of constants that appear frequently in our work: mathematical constants like π and e; fundamental constants like Planck's constant $\hbar$, the speed of light c, the fine structure constant α, the Hubble constant H_0, masses of elementary particles, and so forth. It is not good style to repeat the definitions of these constants in many places in the code. One common solution is to define them in a header file. Namespaces, discussed in Section 2.4, are often used in conjunction with this solution to prevent constants like α from clashing with local definitions. An excerpt from a header file defining physical constants within a namespace is:

```
#ifndef _PHYSICALCONSTANTS_
#define _PHYSICALCONSTANTS_

// This file defines physical constants.  All quantities
// are specified in SI units.

namespace physics {
 static const double hBar  =1.0545718E-34;   // m^2 kg /s
 static const double cLight=299 792 458;     // m / s
 static const double alpha =7.2973525664E-3; // dimensionless
   ...
};
#endif
```

Client code can then access variables by including the header file and referencing the variables as, for example `physics::cLight`. To input the value of a particle moving at 4/5 the speed of light one could type, for example:

```
#include  "physics.h"
...
double velocity = 4.0/5.0*physics::cLight;
```

A related task is converting units. Suppose that one was working cheifly with SI units, but one occasionally needed to input or output the value of some measure of length in mm, cm, feet, or other unit of length. The conversion factors can also be collected in a single header file and scoped within a namespace:

```
#ifndef _UNITS_
#define _UNITS_

// This file contains conversion factors from SI units to
// other common systems

namespace units {
  //
  //  Length.
  //
  static const double meter    =1.0;
  static const double cm       =0.01;
  static const double mm       =0.001;
  static const double micron   =1.0E-6;;
  static const double nm       =1.0E-9;
  static const double feet     =0.3048;
  static const km              =1000.0;
  static const mile            =1609.34;
  ...

};
#endif
```

To input constants one can then give a number and a unit:

```
#include  "units.h"
using namespace units;
double distanceToMoon=238.9E3*mile;
```

while to output a constant one can print something like,

```
#include  "units.h"
...
using namespace units;
std::cout << "The distance from earth to moon is "
          << distanceToMoon/mile
          << " miles."
          << std::endl;
```

No single file is likely to be adequate for defining all of the physical constants and conversion factors for all of physics, but centralizing definitions for particular projects can eliminate redundant definitions, confusion, and errors.

2.7 Summary

In this chapter we have surveyed built-in numerical datatypes, and discussed their representations and the limitations to numerical precision that these binary representations unavoidably imply. We also drew attention to the skillful way in which details of the representation are hidden through clever compiler design so that application programmers do not get hung up on these details. This feature, encapsulation, is also evident in the design of the C++ template class `std::complex`, which we introduced in Section 2.4, and will be part of every serious API, some of which will be introduced in the following chapter. The keywords `public` and `private` control access to methods and data within a class and enforce encapsulation. You will need to know about these as you design your own classes and class libraries, but it's also important to distinguish these access patterns while using classes provided within the C++ standard library or by third parties.

Further Reading

Basics on the representation of floating point numbers can be found in Press (2007). As we are now beginning to discuss those features of C++ which are not a part of C, we recommend a good introduction to the C++ language at this point. The online reference

`www.cplusplus.com/doc/tutorial`

is excellent, and in print works such as Bronson (2013) or Capper (1994) can be consulted. When you get a little more practiced with the basic ideas of C++, you'll start to encounter minor dilemmas about how to write your own classes. We'll scratch the

surface in upcoming chapters, and you will find much more helpful advice in Meyers (2005a and 2005b).

2.8 Exercises

1. Compute the expected machine precision of `float` and `double` datatypes, and then check your calculation with a program.
2. Compute the "stretch-factor" γ for a relativistic particle for speeds approaching the speed of light, i.e. for $\beta = v/c$=0.9, 0.99, 0.999, 0.9999.... Compute this in two ways, first as $\gamma = 1/\sqrt{1-\beta^2}$ and then as $\gamma = 1/\sqrt{(2-\epsilon)\epsilon}$ where $\epsilon = 0.1, 0.01, 0.001,$ 0.0001.... Suppose that the the fractional error in the calculation is required to be one part in one thousand or less. What is the maximum value of β for which this accuracy can be obtained, if one computes it using the former method?
3. Three points labeled $A = 3\hat{x} + 7\hat{y}$, $B = 3\hat{x} + 2\hat{y}$, and $C = 10\hat{x} + 2\hat{y}$ lie in a plane.

   a) Using `std::complex`, write a program `ex1` to compute the area of a triangle whose vertices lie at points A, B, and C. Use the dot product and/or cross product and carry out the computations with complex numbers.

   b) Create a complex number t representing a translation to the point $O = 4\hat{x} + 5\hat{y}$. Apply the translation to A, B, and C, and print out the translated vectors A', B', and C'. Check that the translation leaves your computation of the area of the triangle invariant.

   c) Create a complex number r such that $|r| = 1$ by using the `exp` function to exponentiate a pure imaginary number. Choose the imaginary number such that r represents a clockwise rotation about the origin through 45°. Apply the rotation to the vectors A', B', and C' from the previous step. Print out the rotated vectors A'', B'', and C'', and check that the rotation leaves your computation of the area invariant.

4. Modify the program in the previous exercise such that instead of computing the area of triangle ABC, it computes and prints out the location of the point of closest approach (in the plane) of segment AC to point B. Use the dot product and/or cross product and carry out the computations with complex numbers. Check that this distance is invariant to rotation and to translation.
5. Using `std::complex`, compute the position of the points $A = 3\hat{x} + 7\hat{y}$, $B = 3\hat{x} + 2\hat{y}$, and $C = 10\hat{x} + 2\hat{y}$ after they are rotated by +755° about the point $O = 4\hat{x} + 5\hat{y}$.
6. Write a program, based on the example program `threePi` in this chapter (which is available `EXAMPLES/CH2/THREEPI`). Your program should compute π from the *areas* of small isoceles triangles making up the polygon inscribed within the radius of a circle rather than the length of their base.
7. Consider the quantum mechanical problem of a particle of energy E incident upon a potential barrier in one dimension of height V, which turns on at $x = 0$. Write

a program that determines the wavefunction for all value of x, and compute the transmission and reflection coefficients, R and T, for different values of the V/E. Do not write a fundamentally different function for the case in which $E < V$ and $E > V$; instead carry out the computation in complex numbers, in a manner which is valid for both cases.

8. Two identities which relate the trigonometric functions to hyperbolic trig functions are (for x real): $\sin(ix) = i\sinh(x)$ and $\cos(ix) = \cosh(x)$. Test these relations for a few values using `std::complex`.
9. Write a function to solve the equation $ax^2 + bx + c = 0$ which returns a pair of complex values, and test this for the following values of a, b, and c:

   - $a = 1, b = 4, c = 0$
   - $a = 1, b = 4, c = 2$
   - $a = 1, b = 4, c = 6$
   - $a = 1, b = 4i, c = -6$
   - $a = 1, b = 4i, c = -4$
   - $a = 1, b = 4i, c = 0$
   - $a = 1, b = 2 + 2i, c = -6$
   - $a = 1, b = 2 + 2i, c = -0$
   - $a = 1, b = 2 + 2i, c = 6$

10. Write a function to diagonalize a 3x3 complex matrix by solving its secular equation. The function should take a 3×3 array of complex numbers as input and should output

    - An array of three complex numbers containing the eigenvalues of the matrix.
    - A 3×3 array of complex numbers, the columns of which contain normalized eigenvectors of the matrix.

    Test your function on the following matrix and provide the result:

$$\begin{pmatrix} 0.333333 & -0.244017 & 0.910684 \\ 0.910684 & 0.333333 & -0.244017 \\ -0.244017 & 0.910684 & 0.333333 \end{pmatrix}.$$

BIBLIOGRAPHY

Bronson, G. J. (2013). *C++ for scientists and engineers*. Cengage Learning.

Capper, D. M. (1994). *The C++ programming language for scientists, engineers, and mathematicians*. Springer-Verlag.

Meyers, S. (2005a). *Effective C++ 55 Specific Ways to Improve your Programs and Designs*. 3rd ed. Addison-Wesley.

Meyers, S. (2005b). *More Effective C++ 35 New Ways to Improve your Programs and Designs*. 3rd ed. Addison-Wesley.

Press, W., S. Teukolsky, W. Vetterling and B. Flannery (2007). *Numerical Recipes, the Art of Scientific Computing*, 3rd ed. Cambridge University Press.

3

Some useful classes with applications

3.1 Introduction

We have seen that one of the reasons classes are useful is because they provide a mechanism by which implementation details can be hidden from the user. In this chapter we continue the exploration of classes by applying a number of high quality classes to typical scientific problems. The practice of writing classes will be taken up again in Chapter 6. Users requiring a more structured introduction to the C++ language are directed to the references at the end of the previous chapter.

Use cases for these classes are taken from classical and quantum physics. In this chapter we carry out numerical solutions to problems which admit, possibly, analytic solutions with the idea that their familiarity makes them good learning examples.

In adopting a class library, we shall always prefer *universally* available software to *widely* available software, and widely available software to *custom* software. Universally

Applied Computational Physics. Joseph F. Boudreau and Eric S. Swanson, Oxford University Press (2018).
 DOI:10.1093/oso/9780198708636.001.0001

available software is that which is found in the C++ standard library. Most of the widely available software can be installed using package management software; on Ubuntu, this would be either the `apt-get` command or the graphical installer `synaptic`.

3.2 Coupled oscillations

Consider the problem of N beads, arranged left to right on a taut frictionless wire. They are free to move in one dimension, but they are connected to their neighbors with ideal springs. The beads are labeled with the index $i = 0, 1, 2...N-1$; the i^{th} bead has a mass of m_i; it is connected to its left neighbor by a spring constant k_i and to its right neighbor by a spring constant k_{i+1}. The first and last beads are attached on one side only. Neither the masses nor the spring constants need be taken equal in the general case. The position of the i^{th} bead with respect to its equilibrium position is denoted by $x_i(t)$.

The force on the i^{th} bead is related to the stretch of its neighboring springs, and Newton's law yields:

$$m_i \frac{d^2 x_i}{dt^2} = k_i(x_{i-1} - x_i) + k_{i+1}(x_{i+1} - x_i)\,. \tag{3.1}$$

This is a set of coupled differential equations. For the case $N = 4$ (the generalization is clear) it can be written in matrix form as:

$$\begin{pmatrix} m_0 & 0 & 0 & 0 \\ 0 & m_1 & 0 & 0 \\ 0 & 0 & m_2 & 0 \\ 0 & 0 & 0 & m_3 \end{pmatrix} \cdot \begin{pmatrix} \ddot{x}_0 \\ \ddot{x}_1 \\ \ddot{x}_2 \\ \ddot{x}_3 \end{pmatrix} = - \begin{pmatrix} k_1 & -k_1 & 0 & 0 \\ -k_1 & k_1 + k_2 & -k_2 & 0 \\ 0 & -k_2 & k_2 + k_3 & -k_3 \\ 0 & 0 & -k_3 & k_3 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \tag{3.2}$$

or

$$\mathbf{M} \cdot \ddot{\mathbf{x}} = -\mathbf{K} \cdot \mathbf{x} \tag{3.3}$$

where $\mathbf{M}$ is the mass matrix, $\mathbf{K}$ is the matrix involving the spring constants, and $\mathbf{x}$ is a column vector containing the displacement of each mass from its equilibrium position.

This matrix equation can be solved in the time-honored tradition of guessing a trial solution:

$$\mathbf{x} = \mathbf{a}\, \mathrm{e}^{-i\omega t} \tag{3.4}$$

where $\mathbf{a}$ is a constant undetermined column vector, and where we imply the real part of the right-hand side. Substituting this into Eq. 3.3, one obtains

$$\mathbf{K} \cdot \mathbf{a} = \omega^2 \mathbf{M} \cdot \mathbf{a} \tag{3.5}$$

The matrix $\mathbf{M}$ is diagonal and has positive elements only. We can define a matrix $\mathbf{M}^{1/2}$ such that $\mathbf{M}^{1/2}\mathbf{M}^{1/2} = \mathbf{M}$, and $\mathbf{M}^{-1/2}$ to be its inverse, and re-express the previous equation as:

$$\mathbf{M}^{-1/2}\mathbf{K}\mathbf{M}^{-1/2} \cdot \mathbf{M}^{1/2}\mathbf{a} = \omega^2 \mathbf{M}^{1/2} \cdot \mathbf{a} \tag{3.6}$$

We define the matrix $\mathbf{\Omega^2} \equiv \mathbf{M}^{-1/2}\mathbf{K}\mathbf{M}^{-1/2}$, and $\mathbf{b} \equiv \mathbf{M}^{1/2}\mathbf{a}$, so

$$\mathbf{\Omega^2} \cdot \mathbf{b} = \omega^2 \mathbf{b} \tag{3.7}$$

and we see that we are faced with an eigenvalue problem. The matrix $\mathbf{\Omega^2}$ is real and symmetric; it is a well-known property of such matrices that they possess N eigenvalues and eigenvectors which can be taken real. These will be denoted as $\lambda_j = \omega_j^2$ and $\mathbf{b}_j$, $j = 0, 1..N-1$. The λ's are not only real but also *positive* (Exercise 10), and therefore the ω's are real. The eigenvectors are mutually orthogonal in the case of nondegenerate eigenvalues, and they may be chosen to be orthogonal in the case of degenerate eigenvalues by a Gram-Schmidt orthogonalization procedure. They can be normalized[1] by rescaling them such that $\mathbf{b}_j^T \cdot \mathbf{b}_j = 1$. Then, the eigenvectors constitute an orthonormal set, such that:

$$\mathbf{b}_j^T \cdot \mathbf{b}_k = \delta_{jk}. \tag{3.8}$$

Suppose that the initial positions and velocities of all of the beads are known at a give time $t = 0$. The problem is to find the subsequent motion of all of the beads. There are two steps to this: determining the eigenfrequencies $\omega_j = \sqrt{\lambda_j}$ of $\mathbf{\Omega^2}$ and their corresponding eigenvectors $\mathbf{b}_j$ for $j = 0, 1 \ldots N-1$; and constructing the particular solution from these eigenvalue/eigenvector pairs. This particular solution is the linear combination:

$$\begin{aligned} \mathbf{x}(t) &= \mathrm{Re} \sum_{j=0}^{N-1} \eta_j \mathbf{a}_j e^{-i\omega_j t} \\ &= \mathrm{Re} \sum_{j=0}^{N-1} \eta_j \left(\mathbf{M}^{-1/2} \cdot \mathbf{b}_j \right) e^{-i\omega_j t} \end{aligned} \tag{3.9}$$

where $\mathbf{x}(t)$ is a column vector representing the positions of each of the beads, and η_j ($j = 0, 1..N-1$) is a set of N complex coefficients which are determined as follows. Using the orthonormality (Eq. 3.8) of the $\mathbf{b}_j$'s, we project out the coefficients:

$$\mathbf{b}_j^T \cdot \mathbf{M}^{1/2} \cdot \mathbf{x}(t) = \mathrm{Re}\left(\eta_j e^{-i\omega_j t} \right) \tag{3.10}$$

[1] Numerical methods for diagonalizing symmetric matrices usually yield orthonormal eigenvectors sorted by eigenvalue, freeing the user to think about other issues.

which at $t = 0$ reads

$$\mathrm{Re}(\eta_j) = \mathbf{b}_j^T \cdot \mathbf{M}^{1/2} \cdot \mathbf{x}^0 \,, \tag{3.11}$$

$\mathbf{x}^0$ being the initial position of the beads. Representing the velocity of the beads as a column vector

$$\mathbf{v} = \frac{d\mathbf{x}}{dt} \tag{3.12}$$

and taking the time derivative of Eq. 3.9 we have

$$\mathbf{v}(t) = \mathrm{Re} \sum_{j=0}^{N-1} -i\omega_j \eta_j \left(\mathbf{M}^{-1/2} \cdot \mathbf{b}_j \right) e^{-i\omega_j t} \tag{3.13}$$

which leads, in a similar way, to the equation

$$\mathrm{Im}(\eta_j) = \frac{\mathbf{b}_j^T \cdot \mathbf{M}^{1/2} \cdot \mathbf{v}^0}{\omega_j} \tag{3.14}$$

($\mathbf{v}^0$ indicates the initial velocities) and thus

$$\eta_j = \mathbf{b}_j^T \cdot \mathbf{M}^{1/2} \cdot \mathbf{x}^0 + i \frac{\mathbf{b}_j^T \cdot \mathbf{M}^{1/2} \cdot \mathbf{v}^0}{\omega_j} \,. \tag{3.15}$$

This relation, when substituted into Eq. 3.9 furnishes a complete solution to the program. To obtain it for a variety of configurations and initial conditions we will use a good class library for linear algebra, which is the topic of our next section.

3.3 Linear algebra with the `Eigen` package

To carry out the computations arising in the previous section, we are going to need a decent class library which provides for matrix algebra, vector algebra, and solutions to eigenvalue problems. It would be nice if powerful linear algebra classes were an integral part of the C++ standard library, because if that were the case one could have a reasonable expectation that any compiler on any platform would compile the code with no extra package installation required. However, no such set of classes has, so far, been integrated with the C++ standard.

Fortunately, a class library known as `Eigen`:

```
eigen.tuxfamily.org
```

contains a high-quality set of classes for linear algebra and geometry. It can be freely downloaded and installed on your laptop. The package can be installed (on our reference platform, Ubuntu) as follows:

```
$sudo apt-get install libeigen3-dev libeigen3-doc
```

After the installation is complete, you will find header files in the area

```
/usr/include/eigen3
```

and there are no libraries to install since all of the classes and functions provided by the `Eigen` library are ***inline***; in other words, they are expanded in the body of the calling routine rather than compiled separately and placed into a static or shared library of object code. This area should be added to the include search path (using the `-I/usr/include/eigen3` option) during compilation. Documentation can be found at `eigen.tuxfamily.org`. `Eigen` is a very complete class library which we will not describe in full detail; our goal for the moment is to familiarize you with a small portion of the library that will prove immediately useful.

`Eigen` can handle both real and complex vectors and matrices. To begin with, we consider real vectors and matrices, and return to complex vectors and matrices a little further on.

We assume that you have installed a basic linux operating system on your laptop, desktop, or other computer. Your next task is to install software development kits in order to obtain a minimal environment for compiling and running the examples given in this text, and for conducting the end-of-chapter exercises. The first external package we ask you to install is `Eigen`.

In our first example we use two classes called `Eigen::VectorXd` and `Eigen::MatrixXd` to solve a very simple matrix equation. The `Xd` suffix on these datatypes signifies that the dimensionality of the object (matrix or vector) is determined at runtime rather than compile time, and that the matrix elements are represented by `doubles`. Complex vectors and matrices are denoted (in `Eigen`) by `Eigen::VectorXcd` and `Eigen::MatrixXcd`. The `cd` indicates that `std::complex<double>` is used in the representation of the matrix elements.

Let's start by solving a simple matrix inversion problem. We seek $\vec{x}$ where $A \cdot \vec{x} = \vec{y}$ and:

$$A = \begin{pmatrix} 1.0 & 2.0 \\ 4.0 & 9.0 \end{pmatrix}$$
$$\vec{y} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$$

This can easily be accomplished as follows:

```
#include <Eigen/Dense>
#include <iostream>
int main(int argc, char **argv) {
  Eigen::VectorXd Y(2);
  Y(0)= 1.0;
  Y(1)= 3.0;
  Eigen::MatrixXd A(2,2);
  A(0,0)= 1.0; A(0,1)=2.0;
  A(1,0)= 4.0; A(1,1)=9.0;

  Eigen::MatrixXd AInv= A.inverse();
  std::cout << AInv*Y;
}
```

To obtain a solution to the coupled oscillation problem of the previous section, we need a mechanism for solving eigenvalue problems. A few remarks are necessary before starting. Suppose we have a matrix with real elements, such as the matrix `A` of the above example. It is still possible that the eigenvalues and eigenvectors are complex. Therefore we would in general be looking for complex return values. However, in the case of coupled oscillations, our matrix is real and symmetric and therefore the eigenvalues and vectors will all be real.

`Eigen` provides two classes for the solution of eigenvalue problems; the first class is called `EigenSolver` (namespace `Eigen`) and can always obtain a solution (but it may be complex), while the second is called `SelfAdjointEigenSolver` and requires self-adjoint matrices (and returns real eigenvalues and real, orthonormal eigenvectors). Self-adjoint matrices, real or complex, are those for which $A^\dagger = A$, where $A^\dagger \equiv (A^T)^*$. Real symmetric matrices, like the ones we encounter in the coupled oscillator problem, are a subset of self-adjoint matrices.

Like `std::complex`, the eigenvalue solvers are template classes, you need to furnish the datatype of the matrix which is to be diagonalized as a template parameter. Here is an example, using a simple 2×2 matrix:

```
#include <Eigen/Dense>
#include <iostream>
int main(int argc, char **argv) {

  // Find the eigenvalues and the eigenvectors
  // of the matrix
  // A  = 1.0  2.0
  //      4.0  9.0

  Eigen::MatrixXd A(2,2);
  A(0,0)= 1.0; A(0,1)=2.0;
  A(1,0)= 4.0; A(1,1)=9.0;
```

```
  // initialize an eigensolver with A

  Eigen::EigenSolver<Eigen::MatrixXd> s(A);

  // and solve

  Eigen::VectorXcd  val=s.eigenvalues();
  Eigen::MatrixXcd  vec=s.eigenvectors();
  std::cout << val  << std::endl;
  std::cout << vec  << std::endl;
}
```

The eigenvectors are returned in a complex-valued vector (`VectorXcd val`) and the eigenvectors are returned together as a matrix (`MatrixXcd vec`); each column of the matrix contains a complex eigenvector. Note that if you compute (numerically or analytically) the eigenvalues of the original matrix,

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 4 & 9 \end{pmatrix} \tag{3.16}$$

you will find that they happen to be real-valued, however `Eigen` cannot determine this in advance so the results are returned in complex datatypes. If on the other hand we have to find the eigenvalues(vectors) of

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 2 & 9 \end{pmatrix} \tag{3.17}$$

then it is known in advance from the symmetric form of the matrix that the eigenvalues and eigenvectors are real. The `SelfAdjointEigenSolver` can be used in this case; it returns its results in real-valued datatypes, `VectorXd` and `MatrixXd`. In addition, the matrix containing the eigenvectors will be orthogonal, i.e., it will satisfy the condition that $\Lambda^T \cdot \Lambda = \mathbb{I}$, the identity matrix.

You now know everything you need to solve the coupled oscillators problem. Implementing the solution is left to the exercises.

3.4 Complex linear algebra and quantum mechanical scattering from piecewise constant potentials

In our second example, we develop a computational solution to a simple problem in one-dimensional quantum mechanics. This problem is treated in almost every elementary text, for example Gasiorowicz (2003) or Griffiths (2005). Analytical solutions give insights that computational solutions cannot—and vice versa.

A particle with energy $E > 0$ is incident upon a rectangular potential barrier with barrier height of V and a spatial extent from $x = -a$ to $x = a$. The time-independent Schrödinger equation is:

$$\left[-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x)\right]\psi(x) = E\psi(x) \tag{3.18}$$

We re-write this as:

$$\left[-\frac{1}{k^2}\frac{\partial^2}{\partial x^2} + v(x)\right]\psi(x) = \psi(x), \tag{3.19}$$

where $v = V/E$ and $k^2 = 2mE/\hbar^2$.

We will assume a particle incident on the barrier from the negative-x direction, and we divide space into region I ($x < -a$), region II ($-a < x < a$), and region III ($x > a$). In regions I and III, where the piecewise constant potential is zero, the Schrödinger equation is:

$$\frac{d^2\psi(x)}{dx^2} = -k^2\psi(x) \tag{3.20}$$

while in region II it is:

$$\frac{d^2\psi(x)}{dx^2} = -n^2k^2\psi(x) \tag{3.21}$$

where

$$n \equiv \sqrt{1 - v} \tag{3.22}$$

is a sort of "index of refraction" expressing (for fixed energy) the ratio of wavenumber in a region with constant potential to the corresponding wavenumber in a potential-free region; alternately it is the inverse of the ratio of the wavefunction's phase velocity in the two regions.

The general solution to the Schrödinger equation is:

$$\psi(x) = e^{\pm ikx} \quad \text{regions I and III} \tag{3.23}$$

and

$$\psi(x) = e^{\pm inkx} \quad \text{region II}\,. \tag{3.24}$$

The particular solution to the Schrödinger equation, describing an incident wave and reflected wave in regions I and II and a transmitted wave in region III can be written as:

$$\psi(x) = Ae^{ikx} + Be^{-ikx} \qquad \text{(region I)}$$
$$\psi(x) = Ce^{inkx} + De^{-inkx} \qquad \text{(region II)}$$
$$\psi(x) = Fe^{ikx} \qquad \text{(region III)}$$

The constants A, B, C, D, and F are constrained by the continuity of the wavefunction and its first derivative at $x = -a$ and $x = a$, whence:

$$\begin{aligned} Ae^{-ika} + Be^{ika} &= Ce^{-inka} + De^{inka} \\ Ce^{inka} + De^{-inka} &= Fe^{ika} \\ ikAe^{-ika} - ikBe^{ika} &= inkCe^{-inka} - inkDe^{inka} \\ inkCe^{inka} - inkDe^{-inka} &= ikFe^{ika}\,. \end{aligned}$$

This can be written in terms of the amplitude of the incoming wave as:

$$\begin{pmatrix} e^{ika} & -e^{-inka} & -e^{inka} & 0 \\ 0 & e^{inka} & e^{-inka} & -e^{ika} \\ -ik\,e^{ika} & -ink\,e^{-inka} & ink\,e^{inka} & 0 \\ 0 & ink\,e^{inka} & -ink\,e^{-inka} & -ik\,e^{ika} \end{pmatrix} \cdot \begin{pmatrix} b \\ c \\ d \\ f \end{pmatrix} = \begin{pmatrix} -e^{-ika} \\ 0 \\ -ike^{-ika} \\ 0 \end{pmatrix} \tag{3.25}$$

where $b \equiv B/A$, $c \equiv C/A$, $d \equiv D/A$, $f \equiv F/A$. This set of equations is solved analytically in elementary textbooks; here we aim to solve it numerically with the help of complex matrix classes from the package `Eigen`. An example is furnished in the next section.

We remark that one of the advantages is that the technique can be extended easily to deal with more complicated, piecewise-constant potentials. In fact, any potential in one dimension can be treated in this way by dividing it into sufficiently small pieces. Thus the general one-dimensional problem is equivalent to inverting an infinite size matrix equation of the type just developed.

3.4.1 Transmission and reflection coefficients

The wavefunction ψ determines the particle probability density $\rho(x,t) \equiv \psi(x,t)\psi^*(x,t)$ and the probability current:

$$j(x,t) \equiv \frac{i\hbar}{2m}\left(\psi(x,t)\frac{d\psi(x,t)^*}{dx} - \psi(x,t)^*\frac{d\psi(x,t)}{dx}\right). \tag{3.26}$$

The two are related by a conservation law:

$$\frac{\partial j(x,t)}{\partial x} + \frac{\partial \rho(x,t)}{\partial t} = 0 \tag{3.27}$$

which follows from the Schrödinger equation. In the barrier potential problem, the ***transmission*** and ***reflection coefficients*** can be computed as the ratio of the magnitude of the probability current of the transmitted and reflected waves to that of the incoming wave. These are:

$$\frac{\hbar k}{m}\begin{cases} |A|^2 & \text{Incident wave} \\ -|B|^2 & \text{Reflected wave} \\ |F|^2 & \text{Transmitted wave} \end{cases}. \tag{3.28}$$

3.5 Complex linear algebra with `Eigen`

In developing a numerical solution to the problem of the previous section, we will need to manipulate complex vectors and matrices. The relevant datatypes are `Eigen::VectorXcd` and `Eigen::MatrixXcd`. The following code can be found in the directory `EXAMPLES/CH3/RECTBARRIER`. It computes the coefficients b, c, d, and f, and also computes the transmission coefficient $|f|^2$ and the reflection coefficient $|b|^2$; these should sum to unity, and this can verified as a sanity check. The inputs are V (specified in units of E) and k (specified in units of a^{-1}):

```
#include <Eigen/Dense>
#include <iostream>
#include <complex>
#include <string>
#include <sstream>
typedef std::complex<double> Complex;
int main(int argc, char **argv) {
  using namespace std;
  string usage = string("Usage: ")
    + argv[0]
    + " [-?] [-v potential] [-k wvnumber]";

  if (argc>1 && argv[1]==string("-?")) {
    cout << usage << endl;
    exit (0);
  }

  // default values
   double v=0.5;
   double k=0.2;

  try {
```

```
  // overwritten by command line:
  for (int i=1; i<argc;i+=2) {
    istringstream stream(argv[i+1]);
    if (string(argv[i])=="-v") stream >> v;
    if (string(argv[i])=="-k") stream >> k;
  }
}
catch (exception &) {
  cout << usage << endl;
  exit (0);
}

Complex I(0,1.0);
Complex nk=k*sqrt(Complex(1-v));

Eigen::VectorXcd Y(4);
Y(0)= -exp(-I*k);
Y(1)= 0;
Y(2)= -I*k*exp(-I*k);
Y(3)= 0;

Eigen::MatrixXcd A(4,4);
// First row:
A(0,0)= exp(I*k)      ;
A(0,1)=-exp(-I*nk)    ;
A(0,2)= -exp(I*nk)    ;
A(0,3)=0              ;

// Second row:
A(1,0)= 0             ;
A(1,1)= exp(I*nk)     ;
A(1,2)= exp(-I*nk)    ;
A(1,3)=-exp(I*k)      ;

// Third row:
A(2,0)= -I*k*exp(I*k)  ;
A(2,1)=-I*nk*exp(-I*nk);
A(2,2)= I*nk*exp(I*nk) ;
A(2,3)=0               ;

// Fourth row:
A(3,0)= 0              ;
A(3,1)=I*nk*exp(I*nk)  ;
```

```
  A(3,2)=-I*nk*exp(-I*nk);
  A(3,3)=-I*k*exp(I*k)   ;

  Eigen::MatrixXcd AInv= A.inverse();
  Eigen::VectorXcd BCDF=AInv*Y;
  Complex B=BCDF(0), F=BCDF(3);
  cout << "Complex coefficients" << endl;
  cout << BCDF << endl;
  cout << endl;
  cout << "Reflection coefficient  : " << norm(B)
     << endl;
  cout << "Transmission coefficient: " << norm(F)
     << endl;
  cout << "Sum:                      " << norm(B)+norm(F)
     << endl;

}
```

In addition to the operations illustrated in the above example, real and complex matrices have a rich set of other operations that can be used. Some of these appear in Table 3.1, and others can be found in the `Eigen` documentation.

Table 3.2 shows operations that apply to real or complex vectors. Note that transpose and adjoint operations return a $1\times N$ matrix and not a column vector, which is identical to a $N\times 1$ matrix.

Table 3.1 *Matrix operators.*

Operation	Function	Example
(,)	subscript	`A(2,3)`
+	operator +()	`C=A+B`
-	operator -()	`C=A-B`
*	operator *()	`C=A*B`
*	operator *()	`B=2.0*A`
-	unary -	`A=-B`
tr$A$	trace	`x=A.trace()`
$\|A\|$	determinant	`x=A.determinant()`
A^{-1}	inverse	`AINV=A.inverse()`
A^T	transpose	`AT=A.transpose();`
$A^\dagger$	adjoint	`ADagger=A.adjoint()`

Table 3.2 *Complex and real operations.*

Operation	Function	Example
()	subscript	`v(2)`
+	operator +()	`v=u+w`
-	operator -()	`v=u-w`
*	operator *()	`u=2.0*v`
-	unary -	`u=-v`
return norm	norm()	`double x=norm(u)`
return squared norm	squaredNorm()	`x2=squaredNorm(u)`
normalize in place	normalize()	`v.normalize()`
return normalized copy	normalized()	`u=v.normalized()`
v^T	transpose	`vT=v.transpose();`
$v^\dagger$	adjoint	`vDagger=v.adjoint()`

3.6 Geometry

A "vector" has multiple meanings in mathematics, physics, and computer science. On one hand it is a linear algebraic object, an *n*-dimensional tuple in a vector space, acted upon by linear transformations in that space that are represented by matrices. On the other hand (restricting ourselves to Cartesian 3-space) we identify a vector as an object with magnitude and direction, which supports the operations of cross product and dot product, which is affected by a rotation of spatial coordinates, and which are sometimes called *three-vectors*; such three-vectors may be represented on a computer by a triplet of coordinates, but one should not confuse a particular representation with the actual vector, which is a more abstract notion–a thing with magnitude and direction, which we may visualize as an arrow. Thus there are two possible abstractions: an algebraic vector or a geometric vector. In Section 3.7 we will encounter another kind of "vector", representing a computer scientist's notion of the term, and unlikely to be confused with the algebraic or geometric vectors.

Some class libraries provide different classes for each type of object. The `Eigen` class library stores both types of vector in the same object but provides the extra operation of cross product which is valid when applied to three-dimensional vectors, `Vector3f` or `Vector3d`. The former is represented by `floats`, the latter by `doubles`; and both classes are `typedef`'d template classes. The length of these objects (3) is fixed at compile time. The cross product and dot product for `Eigen::Vector3d` and `Eigen::Vector3f` are invoked like this:

```
#include "Eigen/Dense"
using namespace Eigen;
Vector3d u,v;
Vector3d w=u.cross(v);
double   d=u.dot(v);
```

The vectors interoperate with a set of transformations (rotations and affine transformations) that are defined in the header `Eigen/Geometry`. In the following section we provide a few examples of how one can use the vector classes for simple calculations.

3.6.1 Example: collisions in three dimensions

Consider two free particles in three dimensions. Particles 1 and 2 travel from positions $\vec{P}_1$ and $\vec{P}_2$ with velocities $\vec{v}_1$ and $\vec{v}_2$, respectively. Our aim is to compute how close the particles approach each other, and how close their straight trajectories become (these are not the same thing). To compute how close the two particles become, step into the reference frame of particle 1: the velocity of particle 2 in that frame is $\vec{v} = \vec{v}_2 - \vec{v}_1$, and the position of particle 2 is $\vec{r} = \vec{P}_2 - \vec{P}_1$. The distance of closest approach is the quantity $|\vec{r} \times \hat{v}|$. This can be implemented as a short function as follows:

```
#include "Eigen/Dense"
double distance(const Eigen::Vector3d & P1,
                const Eigen::Vector3d & P2,
                const Eigen::Vector3d & v1,
                const Eigen::Vector3d & v2) {
    // Compute the distance of closest approach of particle 2
    // to particle 1:

    return (P2-P1).cross((v2-v1).normalized()).norm();

}
```

To compute the distance between the two trajectories, we designate points $\vec{Q}_1$ as the point on the trajectory of particle 1 which lies closest to the trajectory of particle 2, and $\vec{Q}_2$ as the point on trajectory 2 which lies closest to trajectory 1. We can make a displacement from $\vec{P}_1$ to $\vec{P}_2$ in three steps, first, by moving an unknown distance s from $\vec{P}_1$ to $\vec{Q}_1$ along the direction $\hat{v}_1$; then by moving an unknown distance t from $\vec{Q}_1$ to $\vec{Q}_2$; then by moving an unknown distance u from $\vec{Q}_2$ to $\vec{P}_2$ along the direction $\hat{v}_2$. The distances s, t, and u are signed quantities, and the displacement from $\vec{Q}_1$ to $\vec{Q}_2$ takes place along a direction which is perpendicular to both $\hat{v}_1$ and $\hat{v}_2$, and thus it lies along the direction $\widehat{v_1 \times v_2}$. We can therefore write:

$$s\hat{v}_1 + t(\widehat{\vec{v}_1 \times \vec{v}_2}) + u\hat{v}_2 = \vec{P}_2 - \vec{P}_1$$

We are interested in knowing the quantity t, which we can obtain by dotting this expression into $\widehat{\vec{v}_1 \times \vec{v}_2}$:

$$t = (\vec{P}_2 - \vec{P}_1) \cdot (\widehat{\vec{v}_2 \times \vec{v}_1})$$

This quantity can be positive or negative, and we only care about the magnitude. In code we could write:

```
#include "Eigen/Dense"
double trajSeparation(const Eigen::Vector3d & P1,
                      const Eigen::Vector3d & P2,
                      const Eigen::Vector3d & v1,
                      const Eigen::Vector3d & v2) {

    // Compute the distance of closest approach of
       trajectory 2
    // to trajectory 1:

    return fabs((P2-P1).dot(v2.cross(v1).normalized()));

}
```

The quantities s and u can also be computed from the simultaneous solution to these two equations:

$$\begin{aligned} s + u(\hat{v}_2 \cdot \hat{v}_1) &= (\vec{P}_2 - \vec{P}_1) \cdot \hat{v}_1 \\ s(\hat{v}_1 \cdot \hat{v}_2) + u &= (\vec{P}_2 - \vec{P}_1) \cdot \hat{v}_2 \,. \end{aligned}$$

3.7 Collection classes and strings

The classes `std::vector` and `std::string` are similar in some regards to the more basic built-in array-of-object and array-of-characters. However, they do considerably more and are much easier to use. Classes like `std::vector` and `std::string` are sometimes referred to as *first-class* datatypes, while their built-in cousins are called *second-class*. First-class datatypes are so powerful and easy to use that they have practically displaced the use of their second-class cousins in most application programming.

The `std::vector` class holds objects, like an array, but it can increase its size automatically. The class is designed to simplify the storage of objects, and no mathematical operations are defined for it. The most common way to add elements to a vector is by pushing them onto the end of the array. The most common way to access the stored element is by random access using the subscripting operator, [ ].

Actually, like `std::complex`, `std::vector` is not a class, but a ***class template***, which means it's a recipe for generating a class, and here the class can be generated by specifying as a template argument what kind of object the vector holds. Templates avoid the necessity of writing multiple classes when the only real difference between them is the datatype used in the representation. To declare a vector of `doubles` one writes:

```
#include <vector>
std::vector<double> myVector;
```

To add a few `doubles` to the new vector you could write:

```
myVector.push_back(2.0);
myVector.push_back(3.14159);
myVector.push_back(7.0);
```

The vector can be copied (or passed by reference) between two functions, like typical C++ objects. The size of the vector can be retrieved, as can the elements:

```
for (int i=0;i<myVector.size();i++) {
  std::cout << myVector[i] << std::endl;
}
```

A good exercise is to type the preceding lines of code into a short program, and see what it prints; then change the template parameter from `double` to `int`, recompile, and look again at the output.

Another useful class is `std::string`. This class represents a basic string of characters (words, sentences, paragraphs), and the interface supports searching strings, appending strings (which is done with the "+" operator), and editing strings. A `std::vector` can hold not only `ints`, `floats`, and `doubles`, but any kind of object, such as a string. We modify the code written above, transforming our vector-of-doubles into a vector-of-strings.

```
#include <vector>
#include <string>
main() {
  std::vector<std::string> myVector;
  myVector.push_back("Trout ");
  myVector.push_back("Fishing ");
  myVector.push_back("In ");
  myVector.push_back("America.");
  for (int i=0;i<myVector.size();i++) {
    std::cout << myVector[i] << std::endl;
  }
}
```

which outputs

```
Trout Fishing In America.
```

The class `std::vector` has a lot of other functionality, which you can explore yourself or by consulting a good text such as Weiss (2014). The `std::vector` class is part of the C++ standard library, which also contains more specialized container classes such as `deque`, `list`, `set`, `map`, and `multimap`, which are also very useful, and make up the so-called Standard Template Library(STL). The STL (Stepanov, 1995) was once a separate package but is now fully integrated into the C++ standard library.

3.8 Function objects

Function-objects, or ***functors***, are a type of first-class object that overload the function-call operator, `operator()`. The unusual name is not a C++ keyword of any kind, but just a descriptive name given to such classes. Our functors come from the `QatGenericFunctions` library, part of the ***Qat libraries***, which is custom software developed by the authors and available through the companion website:

```
http://www.oup.co.uk/companion/acp
```

We shall discuss and use it throughout the book.

> Go to `http://www.oup.co.uk/companion/acp` and follow instructions to install the Qat packages.

The `QatGenericFunctions` library provides many classes, most of which represent functions of one or more variables. These objects are similar to their second-class cousins, normal C/C++ functions, but they do more. Probably the simplest of these is the class `Variable`, (namespace `Genfun`) which we will use to illustrate the most important use cases. The function-call operator for this class simply returns its argument:

```
#include "QatGenericFunctions/Variable.h"
#include <iostream>
main() {
  Genfun::Variable X;
  std::cout << X(3.14) << std::endl;
}
```

The program prints out 3.14. The object `X` can be used in expressions, and these expressions can be assigned to a datatype called `Genfun::GENFUNCTION`. The following

expression manufactures the function $f(x) = 1 + 2x + x^3$; it prints out a table of the function and its derivative at three points:

```
main () {
  Genfun::Variable X;
  Genfun::GENFUNCTION f=1 + 2*X + X*X*X;
  std::cout << 1 << " " << f(1) << " " << f.prime()(1) << std
      ::endl;
  std::cout << 2 << " " << f(2) << " " << f.prime()(2) << std
      ::endl;
  std::cout << 3 << " " << f(3) << " " << f.prime()(3) << std
      ::endl;
}
```

The class library contains basic functions (`Sin`, `Exp`, `Gamma`) and not-so-basic functions, shown in Table 3.3. These functions can all be used interchangeably and combined in expressions as here:

```
#include "QatGenericFunctions/Sin.h"
.
.
.
{
  Genfun::Sin sin;
  Genfun::GENFUNCTION f=(1 + sin)/2;
}
```

Table 3.3 *List of the most basic functions in the `QatGenericFunctions` library.*

Name	Description
Variable	Basic building-block of expressions
F1D	General purpose adapter to any function
FixedConstant, Power, Sqrt, Square	Powers
Sigma	Sum of component functions
ArrayFunction	Defined by an input array
Sin, Cos, Tan, ASin, ACos, ATan	Trig and inverse functions
Sinh, Cosh, Tanh, ASinh, ACosh, ATanh	Hyperbolic Trig and inverse functions
Exp, Log, Gamma, LGamma, Erf	exponential, log, gamma, and error functions

Note, the `sin` *functor* is here given the same name as the `sin` *function* from the math library, it ***shadows*** that function (i.e. makes it invisible). The function `sin` can still be referenced as `std::sin`.

Another more compact way to write the same function is:

```
#include  "QatGenericFunctions/Sin.h"
{
  .
  ..
  Genfun::GENFUNCTION f=(1 + Genfun::Sin())/2;
}
```

How, you may ask, does one express a function like sin 3x? Since `QatGenericFunctions` allows function composition, you can express it like this:

```
#include  "QatGenericFunctions/Sin.h"
#include  "QatGenericFunctions/Variable.h"
{
  .
  ..
  Genfun::Variable X;
  Genfun::Sin      sin;
  Genfun::GENFUNCTION f=sin(3*X);
}
```

or more compactly as:

```
#include  "QatGenericFunctions/Sin.h"
#include  "QatGenericFunctions/Variable.h"
{
  .
  ..
  Genfun::GENFUNCTION f=Genfun::Sin()(3*Genfun::Variable());
}
```

To define a function, you can either manufacture it out of more primitive functions as illustrated above; or you can introduce your own extender functors–which we will cover later since this is a little more difficult. Alternately, you can use the adapter class F1D, which endows any function `double f(double x)` with algebraic properties, in particular the operations (+,-,*, /), as well as composition $f(g(x))$ and the derivative. For example, here we turn the hyperbolic tangent, `tanh`, from the standard c++ library, into a functor:

```
#include "QatGenericFunctions/F1D.h"
Genfun::GENFUNCTION tanh=Genfun::F1D(std::tanh);
```

With this, we can now carry out function-arithmetic in the same way that you are used to carrying out floating point arithmetic. For example, the following expression

```
Genfun::GENFUNCTION f=1-tanh*tanh;
```

defines a functor that computes $f(x) = \text{sech}^2(x)$–something that is impossible using raw C/C++ function pointers.

There is one important difference between functors defined with the adapter class `F1D`, and functors that are predefined in the `QatGenericFunctions` library: usually the predefined functors return analytical derivatives, while the adapters return a derivative computed numerically. Thus the adapter is inferior to either predefined functors, or properly implemented user-defined extender functors.

3.8.1 Example: root finding

An equation of one variable $F(x) = 0$ can have zero, one, or many solutions, commonly called roots of the equation. Finding these in the most general case of a nonlinear equation can be challenging. One of the commonly used algorithms can be traced back to Isaac Newton and Joseph Raphson, and is now called the ***Newton-Raphson method***. It is an iterative method, in which one proceeds from an initial estimate which is then refined until convergence is achieved. One takes the tangent to the curve at the starting point, and intersects it with the x-axis. The intersection provides a refined estimate of the root. This continues until the estimate is invariant under further refinement. Note that the algorithm may not converge if a poor starting point is used, so good initial estimates are essential. The Newton-Raphson method is therefore perhaps more accurately described as an algorithm for refining estimates rather then obtaining them from scratch.

The line tangent to $F(x)$ is obtained by taking the first derivative $F'(x)$, and the intersection with the x-axis is given by $x - F(x)/F'(x)$. The whole algorithm can be coded like this:

```
double newtonRaphson(double x, Genfun::GENFUNCTION P) {
  double x1=x;
  while (1) {
    double deltaX=-P(x)/P.prime()(x);
    x+=deltaX;
    if (float(x1)==float(x)) break;
    x1=x;
  }
  return x;
}
```

The algorithm iterate until convergence is achieved at `float` precision.

The case of multiple roots is more complicated, but after each root is found one can simplify the discovery of the remaining roots. The method, called deflation, is to divide the function $f_i(x)$ by $(x - x_i)$ after each root x_i is found: $f_i(x) \to f_{i+1}(x) = f_i(x)/(x - x_i)$. The updated function $f_{i+1}(x)$ is zero wherever $f_i(x)$ is zero, except at $x = x_i$; and one

can locate the remaining roots by evaluating $f_{i+1}(x)$. To start the procedure one sets $f_0(x) = F(x)$. The process, applied to a fifth order polynomial, is illustrated in Figure 3.1. This example can be found in `EXAMPLES/CH3/NRDEFLATE`. The key elements of this code are summarized in this excerpt::

```
using namespace Genfun;
Variable X;
GENFUNCTION F=(X-1)*(X-2)*(X-3)*(X-M_PI)*(X-4);
const AbsFunction * f=&F;
for (int i = 0; i<5; i++ ) {
  double x = newtonRaphson(-1.0, *f);
  GENFUNCTION F1 = (*f)/(X-x);
  if (f!=&F) delete f;
  f=F1.clone();
}
```

Note that at each iteration of the Newton-Raphson step, we have been careful to step away from the root we have just found; that's because evaluation of the deflated function will yield a divide-by-zero error there. Exercise 15 gives some practice with this algorithm, which is frequently required in other exercises throughout the book. A more complete discussion of these methods can be found in Press (2007).

Eigenvalue methods

In Exercise 10 of Chapter 2, you are asked to diagonalize a matrix by solving a secular equation. If you attempted this exercise, you will have noticed that the `Eigen` classes, particularly `EigenSolver<MatrixXd>` provide a far simpler mechanism. So, you may ask, rather than finding the roots of a polynomial as a means of diagonalizing a matrix, can we diagonalize a matrix as a means of finding the roots of a polynomial? The answer is yes.

A `monic` polynomial is a polynomial whose highest order monomial appears with a coefficient of 1, for example $x^4 - 4x^2 - \frac{1}{2}x - 1$. Obviously any polynomial can be expressed as a numerical factor times a monic polynomial. For purposes of root finding we can ignore the factor, while expressing the monic polynomial as:

$$p(x) = x^n + \sum_{i=0}^{n-1} c_i x^i \tag{3.29}$$

Now define the ***companion matrix*** for the polynomial

$$\mathbf{C} = \begin{pmatrix} 0 & 0 & . & . & . & 0 & -c_0 \\ 1 & 0 & . & . & . & 0 & -c_1 \\ 0 & 1 & . & . & . & 0 & -c_2 \\ . & . & . & & & 0 & . \\ . & . & & . & & 0 & . \\ . & . & & & . & 0 & . \\ 0 & 0 & . & . & . & 1 & -c_{n-1} \end{pmatrix} \tag{3.30}$$