Computational Physics

Second Edition

Darren Walker

Essentials of Physics Series

COMPUTATIONAL PHYSICS

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book (the "Work"), you agree that this license grants permission to use the contents contained herein, but does not give you the right of ownership to any of the textual content in the book or ownership to any of the information or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher.* Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

Mercury Learning and Information ("MLI" or "the Publisher") and anyone involved in the creation, writing, or production of the companion disc, accompanying algorithms, code, or computer programs ("the software"), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to insure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold "as is" without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book, and only at the discretion of the Publisher. The use of "implied warranty" and certain "exclusions" vary from state to state, and might not apply to the purchaser of this product.

Computational Physics

Second Edition

Darren J. Walker

Mercury Learning and Information
Dulles, Virginia
Boston, Massachusetts
New Delhi

Copyright ©2022 by Mercury Learning And Information LLC. All rights reserved.

Original title and copyright: *Computational Physics: An Undergraduate's Guide, 2/E.* Copyright ©2021 by D.J. Walker. All rights reserved. Published by Pantaneto Press.

This publication, portions of it, or any accompanying software may not be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means, media, electronic display or mechanical display, including, but not limited to, photocopy, recording, Internet postings, or scanning, without prior permission in writing from the publisher.

Publisher: David Pallai
Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
(800) 232-0223

D. J. Walker. *Computational Physics, Second Edition.*
ISBN: 978-1-68392-832-4

The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products. All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks, etc. is not an attempt to infringe on the property of others.

Library of Congress Control Number: 2021953014

222324321 Printed on acid-free paper in the United States of America

Our titles are available for adoption, license, or bulk purchase by institutions, corporations, etc. For additional information, please contact the Customer Service Dept. at 800-232-0223(toll free).

All of our titles are available in digital format at *academiccourseware.com* and other digital vendors. The sole obligation of Mercury Learning and Information to the purchaser is to replace the book or disc, based on defective materials or faulty workmanship, but not based on the operation or functionality of the product.

To Charlotte

CONTENTS

Chapter 1:	**Introduction**		**1**
	1.1	Getting Started with Coding	1
	1.2	Getting To Know The Linux Command Line	3
	1.3	Bonjour Tout Le Monde	6
	1.4	The Rest of the Book	12
Chapter 2:	**Getting Comfortable**		**15**
	2.1	Computers: What You Should Know	15
		2.1.1 Hardware	15
		2.1.2 Software	17
		2.1.3 Number Representation and Precision	19
	2.2	Some Important Mathematics	24
		2.2.1 Taylor Series	25
		2.2.2 Matrices: A Brief Overview	27
	Exercises		32
Chapter 3:	**Interpolation and Data Fitting**		**35**
	3.1	Interpolation	35
		3.1.1 Linear Interpolation	35
		3.1.2 Polynomial Interpolation	38
		3.1.2 Cubic Spline	43
	3.2	Data Fitting	45
		3.2.1 Regression: Illustrative Example	45
		3.2.2 Linear Least Squares: Matrix Form	48
		3.2.3 Realistic Example: Millikan's Experiment	50
	Exercises		53

Chapter 4:	**Searching for Roots**	**55**
	4.1 Finding Roots	55
	4.1.1 Bisection	56
	4.1.2 Newton–Raphson	58
	4.1.3 Secant	60
	4.2 Hybrid Methods	62
	4.2.1 Bisection–Newton–Raphson	62
	4.2.2 Brute Force Search	64
	4.3 What's The Point of Root Searching?	65
	4.3.1 The Infinite Square Well	65
	4.3.2 The Finite Square Well	69
	4.3.3 Programming the Root Finder	72
	Exercises	77
Chapter 5:	**Numerical Quadrature**	**81**
	5.1 Simple Quadrature	82
	5.1.1 The Mid-Ordinate Rule	82
	5.1.2 The Trapezoidal Rule	83
	5.1.3 Simpson's Rule	84
	5.2. Advanced Quadrature	85
	5.2.1 Euler–Maclaurin Integration	85
	5.2.2 Adaptive Quadrature	86
	5.2.3 Multidimensional Integration	90
	Exercises	93
Chapter 6:	**Ordinary Differential Equations**	**95**
	6.1 Classification of Differential Equations	96
	6.1.1 Types of Differential Equations	96
	6.1.2 Types of Solution and Initial Conditions	98
	6.2 Solving First-Order ODEs	99
	6.2.1 Simple Euler Method	99
	6.2.2 Modified and Improved Euler Methods	102
	6.2.3 The Runge–Kutta Method	104
	6.2.4 Adaptive Runge–Kutta	107
	6.3 Solving Second-Ordered ODEs	108
	6.3.1 Coupled 1st Order ODEs	108
	6.3.2 Oscillatory Motion	110
	6.3.3 More Than One Dimension	116
	Exercises	117

Chapter 7:	**Fourier Analysis**		**119**
	7.1	The Fourier Series	120
	7.2	Fourier Transforms	124
	7.3	The Discrete Fourier Transform	127
	7.4	The Fast Fourier Transform	129
		7.4.1 Brief History and Development	129
		7.4.2 Implementation and Sampling	130
	Exercises		135
Chapter 8:	**Monte Carlo Methods**		**137**
	8.1	Monte Carlo Integration	137
		8.1.1 Dart Throwing	137
		8.1.2 General Integration Using Monte Carlo	143
		8.1.3 Importance Sampling	146
	8.2	Monte Carlo Simulations	148
		8.2.1 Random Walk	148
		8.2.2 Radioactive Decay	154
	Exercises		156
Chapter 9:	**Partial Differential Equations**		**159**
	9.1	Classes, Boundary Values, and Initial Conditions	160
	9.2	Finite Difference Methods	164
		9.2.1 Difference Formulas	165
		9.2.2 Application of Difference Formulas	168
	9.3	Richardson Extrapolation	174
	9.4	Numerical Methods to Solve PDEs	178
		9.4.1 The Heat Equation with Dirichlet Boundaries	178
		9.4.2 The Heat Equation with Neumann Boundaries	190
		9.4.3 The Steady-State Heat Equation	193
		9.4.4 The Wave Equation	196
	9.5	Pointers To The Finite Element Method	199
	Exercises		200
Chapter 10:	**Advanced Numerical Quadrature**		**203**
	10.1	General Quadrature	203
	10.2	Orthogonal Polynomials	207
	10.3	Gauss–Legendre Quadrature	210

10.4 Programming Gauss–Legendre 214
10.5 Gauss–Laguerre Quadrature 217
Exercises 219

Chapter 11: Advanced ODE Solver and Applications 221

11.1 Runge–Kutta–Fehlberg 221
11.2 Phase Space 225
11.3 Van Der Pol Oscillator 227
11.3.1 Van der Pol in Phase Space 227
11.3.2 Van der Pol FFT 228
11.4 The “Simple” Pendulum 230
11.4.1 Finite Amplitude 231
11.4.2 Utter Chaos? 233
11.5 Halley’s Comet 235
11.6 To Infinity and Beyond 237
11.7 To The Infinitesimal and Below 242
Exercises 247

Chapter 12: High-Performance Computing 251

12.1 Indexing and Blocking 252
12.1.1 Heap and Stack 252
12.1.2 Computer Memory 255
12.1.3 Loopy Indexing 257
12.1.4 Blocking 259
12.1.5 Loop Unrolling 262
12.2 Parallel Programming 263
12.2.1 Many (Hello) Worlds 264
12.2.2 Vector Summation 266
12.2.3 Overheads: Amdahl versus Gustafson 268
Exercises 272

Bibliography 275

Appendix: A Crash Course in C++ Programming 279

Index 333

CHAPTER 1

INTRODUCTION

Computational physics sits at the juncture of arguably three of the cornerstone subjects of modern times, physics, mathematics, and computer science. Many see it as sitting between theoretical physics, where there is a focus on mathematics and rigorous proof, and experimental physics, which is based on taking observations and quantitative measurements. The computational physicist performs numerical experimentation within the confines of the computer environment, applying mathematics to both simulate and examine complex models of physical systems. Just as the theoretician needs to master analytical mathematics, the experimentalist requires a working knowledge of laboratory apparatus, so does the computational physicist need to know about numerical analysis and computer programming. Any of these skills require (significant) practice to master but it is up to the physicist to know how to use them to interpret and, ultimately, understand the physical universe.

1.1 GETTING STARTED WITH CODING

You need two things to produce a computer program:

1. A text editor in which to write all your code in whatever language you choose.
2. A compiler to convert the code you have written into machine language (binary executable).

There are two methods by which you can write computer programs. The first method is via command-line control whereby you explicitly type in commands to compile a source code file written in a text editor. The second method uses what is called an Integrated Development Environment (IDE) that is essentially a compiler and text editor wrapped up into one neat application, for example, Microsoft's Visual Studio. I would suggest trying out different text editors and IDEs to discover what suits you best. If your university uses Unix-based operating systems and you find it easier to code on those machines but do not want to splash out either on a Unix based machine (though the Raspberry Pi is reasonably priced) at home or make your Windows PC dual-booting (it can run either a Unix OS or Windows OS on one machine) an alternative is Cygwin. Cygwin creates a Unix type feel on a Windows PC and it's free to download and install. Cygwin also comes with many different optional libraries and programs that are extremely useful to scientific programming, including the linear algebra package (LAPACK) library and Octave, a free alternative to MATLAB. If you can get your hands on a student version of MATLAB, I recommend you use it as it is a powerful programming tool and can be used to find quick programming solutions to problems, or as a first step towards a solution. A further alternative is to use a Virtual Machine.

For a list of freely available text editors just use your favorite search engine. Emacs is a popular programming text editor and is the default editor on most Unix-based machines; Cygwin also contains the GNU version of Emacs. On Windows you could use Notepad, however, it does not have any of the functionality of text editors specifically designed for coding. For example, programming languages have certain keywords reserved that have special meaning, for example, *if*, *for*, and *while* to name but a few. Once written these keywords are automatically distinguished from the rest of the text in some way, different color, different font, bolded, and so on. In Notepad all you will get is the same black text on a white background, which is not useful for reading and debugging the code you have written. Notepad++ is a good (and free) programming text editor for Windows that supports multiple languages.

If you prefer to use IDEs, there are a number available that are free to use. Some of these only support one language, for example,

Dev C++, whereas others support multiple languages, for example, NetBeans, Code::Blocks or Eclipse. Microsoft do a "Community" version of their Visual Studio IDE which is free to use, as is the arguably more powerful Visual Studio Code.

Most of the code that accompanies this book has been written using the C++ programming language (C++ 11 onwards), using the Eclipse IDE for C/C++ Developers. I will not review the merits of the different programming languages here as the differences only really come into their own once you start to consider high-performance computing, Web applications, game programming, or other more specific applications. The basics of programming are sufficiently covered using just one language. That said, please be aware of different programming languages and how they can be used to produce different applications. For a challenge, you could convert the programs in this book into another language.

The next section gives a crash course in using the Linux command line.

1.2 GETTING TO KNOW THE LINUX COMMAND LINE

On modern operating systems a terminal emulator is a program that allows the use of the terminal in a graphical interface. In a Linux system, the shell is a command-line interface that interprets the user's commands and passes them on to the underlying operating system. There are several shells in current use, such as the Bourne-Again shell (bash) or The C shell (tcsh), and each has its own set of features and operations, but all provide a means to interact with the machine.

When you open a new terminal emulator window the command prompt will be at the home directory (synonymous with "Folder" on Windows) of the current user. The information displayed in the command prompt is customizable by the user but typically consist of the user's username, the host machine name, the current directory, and is ended by the prompt symbol. For an example of what this looks like please see Figure 1.1 that shows a macOS terminal.

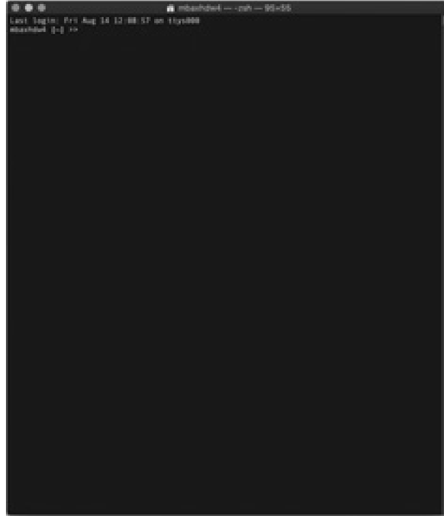

FIGURE 1.1: Example of a Linux terminal emulator.

Commands can be issued after the command prompt by typing the name of an executable file, which can be a binary program or a script. There are many standard commands that are installed as default with the operating system that allows for system configuration, file system navigation, creation of new directories and files, installing third party programs, among other operations.

A useful command to start off with is *pwd*. It displays the full path to the current, working directory and can be useful if we ever get lost in the directory structure. The *ls* command will list, on the terminal, all the files and subdirectories of the current directory. Commands can also take arguments and options (or flags) that can affect their behavior. For instance, *ls -l* will nicely format the files and subdirectories with additional information such as attributes, permissions, sizes, and modification dates. The *cd* command is typically passed an argument of the directory to which we would like to navigate. For example, *cd foo/bar* will navigate to the subdirectory *bar* of the directory *foo*, assuming *foo* is a subdirectory of the current directory. The command *cd* alone will navigate us back to the user's home directory. The Linux file system has two symbols reserved to represent the current directory and the parent directory

namely "." and "..," respectively. Issuing *cd .* does not do much but *cd ..* will take us up one level into the parent directory. Using our example if we were in the *bar* subdirectory and we issued *cd ..* our current directory would now be *foo*. All these commands and more can be found on the infamous Linux Man pages (*linux.die.net/man/* or *man7.org/linux/man-pages/index.html*) that outline all the possible options and arguments for these commands take. The man pages are daunting at first but once you learn how to read them offer a particularly useful resource when discovering or reusing various Linux commands.

The following table summarizes some of the more common commands you will likely use:

Command	**Examples of use**
mkdir	**`mkdir foo`** creates a directory called foo in the current directory
	`mkdir foo/bar` creates a directory called bar in the directory foo
cd	**`cd`** changes the current directory to the home directory
	`cd bar` changes the current directory to subdirectory bar
	`cd ..` changes the current directory to one level up
rmdir	**`rmdir foo/bar`** removes directory bar from directory foo, if bar is empty
ls	**`ls`** lists directories and files in current directory
	`ls foo` lists the directories and files in subdirectory foo
pwd	**`pwd`** displays current location within the tree
touch	**`touch foo.log`** if foo.log does not exist creates file "foo.log" in the current directory, else modifies the file's timestamps
rm	**`rm foo.log`** deletes the file "foo.log" in the current directory

1.3 BONJOUR TOUT LE MONDE

This section explains the basics of C++ syntax and structure. If you are already familiar with the C++ language, then please skip this section. A more in-depth introduction to the C++ language can be found in the Appendix.

The *Hello World* program is typically the first one that anyone learning a programming language gets to write. It gives us the basic syntax of a particular language and how to output something to the terminal or console. To begin let us first create a suitable directory structure to hold our code. Start by creating a "root" directory for our project called *CompPhys* in your home directory. Change into our newly created directory and create a subdirectory called *Hello-World*. Change into that directory and create the file *helloworld.cpp*. We now want to edit that file to fill it with the ground-breaking code for our hello world program.

Open the *helloworld.cpp* file in your text editor (or IDE) of choice and type the following:

```
// Helloworld program – displays message on stdout
#include <iostream>
int main () {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

Let us examine this line-by-line. At the very top, we have a comment line. These are either started using a double forward slash for a single-line comment or for a multiline (block) comment anything between "/*" and "*/" is treated as a comment. Comments are important. They should be used to explain the intention of code where this is not obviously apparent from the code itself. I urge you to use comments liberally; can you remember what you were doing yesterday, last week, last month, last year?

The next line down is how we include header files in source files. The hash symbol "#" is used to send instructions or directives to the pre-processor that is run before the compiler. In this case,

we are using the directive *include* to insert the contents of the file *iostream.h* at this point in the source file, and that's all it does. Note that *iostream.h* is a standard library header and as such we can drop the dot "h" extension from the filename when including it in source code. The angled brackets tell the preprocessor to look for the file in the standard external locations and implementation specified include directories only. A pair of double quotes around an included file tells the pre-processor to check the local directory of the source code first before checking the other locations. Generally, angled brackets are used for standard and system headers, and double quotes are used for programmer-defined headers.

White space is typically ignored by the compiler but is useful to humans by writing code that is easier to read. Having at least one blank line between the header include statement(s) and the start of the *main* function definition nicely separates the different parts of the source code.

The *main* function is the entry point to our program. It is this function that is called by the shell to process the code therein. As such it must be defined as a function that returns an integer value and the keyword *int* is that which represents the integer type in C and C++. The value of the returned integer indicates the "exit status" of the program; it is a convention that zero indicates success and that any other value indicates failure. The empty parenthesis tells the compiler that the *main* is a function declaration. Without them *main* would just be an integer variable declaration. We will discuss the different types of declarations later on.

The curly braces contain the definition of the main function and give us our first idea of scope. All that is contained within the curly brackets is *scoped* to the main function, but more on scope later. We also call the contents between curly brackets a block; it is a block of statements. Statements are the instructions we ask of the computer in order to perform particular tasks. For example, *int x = 1;* states that we wish to create an integer "x" and initialize it with the value of one. Statements are terminated using a semicolon that tells the compiler that we have reached the end of the current statement and will be beginning a new statement unless we have reached the end of the return statement. It is quite common amongst beginner

C/C++ programmers to forget to use the semicolon to terminate a statement (and for more experienced programmers for that matter). Fortunately, most IDE's worth their salt will highlight a syntax fault, such as a missing semicolon, in the source code editor. If not, the compiler will definitely "highlight" the error for sure.

The first statement in our *Hello World* program uses the C++ standard output stream object *std::cout* in conjunction with the output stream operator "<<" to send to the standard output (typically the terminal or console) the string sequence *Hello World*. Here I have introduced many new concepts namely streams, objects, and operators which we will get to in due course. All you need to know for the present is that this is how we can output data from a C++ program. Note that output from a program is stored in a data buffer and will only be printed on the terminal once that buffer is full, or the output is "flushed." A flush means that a program will produce a line of output immediately. The *std::endl* is a stream "manipulator" that inserts a newline character into the output sequence *and* flushes the sequence.

As an aside the *::* (double colon) is called the scope resolution operator. It is used to resolve the names of code elements (classes, functions, variables) that are contained in different *namespaces*. For example, *cout* and *endl* are scoped to the *namespace* "std," short for standard. If we had omitted the namespace and scope resolution operator in the Hello World program, we would have received a compilation error on building the binary; remove the namespace and scope resolution operator from *cout* and *endl* to see the specific compilation error. The point of the namespace feature in C++ is to avoid naming clashes between various code elements, which is especially useful when developing large projects that might involve several third-party libraries. A more detailed discussion of the use of *namespaces* can be found in the Appendix.

The *return* statement defines the exit point of the function. In this case, we are returning the integer value zero to the calling environment (the shell) to indicate the successful execution of the program. At this point a program will flush all of its output buffers meaning any remaining data will be printed to the terminal; in our case, the buffer is already empty due to the use of the *std::endl* stream manipulator.

The code now has to be compiled into binary so that the computer can read and execute the instructions. In order to do this, we need a compiler. Most Linux distributions will come with Gnu's C++ compiler installed as standard. To check that it is installed, and discover what version you have, type on the command line:

```
g++ -v
```

If the terminal output reads something along the lines of "g++ command not found" you will need to install the compiler; refer to your specific OS manual. Once you have confirmed the installation of the compiler, ensure you are in the same directory as your "helloworld" source file and type the following at the prompt:

```
g++ helloworld.cpp -o helloworld
```

All being well the compiler will have translated your source code contained within the helloworld.cpp file and produced a binary or executable file called "helloworld." The "-o" flag tells the compiler to name the executable as the text you specify after the flag. If you do not give an output name, the file gets saved as "a.out" by default. To run this program type at the command prompt

```
./helloworld
```

and it will print to the command terminal the text "Hello World." Note that if you are doing this in Windows system, binary executables are given a ".exe" extension.

We can now edit our source code to make the program more sophisticated, though only a little. Open the helloworld.cpp source file and modify the code as follows:

```
// Helloworld program – displays message on stdout
#include <iostream>
int main (int argc, char * argv[]) {
   if (argc < 2) {
std::cerr << "usage: " << argv[0] << " <name>";
          return -1;
}
```

```
    std::cout << "Hello " << argv[1] << std::endl;
    return 0;
}
```

Our program can now access the so-called *command-line arguments*, the parameters that are given on the command line after the program name when it is run. The *argc* variable contains the total number of command-line arguments, and the individual values of those arguments are available as strings accessed via the array *argv*. Note that the program name is counted as a command-line argument and is accessed as the first entry in the array, *argv[0]*; array indexing starts at zero for both C and C++. Compile this modified code and have a play with the command line arguments to gain an understanding of how it works. Note the use of a new stream object *std::cerr* to print an error message for when we do not pass sufficient arguments to the program. When a program is started by the shell it normally gains three open file descriptors: descriptor 0 is standard input, descriptor 1 is standard output, and descriptor 2 is the standard error. These descriptors are usually connected to the shell terminal that started the program but can be redirected to separate locations. As you may have guessed *std::cout* uses the standard output descriptor and *std::cerr* uses the standard error descriptor. The standard (input) stream object *std::cin* uses the standard input descriptor and can be used as an interactive means to obtain input from the user. Feel free to modify the *Hello World* program to use *std::cin* in some way; I find *cppreference.com* a useful resource.

What you have just done is a simple development cycle. Software development generally goes through three main phases:

1. Edit,

2. Compile, and

3. Execute.

Editing means writing or modifying the program contained in a text file using a text editor or IDE.

Compiling means translating the program to executable code, at this phase the code is checked for syntax errors and if found these

are flagged for additional editing. We may also get linker errors at this stage.

Executing the program means running the code on your machine. At this phase, we may get logical errors or errors with the semantics (the meaning) of our code. These are known as runtime errors, and it is the process of debugging that removes these errors or bugs.

As you write computer programs in C++ and other languages, you will repeatedly edit, compile, and run programs. Sometimes the compiler will give you error messages. Often the messages can be quite cryptic. Just as trainee doctors learn best by giving known diseases to patients it is a good idea to make some deliberate errors, to get a feel for what the errors that you will encounter in the future might mean. This knowledge of deliberately inflicted "diseases" and observing their "symptoms" should help you diagnose future errors more effectively.

Errors in the semantics of your code can be quite insidious, especially if they cause undefined behavior; it is these runtime errors that we refer to as bugs. For instance, some piece of code complies with no syntax errors, linker errors, or any warnings. The code also runs with no logical errors and exits successfully. But the results you get are garbage. This will be highly likely due to a misunderstanding of the programmer as to how a particular feature of the language works, or a mistake in the use of a third-party library. Whatever the problem is, to fix the bug, you will first need to find its location within your code. To do this we step through the code line-by-line and observe how the data contained in the program variables change with each instruction. It is this stepping and observing a process that is called debugging. Most Linux distributions will come with the GNU debugger tool installed, invoked using *gdb* on the command line, and I highly recommend you familiarize yourself with its use. Note that this is an extremely brief exposition of debugging and to go into its details is beyond the scope of this book.

Just a quick word on nomenclature. A "program" can refer either to the source code contained in a text file or the binary executable file itself, they are semantically the same thing. The source code is readable by humans, the binary is readable by machines. An

instance of a running program is called a "process" such that you can have several processes running the same program.

1.4 THE REST OF THE BOOK

Each chapter describes an individual topic within the general subject area of computational physics. Where there is a cross over between topics this has been explicitly referred to in the text. Throughout the remaining chapters, there are frequent references to C++ files that contain example programs for your study and use. These can be found on GitHub, an online repository for all sorts of different coding projects and applications, at the following URL: *github.com/DJWalker42/laserRacoon*.

These source code files come with GNU Makefiles such that compiling the code can be done by just typing "make" in the appropriate directory. These were developed on Mac OSX so contain variables specific to that operating system. You will have to modify some of the variables if you have a different OS. For more information on the GNU Makefile framework go to: *gnu.org/software/make/manual/make.html.*

The *laserRacoon* library makes use of OpenCV for a "visualization" module. If you do not have OpenCV installed on your system you can either use your install manager to get a copy (the library has been tested with OpenCV3) or visit their official site, opencv.org, for more options. If you cannot get OpenCV or would prefer not to use the visualization module then you will have to remove the related header and source files from the library (*Visualise.h* and *Visualise.cpp*) and remove any use of that module from the programs provided (anything using **`namespace`** phys::visual). Note that OpenCV is not really plotting software; OpenCV is an open-source library that performs image processing, video analysis, object and feature detection, camera calibration, 3D reconstruction, among other functions. At the time of writing the *laserRacoon* library, I needed a built-in way of visualizing the data being produced by the C++ programs. I had some experience of using OpenCV so challenged myself to

make it able to plot data. The Viewer class does just that but know that it is not a fully optimized class and may contain bugs that have yet to be discovered (but that's part of the fun of coding, right?).

Please note that the *laserRacoon* code has been designed by me and, as such, it should **NOT** be taken as gospel. I have taken the utmost care to make the classes, functions, and algorithms perform correctly but they have not been rigorously tested. It certainly could be redesigned to be optimized for performance or made more user friendly, I am not precious about it. Use it, abuse it, change it, that is how you learn. The code in the repository will only get updated if major bugs are found.

The code for the Fortran version of this book can also be found on the GitHub repository and may lend additional insight into the topics we discuss in this book. Indeed, I have not (directly) converted the Fortran code written for Chapter 9 on partial differential equations into C++ code; this is left as a challenge for the reader (and partly because we all have time constraints).

The chapters are arranged to provide some logical flow to the exploration of computational physics, starting out with the basic topics such as data fitting and root finding, and building to more advanced techniques, such as performing Fourier transforms and solving partial differential equations. At the end of each chapter are some exercises for the reader to do. These are designed to test you and to get you thinking like a physicist so do not be put off if you find them overly difficult at first. Use the resources available to you to find solutions, which includes fellow students, tutors, and professors, as well as that repository of all knowledge, the Internet—do not forget the library also.

In the Bibliography, you will find a guide to more general reading around each of the topics discussed, including pointers to other introductory texts in computational physics, the C++ programming language, and the Linux operating system. The Appendix contains a more thorough crash course in the C++ language.

CHAPTER 2

GETTING COMFORTABLE

2.1. COMPUTERS: WHAT YOU SHOULD KNOW

Computers are machines that help solve complex or tedious numerical problems. To make the hardware perform such tasks it must be programmed; in other words, told what to do. Remember a computer program cannot think by itself and is only as clever as the programmer who wrote the code. Understanding the underlying structure of a computer can help the programmer write smart code that takes advantage of that structure. For a comparison think about driving a car. You do not need to know how the car works at a component level to drive one. However, should you wish to improve the performance of the car, for racing, or rallying, or off-roading say, then you will have to know about the engine, the suspension, gearing, different types of tires and fuels, streamlining the bodywork, and so forth. This is no different for computers. Anyone can use a computer, but you really need to understand the details in order to get the most out of it.

2.1.1 Hardware

Due to the rapid advancements in computer technology, quantifying statements made in this section may well be out-of-date.

However, the general qualifying remarks should still hold true (unless some paradigm-shifting technology has been invented since).

The physical elements that make up your computer is called hardware and consists of several components. The motherboard is the large, printed circuit board that contains all the ports, plugs, and electronics required to make the required components talk to each other. The central processing unit (CPU) handles most tasks in the computer. The speed at which the CPU handles these tasks is dependent on its clock frequency measured in Hertz. A 2 GHz single-core processor can handle at most 2 billion operations per second; operations may include additions, logic comparisons, and memory calls among others. Before 2004, clock frequencies were roughly doubling every 18 months. This followed the prediction made by Moore in the 1960s that the transistor density on silicon chips would double every 18 months. However, as the power consumed by the CPU goes up as the clock frequency squared, and with global concerns over energy usage, the frequency of the CPU is now capped at or around 4 GHz. The performance of computers has continued to increase according to Moore's prediction using multiple core machines. At the date of writing the current commercially available state-of-the-art is 16 cores, with most "standard" computers having 4 cores, though that is rapidly changing to 8 cores. Multiple cores allow for parallel operation, whereby tasks can be handled simultaneously rather than having to be performed serially. For an introduction to parallel programming see Chapter 12 in this book.

For the CPU to be useful it must have a place to store information. Generally, there are several places for this information to be stored namely cache levels I and II (some CPUs have an additional third level of cache), random access memory (RAM), and storage either on a hard disk drive or in more modern systems on a solid state drive (SSD); from here onwards we will just refer to the storage device as such, or simply storage. This memory system has a hierarchical structure whereby the caches are the fastest but smallest memory levels and the storage is the largest but slowest memory level. Level I cache typically has a size of several tens of kilobytes (if not hundreds of kilobytes in modern CPUs) and can be accessed at the full processor speed. It is split into two separate areas, one for data and one for instructions, both required by the CPU to function.

Level II cache has a typical size of several megabytes and can also be accessed at the full processor speed. Level II cache acts as a fast storage area for program code or variables required by that code. If the level II cache is filled by program code, then the overflow is put into RAM. RAM typically has a size of several gigabytes but is accessed at slower speeds than the caches. If the RAM is filled, then the CPU can store information on the storage device in a place called virtual memory. Storage devices today are immense coming in at relatively conservative 100 GB all the way up to 1 TB and beyond. However, the communication between CPU and virtual memory is limited by the speed at which data can be read from and written to the storage device. This speed of access can be a bottleneck for programs requiring large portions of memory; this is less true for modern PCIe/M.2 SSDs that can achieve around 1.5 GB/s read/write speed. Typically, memory considerations only come into play if it is dealt with high-definition images, video, or 3D graphics. However, some numerical methods can produce matrices of extremely large size that must be dealt with efficiently for a computer to produce timely (and accurate) results. The rise of the graphics card sometimes referred to as a graphical processing unit (GPU) has allowed for the development of some very sophisticated software without the need to use up CPU resources.

Other parts of a computer consist of input and output devices. Input devices are the things with which you communicate with the computer, for example, the keyboard and mouse. Output devices are how the computer communicates with the user, for example, the monitor and printer. Other devices can be considered as "slaves" being both controlled by the computer and relaying data back to the computer on command, for instance, a thermostat used to keep the room temperature constant.

2.1.2 Software

How do you make all that hardware do something? Computers are controlled using programs, referred to as software. The main program that is run on your computer is the operating system or OS. Mostly, Microsoft Windows OS of some version is used in the past; the latest version at the time of writing is Windows 10. Another widely available OS is UNIX which comes in various flavors.

An undergraduate will almost certainly encounter a UNIX OS called Linux in their computer lab.

Programs are written in what is known as high-level languages, for example, Fortran and C++, and are compiled into machine language or binary via another program called a compiler. Note that programs such as MATLAB and Python are interpreted languages that are designed to run effortlessly on multiplatform machines (different OSs). Java is interesting in that it compiles sources into what is called "Byte Code," files identified with the *jar* extension. Byte code can be interpreted by any machine that has the Java Runtime Environment installed.

Before you attempt any programming, please have a look at the following guidelines that may make your life easier:

1. Use your universities' or work's resources, which includes those sat next to you should you be in a computer lab or in your office. Failing an actual person who can communicate at least on some level, use the Internet. If you've got a complicated problem to solve it is very likely someone else has solved it already, and elegantly too (though never believe they managed it in one go without scratching their head at least once, drinking a lot of caffeine-based beverages, and swearing on several occasions). Try not to treat their solution as a black box that takes your inputs and gives the desired outputs without at least trying to understand what the code is doing. There is a practical limit to everyone's knowledge and if it really makes no sense to accept that it works and that, out there, somewhere, is someone much cleverer than you and you're ok with that.

2. Design your program first. Sit down, go to the old school with a pencil and paper, and write down the problem you are going to solve. What do you want to get as the output and what are going to be your inputs? Draw a flow chart if it helps. Write it out as pseudo-code; English phrases that mimic actual code and describe the program's intended function line-by-line. This will, in the long run, save you time. Probably not straight away but practice makes perfect, allegedly. Now once this is done open your favorite text editor/IDE and start tapping away, but be aware (or beware) ...

3. *Make certain to comment on your code*. As previously described, comments not only let others know what is intended with the code but also tell you what to be done. Comments should be clear, concise, descriptive, and written as understandable by the user. Do not worry if you have more comments than code.

4. Make names descriptive. This includes programs, classes, functions, and variables. If the program spanning is large as many hundreds of lines and/or several source files then it would be grateful to have given names that mean something. Additionally, many software companies will have their own naming convention for the various data types, structures, classes, and so forth that can be defined and declared in a program.

5. Do not be afraid to try something out. The worse thing that can happen is that your program crashes at runtime. Control-C starts over. Nowadays, it is very unfortunate to crash the entire computer but just turn it off and on again.

Some suggest that before anything else you should check that the problem you want to solve is suited to the use of a computer to avoid wasting your time and computer resources. While this is a helpful tip for experienced programmers (and who have several higher degrees in mathematics and physics), and it is arguable that only after getting into this habit, it will be comfortable to write a computer code. Sometimes the simple problems allow to explore writing novel and occasionally elegant or clever code that one may have missed trying to tackle a more complex problem.

2.1.3 Number Representation and Precision

As we are scientists, we will be dealing with real numbers obtained from measurements. During A-level physics course (or equivalent) teachers will likely have banged on about significant figures, rounding off, and the difference between precision and accuracy, when taking measurements from experiments. They would have found it bemusing that you quoted every figure on your calculator when figuring out, say, the strength of gravity at the Earth's surface using a free-fall technique. Using a simple stopwatch to determine the time

in free-fall and distanced traveled measured with a ruler the best you could hope to achieve is around three significant figures, limited by human reactions on the stopwatch. This precision could be increased using more precise equipment; for example, a computer control timing circuit measuring the distance using a laser. The point is that the precision of the results depends on the equipment used.

Computers can only express integer values exactly and are limited to a maximum integer value that can be expressed. Numbers in computers are stored as bits in binary format. A bit can have a logical value of 0 or 1, and strings of bits can be used to express integer numbers. A byte generally means a string of 8 bits, and 4 bytes, that is, composed of 32 bits is referred to as one word. Here, the binary format is referred as a big-endian, that is with the most significant bit written first at the left, as one would write decimal numbers. In contrast, little-endian puts the most significant bit at the end on the right, which is a natural format when performing binary addition, and the bits are in arithmetic order.

Take into consideration a byte or 8 bits. Each bit represents a power of two, starting at seven and ending with zero:

$$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$$

For example, the decimal number 6 would be represented by 0000 0110 in binary format; the equation governing this is

$$\sum_{k=0}^{7}(2^k \times s)$$

where k represents the bit location and s represents the bit value. Note that binary is easily read in 4-bit strings that the astute reader may notice leads naturally to the hexadecimal format – honest. The maximum integer can be expressed with 8 bits, that is $2^8 - 1 = 255$. Note that $2^8 = 256$ numbers can be represented, one of which is zero hence the -1. This data type is known as an 8-bit, unsigned integer. Note that color images tend to be saved in this format with 8-bit unsigned integer values defining the three color channels (RGB) leading to the statement that color images have 16 million colors $(256 \times 256 \times 256)$. A single channel image is generally referred to as a grayscale image, zero representing black, 255 representing white.

Thus, there are 256 shades of grey but this makes less of a snappy title for a book.

Negative integers may also be expressed using this binary format. To do this the first bit (most significant bit) is taken as the sign bit. This now leaves us with only 7 bits to represent a number which gives a maximum number of $2^7 - 1 = 127$. However, negative numbers can now be formed by taking the two's compliment of a positive number. To do this you first form the one's compliment by swapping the ones and zeros in the number, then add one to the result. For instance,

$$+6 = 0000\ 0110 \leftrightarrow 1111\ 1001 \leftrightarrow 1111\ 1010 = -6.$$

Zero is still represented by all zeros in the bit locations (take the two's compliment of zero and you should still get zero). Given this conversion what is the largest negative number we can represent in this format? Take the largest positive number we can represent and take its two's compliment:

$$+127 = 0111\ 1111 \leftrightarrow 1000\ 0000 \leftrightarrow 1000\ 0001 = -127$$

However, note that the one's compliment of +127 is available for use and, by definition, it is one less than the two's compliment. Hence the largest negative number we can represent is -128. Note we have not lost any depth of numbers, we can still represent $2^8 = 256$ numbers; 128 negative numbers plus 127 positive numbers plus 1 for the zero.

Larger numbers can be represented using larger bit stings. A 32-bit word length can represent a maximum unsigned integer of $2^{32} - 1 = 4,294,967,295$ or the signed integers in the range $[-2^{31}, +2^{31} - 1]$.

Computational physics would be somewhat limited if computers could only use integer numbers. We need a way of representing floating-point decimal numbers. To do this, we take our 32-bit word length and split it into three blocks. Figure 2.1 illustrates this representation. The first block is one bit long and represents the sign of the number, 0 and 1 representing positive and negative values, respectively. The second block, typically 8 bits long represents the exponent, and the third block, containing the remaining 23 bits is the mantissa.

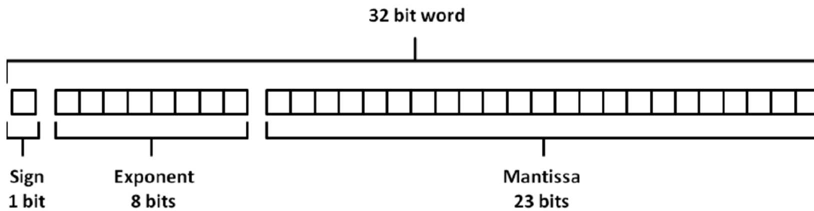

FIGURE 2.1: 32-bit representation of a floating-point number.

The most significant bit in the *mantissa* is on the left and represents 2^{-1}. The next bit represents 2^{-2} and so forth. To calculate the floating-point decimal from the 32-bit representation we use the following equation

$$x_{float} = (-1)^{s} \times mantissa \times 2^{exponent-bias}$$

where s is the value of the sign bit, and the *mantissa* and *exponent* are the decimal values obtained from their respective binary format blocks. The *bias* is an implicit value that is included for the following reason. The 8-bit exponent does not contain an explicit sign bit and so can only represent positive numbers up to the maximum of 255. To circumvent this drastic limitation on floating-point number representation an implicit bias of 127 is included in the floating-point calculation. The range of exponents hence becomes [−127, 128]. The largest positive or negative number that can be represented is then approximately $\pm 1.7 \times 10^{38}$, and the smallest, not considering zero, is approximately $\pm 7 \times 10^{-46}$. However, do not confuse this number as the computer's precision. The computer's precision is governed by the bit length of the mantissa; the exponent just defines the range of representable numbers.

The machine precision is best described in terms of how the computer performs floating-point arithmetic. Say you have the number 5 and wanted to add 10^{-7}. Both numbers can be represented by the computer in floating-point notation, so far so good. To add them together the computer must match their exponents meaning that the bits in the mantissa of the smaller number get shifted to the right. By the time, the bits have been shifted to represent 10^{-7} with the same exponent as 5 they have all gone past the least significant place and have been lost, in essence making 10^{-7} equal to zero. The result of the addition would be 5.

The number 10^{-7} has not just been plucked out of thin air. The least significant bit in the mantissa has a value of $2^{-23} = 1.2 \times 10^{-7}$ (2s.f.). This value represents a kind of number resolution; it is the smallest discernible difference between two numbers on a computer using a 32-bit word length. Note that it is a relative value; if you take the number $2 \times 10^6 = 2{,}000{,}000$ then the next discernible number as far as the computer is concerned is 2000000.1. The machine epsilon or precision is the unit round-off error, essentially half the number resolution. For example, numbers in the range 2000000.000 to 2000000.049 would round down to 2000000.0, whereas numbers in the range 2000000.050 to 2000000.099 would round up to 2000000.1. Any result quoted from the computer should really include this rounding error, for example, $2{,}000{,}000 \pm 0.05$. Because of the machine epsilon you should always consider whether the precision you are using is fit for purpose. If your calculations involve extreme differences between variable values, then unit round off may lead to large errors.

Clearly, the precision can be improved by adding more bits to the mantissa. This can be done by taking bits from the exponent but at the expense of the range of representable numbers. The other way of increasing the bit length of the mantissa is to double the word length from 32 to 64 bits. The standard format of a double data type is an 11-bit exponent and 52-bit mantissa plus the sign bit. What should the machine epsilon be using a double precision data type? To check you can write a few short lines of code to calculate the machine epsilon for both single and double precision variables. The pseudo-code for this task is written as follows:

```
Pseudo code for the calculating the machine epsilon:
Calculate the machine epsilon for both single
(32 bit) and double (64 bit) precision data types.
Divide a value by 2 in a loop and test the condition
that 1 plus the value is greater than 1. Break when
the condition is not satisfied.
Program Epsilon
!!Declare the variables you are going to use
Single    eps_s = 1
Double    eps_d = 1
```

```
!!performs command while the condition is true
While( 1 + eps_s > 1 )
         eps_s = eps_s/2 !!command to execute
end While
While( 1 + eps_d > 1 )
         eps_d = eps_d/2 !!command to execute
end While
!!print results to screen

output( "single machine epsilon = ", eps_s)
output( "double machine epsilon = ", eps_d)
end Epsilon
```

Be wary that some compilers have been written to be smart and will try to "help" when producing the binary (executable) output. For instance, a C++ program written using the pseudo-code above gave a result that the single and double precision were both equal to 5.42×10^{-20}, a precision of 64 bits. This clearly is incorrect. Changing the optimization flags one could managed to recover the expected result of 5.96×10^{-8} for single precision and 1.11×10^{-16} for double precision. The incorrect result is probably due to the compiler "helpfully"" converting the variables to extended precision that has a length of 80 bits, with a 64-bit mantissa. In any case, the initial result was clearly incorrect, and that brings us to an important point. Do not blindly accept what the computer outputs. If the answer looks wrong, then it most probably is wrong. When *that* guy in (A-level) physics class stated boldly that the strength of Earth's gravity is two orders of magnitude larger than it is because that is what the calculator outputted, only to later realized that it would been using centimeters rather than meters.

2.2 SOME IMPORTANT MATHEMATICS

Physics describes the universe from tiniest sub-atomic particle to the shape of the universe itself. The language of physics is mathematics. However, do not confuse the two; physics is not the study of mathematics (and vice versa) but uses mathematics as a tool to

describe and interpret the observation that we make of the universe. Nowhere is this truer than when dealing with computers that are, at the most basic level, efficient number crunching machines.

In this section, we will briefly review some fundamental mathematical concepts that are vital to any computational scientist performing numerical analysis.

2.2.1 Taylor Series

Brook Taylor was an English mathematician born in 1685 who devised an extremely useful way of approximating a function, the Taylor series expansion. This series expansion is arguably one of the most useful in mathematics and certainly within numerical analysis and will play a major role in much of the subject matter contained in this book.

The Taylor series is a mathematical technique for expressing a (potentially) complicated function in the form of a polynomial. The polynomial will have a similar value to the approximated function at least in some small neighborhood of a particular point. More precisely, a Taylor series is an infinite sum of power terms that represent a function at a single point. The summation terms are calculated from the values of the function's derivatives at that point. Mathematically we write

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!}f''(a) + \cdots + \frac{(x-a)^{n-1}}{(n-1)!}f^{(n-1)}(a) + \frac{(x-a)^n}{n!}f^n(\xi) \tag{2.1}$$

where a is some point on x and we have used the notation that

$$f'(x) = \frac{df}{dx}. \tag{2.2}$$

The last term in Equation (2.1) is the remainder or the error in the approximation where $a \le \xi \le x$.

Usually, functions are approximated by using a finite number of terms of its Taylor series. Any finite number of initial terms of the Taylor series of a function is called a Taylor polynomial, the order of the polynomial governed by the highest power left in the

approximation. For instance, a first ordered Taylor polynomial has the form

$$f(x) \approx f(a) + (x-a) f'(a). \tag{2.3}$$

Note the use of the approximately equals to sign as we have not included the remainder term here.

Please recollect the first ordered Taylor polynomial approximation as attempting to match the local neighborhood of the function at the point $x = a$, using the function's slope at that point. The second order polynomial, then, includes the curvature of the function at the point of interest. As more terms are added, higher ordered derivatives become utilized leading to a more accurate approximation of the function around the point of interest. Typically, the approximation is only usefully accurate over a *closed interval* about the point. A function that is *equal* to its Taylor series in an *open interval* is known as an analytic function. For instance, a straight-line function with some non-zero gradient would be given exactly by Equation (2.3) and is thus analytic.

The upper bound to the error in a Taylor polynomial can be estimated by analyzing the next term in the series from where we truncated the approximation. For example, take the Taylor series for the sine function taken about zero and truncated so that it is a seventh ordered polynomial approximation

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}. \tag{2.4}$$

The upper bound to the error is then calculated by the next term in the series thus

$$\varepsilon = \pm \frac{x^9}{9!}. \tag{2.5}$$

This upper bound is ignored further, higher ordered terms, which tend to improve the accuracy of the approximation, that is, to reduce the error.

The sine function and its seventh ordered Taylor polynomial are plotted in Figure 2.2. Here we can see the approximation is only reasonably (to the eye) accurate on the interval $[-\pi, \pi]$.

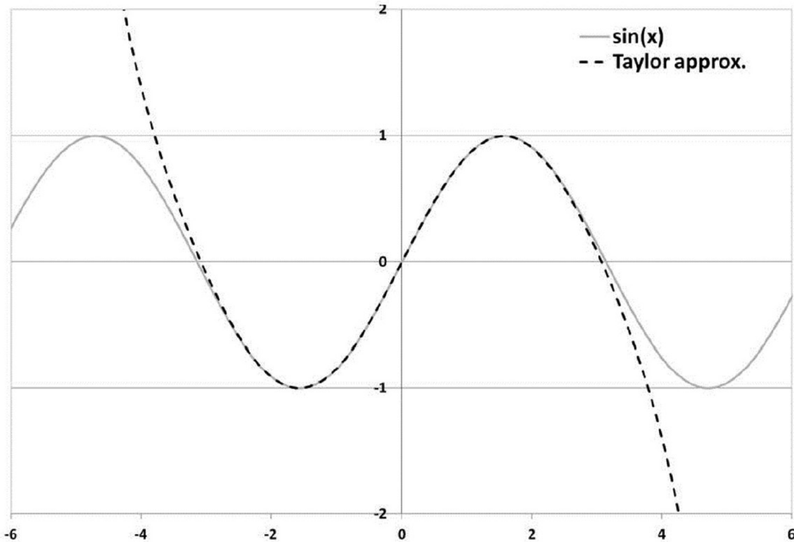

FIGURE 2.2: Seventh ordered Taylor polynomial approximation of the sine function.

2.2.2 Matrices: A Brief Overview

Matrices are incredibly important structures within mathematics, and thus within physics also. A very brief overview of their form and function were described in this section.

A matrix is an array of numbers. The dimensions of a matrix specify the number of rows and the number of columns the matrix has, in that order. Hence, when we say an n-by-m matrix we imply it has n rows and m columns. Vectors are essentially matrices of dimension n-by-1, for instance, a point in three-dimensional space is represented by a 3-by-1 matrix, normally referred to as a position vector. When $n = m$ we have a square matrix; these occur often when solving problems in physics.

When writing an algebra for matrices the notation is conventionally an uppercase letter for the entire matrix, and the corresponding lowercase letter for its elements. The elements also come with numbered subscripts to denote their position within the matrix, row index first. For example, the element found in the first row and the first column of matrix A would be denoted a_{11}, whereas element a_{34} is located at the third row and fourth column of A. In general element

a_{ij} is found in the ith row and the jth column of the matrix A. Note that the numbering starts from one.

Diagonal elements of a matrix are identified by the fact that the row index i will equal the column index j. Sub-diagonal elements are identified by $i > j$, and conversely super-diagonal elements are identified by $i < j$. To illustrate, a general n-by-m matrix can be written as

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1m} \\ a_{21} & a_{22} & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & a_{ii} & \vdots \\ \vdots & & & & \vdots \\ a_{n1} & \cdots & \cdots & \cdots & a_{nm} \end{bmatrix} \tag{2.6}$$

where in this case $n > m$.

Matrix addition is a straightforward extension to addition with real numbers. The corresponding elements are added between the matrices as per user preference; note that the matrices are of the same dimensions and the addition will result in a matrix also of the same dimensions. Thus for 2-by-2 matrices

$$A + B = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix} \tag{2.7}$$

As this is a simple extension to addition with real numbers the properties of addition apply to matrices. In other words, we have the

Commutative property: $A+B=B+A$;

Associative property: $(A+B)+C=A+(B+C)$;

Additive Identity property: $A+ZERO=A$; and the

Distributive property: $C(A+B)=CA+CB$

where ZERO is a matrix of the same dimensions as A but every element is zero; in technical parlance this is the called the *null* matrix. Be aware that we must be somewhat careful with the distributive property so as to maintain the proper order of the multiplication oth-

erwise we run into problems which will be discussed subsequently. Subtraction follows these same rules.

Matrix multiplication is somewhat more complicated than addition. If you have the n-by-m matrix A that is multiplied with the m-by-p matrix B, then the result will be the n-by-p matrix C. The elements in C are then given by the equation

$$c_{ij} = \sum_{i=1}^{n}\sum_{j=1}^{p}\sum_{k=1}^{m} a_{ik} b_{kj} \,. \tag{2.8}$$

Notice that the inner dimensions of the two matrices must match. In other words, the number of columns of matrix A must equal the number of rows of matrix B. The way Equation (2.8) has been written mimics how you will have been taught to do matrix multiplication, moving along the rows of A, and down the columns of B. However, notice that the summation limits are not dependent on each other meaning that their order could be swapped without affecting the result.

As a result of Equation (2.8), matrix multiplication is not commutative, that is $AB \neq BA$. However, it is associative such that $A(BC)=(AB)C$. And, as we have already seen, it is distributive over matrix addition so long as you maintain strict matrix order.

When multiplying a matrix by a scalar that scalar gets *broadcast* across the entire matrix i.e., every element gets multiplied by the scalar. As the scalar is just a number then

$$\rho\,(AB) = (\rho\,\mathrm{A})B = A(\rho\,B) = (AB)\rho \tag{2.9}$$

where ρ is any scalar.

The trace of a square matrix is the sum of the main diagonal elements of that matrix. In equation form we write

$$tr(\mathrm{A}) = \sum_{i=1}^{n} a_{ii} \tag{2.10}$$

where A is a n-by-n matrix. The trace of a (square) matrix has some interesting properties not least the fact it is equivalent to the sum of the eigenvalues of the matrix A.

Eigenvalues of a matrix are related to its eigenvectors such that

$$A\underline{e} = \lambda\,\underline{e} \tag{2.11}$$

where $\underline{e}$ is an eigenvector of A and λ is its corresponding eigenvalue. Note that λ is just a number. You have probably already performed calculations using the characteristic polynomial to determine the eigenvectors and related eigenvalues of some relatively simple matrices. As a reminder the characteristic polynomial is calculated as

$$\det(A-\lambda I)=0, \tag{2.12}$$

which produces an n ordered polynomial in terms of the eigenvalue λ. The "det" means calculate the determinant of the matrix contained within the brackets, which is relatively easy to do for 2-by-2, and 3-by-3 matrices but not for larger dimensions of matrix. The matrix I is the identity matrix that has ones on its diagonal elements and zeros in the other elements.

As an aside, eigenvectors and eigenvalues are important concepts with the realm of quantum physics. For instance, the time independent Schrödinger Equation can be written in the form

$$H\psi = E\psi \tag{2.13}$$

where the matrix H represents the Hamiltonian of the system; the differential operators governing the potential and kinetic energies, ψ (pronounced psi) is the wavefunction of a quantum particle, for example, an electron, and E is the (total) energy value of that wavefunction. In other words, ψ is the eigenvector of H, and E is the corresponding eigenvalue. If this at present makes little sense to you do not worry, just be aware that eigenvectors and eigenvalues are particularly important concepts in mathematics and physics.

Matrices can be transposed which means that row i is swapped with column i of the matrix. I tend to think of this as putting a double-sided mirror along the diagonal of the matrix and the transpose is that which can be seen in the reflection. For instance, if we transposed the m-by-p matrix B the result would be the p-by-m matrix B^T, where the superscript T denotes the transposition. Notice that the row and column dimensions have swapped. In terms of matrix multiplication, we can write

$$(AB)^T = B^T A^T. \tag{2.14}$$

A symmetric, square matrix is one that is equivalent to its own transpose. A positive definite matrix is a special type of symmetric matrix with all positive eigenvalues. Determining if a matrix is positive definite can be difficult but they are mentioned here as they tend to crop up quite often in the solutions to physics problems.

If the matrices elements are complex numbers (they contain both real and imaginary terms) then we can take what is known as the Hermitian conjugate; take the complex conjugate of the elements, then transpose the matrix. In mathematical notation we can write the equation

$$(AB)^{\dagger} = B^{\dagger}A^{\dagger} \tag{2.15}$$

where the dagger symbol (†) denotes the Hermitian conjugation.

For square matrices there is the multiplicative identity property such that

$$AI = IA = A \tag{2.16}$$

where I is the identity matrix. When the matrix multiplication of two matrices, say X and Y, results in the identity matrix then we can say that Y must be the inverse matrix of X (or vice versa) by definition.

To compute the inverse of a matrix directly you find its matrix of cofactors and divide through by its determinant. For 2-by-2 and 3-by-3 matrices this can be done with relative ease but as the order of the matrix increases the computational effort required grows exponentially both in calculating the matrix of cofactors and finding the determinant. There are other methods for "inverting" a (square) matrix such as elimination and decomposition techniques that are much more computationally friendly. The Fortran library LAPACK has a plethora of subroutines that employ such techniques. Generally, we are solving the linear set of equations

$$A\underline{x} = \underline{b} \tag{2.17}$$

where $\underline{x}$ is the vector we wish to find, $\underline{b}$ is the vector of known values, and the matrix A represents some relevant coefficients of the system for which we are trying to solve.

Note that if the determinant of a matrix is zero then we say the matrix is singular and non-invertible. Where say the matrix represents the coefficients of a linear system of equations this would be interpreted as the system as either having no solutions or many solutions. When the determinant is non-zero the system of equations will have exactly one unique solution.

The preceding discussions provide an (extremely) brief exposition of matrices and their properties and this section will provide a quick recalling on fundamental concepts of matrices. It is recommended to refer to a book dedicated to matrices and linear algebra for further understanding.

EXERCISES

2.1. How is the number +5 represented by a 32-bit floating-point notation? Use as the equation given as a guide.

2.2. What happens if you change the conditional statements in the while loops to eps_s > 0 and eps_d > 0 in the machine epsilon program? Why?

2.3. Add to the machine epsilon code to calculate the machine precision in terms of the mantissa bit length (Hint: how many times has it divided by two?)

2.4. Try to write pseudo-code to calculate the machine epsilon using a recursive function (a function that calls itself). Think about how to terminate the recursion.

2.5. Investigate the upper bound of error for the Taylor series approximation for the sine function. Is it well estimated by the next term in the series from where we truncated the series?

2.6. π is one of those fundamental numbers that just keeps cropping up. One way to estimate π is to analyse the perimeter of polygons inscribing a circle. For a circle of unit diameter, we may formally write the expansion

$$\pi_k = \pi_\infty + \frac{c_1}{k} + \frac{c_2}{k^2} + \frac{c_3}{k^3} + \cdots$$

where k is the number of sides of the polygon, π_k is the approximation, π_∞ is the actual value of pi to be determined, and the c_i are coefficients also to be determined. Given that $\pi_8 = 3.061467$, $\pi_{16} = 3.121445$, $\pi_{32} = 3.136548$, and $\pi_{64} = 3.140331$ compute a value for π_∞. (Tip: think about this as a set of simultaneous equations in matrix form and look-up Gaussian Elimination.)

CHAPTER 3

INTERPOLATION AND DATA FITTING

3.1 INTERPOLATION

3.1.1 Linear Interpolation

The principles behind interpolation and extrapolation are something every scientist should understand. Most measurements of a system, whether that is a physical experiment or theoretical calculation, will consist of pairs of discrete values; an independent variable x, which will vary, and a dependent variable y, which is measure. To extract information from these pairs of values one would, ideally, find an analytical function that would give y for any arbitrary x. Often an analytical solution does not exist or is too tedious or complicated to solve. In this case, how to find a value for y that sits between measured values in x? We can either try to fit the data to some function (typically a polynomial) or interpolate the data. The data should be *extrapolated* to find a y beyond measured range in x. The difference between the two methods is that interpolation is constrained so that the function used to approximate the data must pass through the measured data points, whereas data fitting only requires that some error function is minimized.

As the data points can be approximated by any number of functions, we must have some guidelines that outline a reasonable approximation. As a rule, these guidelines usually rely on the consistency of the gradients or derivatives of the approximation and as a result may not be suitable for functions that have rapid variations, such as those with oscillatory behavior. Sometimes, an important detail about the behavior of a function may be missed should the measurements be too sparsely spread. As a crude example of this, think about measuring the displacement of a mass on a spring as a function of time. If the sample frequency (how often you take a measurement) matches the period of oscillation then the interpolated result would show that the mass does not move at all, which is clearly an error.

Linear interpolation is probably the most intuitive method, and probably one which is used to quite regularly without realizing. Essentially, a straight line is assumed to approximate the function between two neighboring data points, with the line passing through both points. Indeed, this is a fundamental concept of mathematics to find the derivative of a function; on an infinitesimally small interval, any function is a straight line. Obviously, on a practical level, one cannot make measurements that are infinitesimally distinct, the best that can possibly achieve is the precision of the measurement device.

The following forms of writing the equation of a straight line is the most familiar one

$$y = mx + c \tag{3.1}$$

where m is the gradient and c is the intercept with the y axis, or

$$\frac{y - y_1}{x - x_1} = \frac{y_2 - y_1}{x_2 - x_1} \tag{3.2}$$

where the line passes through the points $(x_1, y_1), (x_2, y_2)$.

In the world of academia, these equations typically take the form

$$g(x) = a_0 + a_1 x \tag{3.3}$$

where a_0 and a_1 are called the coefficients of the linear functions; they still have the same meaning as c and m respectively in the other equations. The reason for writing the coefficients as a single letter

with a subscript is that it is both elegant and descriptive; the letter immediately represents that it is a coefficient rather than a variable, and the subscript describes to which power of x the coefficient belongs. Another good reason for the subscripts is that they lend themselves quite naturally to being stored as a vector or an array in computer memory, but more on this later.

With any interpolation, we are approximating the unknown function f(x) with a function g(x) with the constraint that they are equal at the measured data points which we label x_j. Thus, for neighboring data points using linear interpolation:

$$g(x_j) = f(x_j) = f_j = a_0 + a_1 x_j \tag{3.4}$$

$$g(x_{j+1}) = f(x_{j+1}) = f_{j+1} = a_0 + a_1 x_{j+1} \tag{3.5}$$

Note that they share the same coefficients as the straight line approximation is constrained to pass through both points. Solving for the coefficients, that is, finding a in terms of f and x, the function g(x) takes the form

$$g(x) = f_j + \frac{x - x_j}{x_{j+1} - x_j}\left(f_{j+1} - f_j\right) \tag{3.6}$$

valid for the range $[x_j, x_{j+1}]$. Take a moment to verify that this equation is a straight line and equivalent to those you are familiar with. Equation 3.6 can be written in what is called symmetrical form as follows

$$g(x) = f_j \frac{x - x_{j+1}}{x_j - x_{j+1}} + f_{j+1} \frac{x - x_j}{x_{j+1} - x_j} \tag{3.7}$$

If you are wondering why it has been rewritten in this form, it's use will become apparent in the next section discussing polynomial interpolation and Lagrange's interpolation scheme.

The code *linearInterp.cpp* implements the linear interpolation scheme on the function $f(x) = sinc(x)$, using the symmetrical form of the equation. The application source code takes advantage of the class `phys::interp::Linear` defined in the header file

Interpolation.h and implemented in the source *Interpolation.cpp*. The output from this code is plotted in Figure 3.1. A 10 equidistant points were selected to represent the "measured" data on the interval [0.05, 5.0]. The linear interpolation is applied to each interval pair. From the figure, the linear interpolation does a reasonable job at approximating the function when the second- and higher-order derivatives are small. However, as the derivatives increase in size it becomes much less accurate. This is to be expected; the linear interpolation approximation contains no higher-order terms above one and thus cannot be expected to deal with rapidly changing functions that have sizable higher-order derivatives. The interpolation may be improved by taking more data points over the total range, which in essence applies the mathematical notion of the function approaching a straight line as the interval approaches zero.

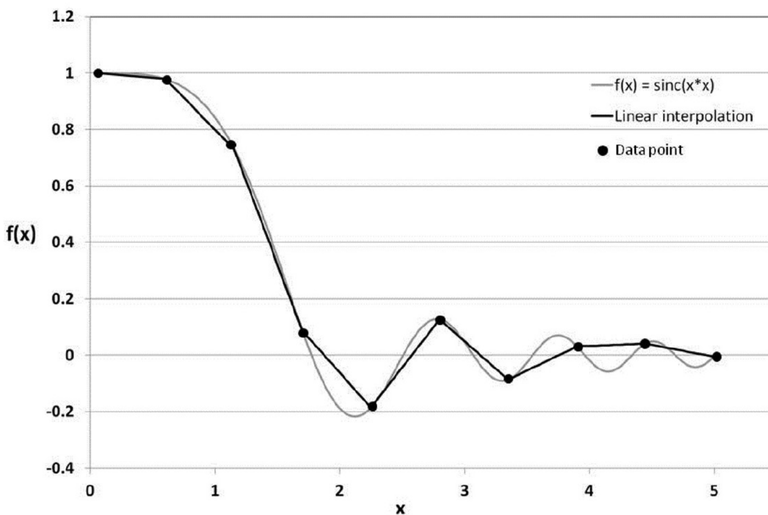

FIGURE 3.1: Linear interpolation of the ***sinc*** function using 10 equidistant "measurements".

3.1.2 Polynomial Interpolation

Equation (3.3) is called a first-order polynomial. By adding higher powers of x, one can modify this to higher-order polynomials. For instance, if the highest power of x were two then it would

be a second-order polynomial (also called a quadratic) and so on. Higher-order polynomials will be better at approximating rapidly changing functions but there is a practical limit to this, which will be discussed in the subsequent sections.

First, we can extend Equation (3.3) so that it forms an n ordered polynomial

$$g(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{3.8}$$

Using our interpolation constraint that the approximation must pass through the measured values gives

$$f(x_j) = f_j = g(x_j) = a_0 + a_1 x_j + a_2 x_j^2 + \cdots + a_n x_j^n \tag{3.9}$$

This is a system of n+1 linear equations (you may know them as simultaneous equations) that we would use to solve for the coefficients. Notice that to perform an n ordered interpolation you need n+1 data points. For instance, the first-order (linear) interpolation requires two points; a second-order interpolation requires three points, and so forth. How a linear system of equations can be solved explicitly using a LAPACK routine is discussed later in this chapter. For the moment, one could formulate the coefficients using an alternate method.

Consider a second-order interpolation for three given points (x_j, f_j) at j, j+1, and j+2:

$$\begin{aligned} f_j &= a_0 + a_1 x_j + a_2 x_j^2 \\ f_{j+1} &= a_0 + a_1 x_{j+1} + a_2 x_{j+1}^2 \\ f_{j+2} &= a_0 + a_1 x_{j+2} + a_2 x_{j+2}^2 \end{aligned} \tag{3.10}$$

The coefficients a_0, a_1, and a_2 can be found from these equations using the methods you should have learned in an A-level mathematics course at least. Give it a go. Remember your finding the coefficients in terms of f and x. Once the coefficients are found they can be substituted into equations (3.10) and rewritten into the symmetrical form giving

$$g(x)=f_j\frac{(x-x_{j+1})(x-x_{j+2})}{(x_j-x_{j+1})(x_j-x_{j+2})}+f_{j+1}\frac{(x-x_j)(x-x_{j+2})}{(x_{j+1}-x_j)(x_{j+1}-x_{j+2})}$$
$$+\quad f_{j+2}\frac{(x-x_j)(x-x_{j+1})}{(x_{j+2}-x_j)(x_{j+2}-x_{j+1})} \tag{3.11}$$

If you find rearranging equations fun, then feel free to have a go at obtaining this form for yourselves but do try to get out more. If you compare Equation (3.7) with Equation (3.11) you will hopefully see that we can generalize the symmetrical form to an n ordered polynomial interpolation scheme:

$$g(x)=f_1\frac{(x-x_2)(x-x_3)\cdots(x-x_{n+1})}{(x_1-x_2)(x_1-x_3)\cdots(x_1-x_{n+1})}$$
$$+f_2\frac{(x-x_1)(x-x_3)\cdots(x-x_{n+1})}{(x_2-x_1)(x_2-x_3)\cdots(x_2-x_{n+1})}+\cdots$$
$$+f_{n+1}\frac{(x-x_1)(x-x_2)\cdots(x-x_n)}{(x_{n+1}-x_1)(x_{n+1}-x_2)\cdots(x_{n+1}-x_n)} \tag{3.12}$$

This is the infamous Lagrange formula for polynomial interpolation. This form is somewhat cluttered and can be written more elegantly as

$$P(x)=\sum_{k=1}^{n}\lambda_k(x)f(x_k), \tag{3.13}$$

where

$$\lambda_k(x)=\frac{\prod_{l=1\neq k}^{n}(x-x_l)}{\prod_{l=1\neq k}^{n}(x_k-x_l)} \tag{3.14}$$

The Π symbol is a capital pi and is the mathematical symbol meaning product of a sequence.

The member function `Lagrange::interpolate(double)`, found in the source file *Interpolation.cpp*, implements the Lagrange interpolation formula as expressed in Equations 3.13 and 3.14. Read through this code to convince yourself of this statement.

An exercise for the reader to write a program to apply the Lagrange interpolation to the function $f(x) = sinc(x)$ using approximating polynomials of increasing order (use *linearInterp.cpp* as a guide). Take note that the order of the approximating polynomial is governed by the number of data points passed to the object of type `Lagrange`, and an n ordered polynomial interpolation requires n+1 data points.

Figure 3.2 shows the plotted output from a program written using the Lagrange class for the first, second, fourth, sixth, eighth, and ninth ordered polynomial interpolations for the function $f(x) = sinc(x)$ over the interval (0.0, 5.o] using 10 equidistant "measured" points. For the lower ordered polynomials, a sliding window approach had to be used to cover the interval as far as possible. For n=1 the data matches that computed from the `Linear` interpolator class.

From inspection of these plots, we can see that the second-ordered polynomial interpolation is an improvement over the first but still has a bad time coping as the function oscillates more rapidly. The fourth- and sixth-ordered interpolations are again an improvement over the second, however, there are two things to note. First, there is a small artifact within the first interval that does not follow the function at all well. Second both interpolations have only covered a fraction of the data points; in fact, only the first and ninth order interpolations have covered the total number of data points. The astute among you will have realized this is since an n ordered polynomial interpolation requires n+1 points. If the value of n+1 is not a factor of the total number of data points, then the scheme will not be able to interpolate those points. Note the wild oscillations in the eighth and ninth ordered interpolation at the beginning and end of the interval. This is a tendency of higher-order polynomial interpolation to introduce more vigorous oscillations than perhaps the data points suggest. This is the practical limit of polynomial interpolation which is referred to earlier. As a rule of thumb try not to use higher than order five polynomials to do interpolation. If greater accuracy is required, you could always take more measurements or apply an alternative interpolation method, for example, spline interpolation.

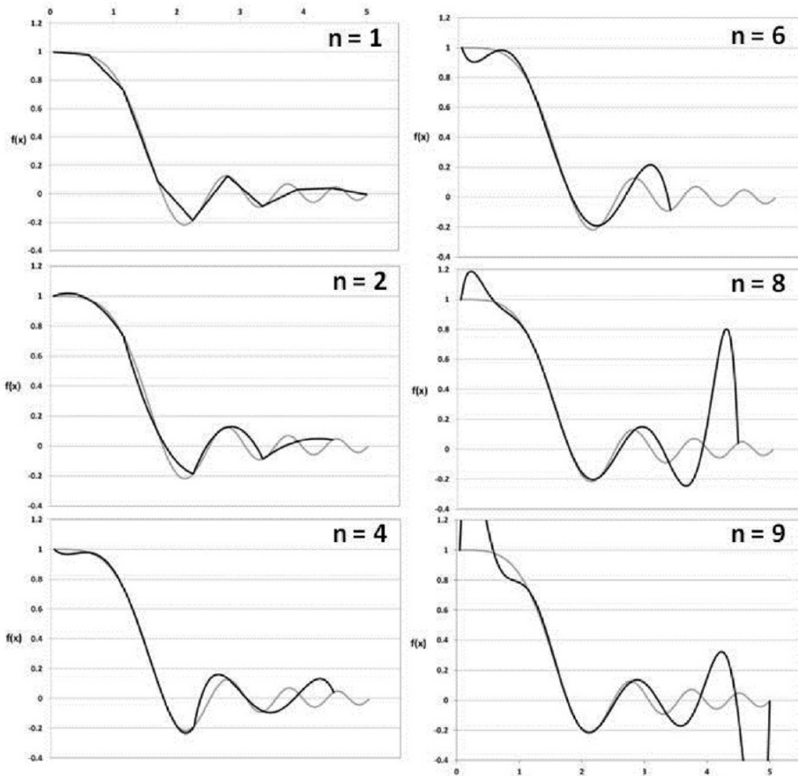

FIGURE 3.2: Polynomial interpolation of the function $f(x) = sinc(x)$ with polynomial orders of 1, 2, 4,6,8, and 9.

To note, the simulated measurements are mentioned here by taking values from the function at equidistant points. In real measurements, the data points will likely not lie on the function that describes them due to the precision of the measuring equipment; measurements are usually plotted with their error bar. Depending on the relative size of the error this may have a significant effect on the interpolation. Additionally, one do not have to take measurements that are equidistant and it is obvious that physics teacher would have told to take more closely spaced data points about the region where the measured variable (y) changes rapidly with the independent variable (x).

3.1.2 Cubic Spline

One of the limiting factors of polynomial interpolation is due to the discontinuities in the derivatives at the data points (see the order 2 polynomial interpolation in Figure 3.2 for a clear illustration of this issue). To overcome this issue, we can use spline interpolation. The term spline has its origins in the shipbuilding industry whereby thin sheets of wood threaded through discrete points (or knots) would form smooth curved shapes due to the minimization of strain within the wood. In essence, the spline approximation not only matches the function at the measured data points but also matches the derivatives of the function at the data points.

The cubic spline is the most popular version of spline interpolation due to its (relatively) simple form and construction, and that it generally gives reasonably accurate results. The cubic part of the name comes from the order of the polynomial used to approximate the function. Cubic splines are said to have an order of four, which means that not only are the polynomial values matched at the data points but so are their first- and second-order derivatives. Given this definition the linear interpolant explored earlier in this chapter is an order two spline. What would an order one spline look like?

Cubic splines tend not to have any inherent advantage over polynomial interpolation for smooth functions or for dense sampling along the x-axis. However, they are particularly good at interpolating sparse data points for smooth functions or when the data points vary rapidly over a region of interest, for instance, in a typical spectral measurement that contains several peaks and troughs. For a decent exposition of how to set up a spline approximation, it is recommended reading Section 2.4 of T. Pang's book listed in the Bibliography. It gets quite heavy on the mathematics of setting up the spline which includes generating matrices and factorizing them using the lower-upper (LU) decomposition method. As we are physicists, we like to use the fruits of the mathematicians' labors, and rather than writing our own spline approximation let us take a shortcut.

Your Unix/Linux distribution may come with *octave* installed, if not it would be able to install it via install manager. Once installed, open a terminal and type "octave" at the prompt. All being well this

will open the Octave program in your terminal. You will know that it is working as you will see the octave prompt. Type the following at the prompt:

```
xf = [0:0.05:5];
yf = sinc(xf);
xp = [0:0.5:5];
yp = sinc(xp);
lin = interp1(xp, yp, xf);
spl = interp1(xp, yp, xf, "spline");
cub = interp1(xp, yp, xf, "cubic");
near = interp1(xp, yp, xf, "nearest");
plot(xf, yf, "r", xf, lin, "g", xf, spl, "b", xf,
cub, ...
"c", xf, near, "m", xp, yp, "r*");
legend("function", "linear", "spline", "cubic",
"nearest");
```

You should now have a neat plot of the function $y = sinc(x)$ with the four different types of interpolation of that function shown; the "measured" points are the red asterisks. Describing this a line at a time we have set up a line space for x on the interval [0,5] using 101 points, then calculated the function $y = sinc(x)$ for that line space. The next line sets up a line space of 11 points on the same interval that represents our "measured" data points, and the corresponding y is then calculated. Then we use the Octave function *interp1* to interpolate our "measured" data points using linear, cubic spline, cubic polynomial, and nearest neighbor interpolation methods. The *1* in the function name refers to the fact we are interpolating in one dimension. We then plot the results on the same figure with the legend as labeled. Note that the ellipsis, …, is a continuation symbol for Octave. Notice also that the "cubic" interpolation uses what is called a piecewise cubic Hermite interpolating polynomial, which preserves the shape of the function.

So, all that coding in C++ to implement the interpolation classes, writing the application logic to use them, followed by importing the resulting data file into an external program for plotting has been handled in ten relatively simple lines of Octave code.

This brings us back to the point made in Chapter 1 that the problems you will encounter are likely to have already been solved and reduced to an elegant form. However, the idea here is not to blindly use the programs written by someone else but at least have a basic understanding of how they function. In the future, you may find yourself with a problem that has yet to be tackled. In attempting a solution, you will require the skill of implementing mathematical equations in computer code, and be able to comment on the accuracy, precision, and limits of what you have written. You can only do this if you have a solid understanding of the underlying theories and equations that govern the problem and your attempted solution.

Other interpolation schemes that you may wish to investigate but are beyond the scope of this book include Rational function interpolation; B-splines; T-splines; Newton Interpolation; Neville's algorithm; and the Aitken Method. This list is not exhaustive.

3.2 DATA FITTING

3.2.1 Regression: Illustrative Example

Regression is a form of data fitting that allows us to mathematically determine the line (or curve) of best fit to measured data. It is like interpolation in that we use measured data points to mathematically approximate a solution. However, it differs from interpolation in that instead of finding a local approximation, that is, a function value located between two data points, we are finding the global behavior or trend of the measurements. In that respect, regression is *not* constrained to pass through the data points. In technical parlance, regression attempts to solve an overdetermined (more equations than unknowns) set of simultaneous linear equations that likely have no exact solution but will have a best-fit polynomial approximation. Regression methods find the coefficients of that best-fit polynomial. One of the most well-used regression schemes is called linear least squares where the best fit is that polynomial which minimizes the sum of the squared differences between the data points and the modeled solution.

To illustrate, let us have a look at a simple example. As a result of an experiment, four data points were obtained as follows: (1,2), (2,1), (3,3), (4,6) each describing an (x, y) coordinate. The experimenters want to find a line that provides the best global trend in these four data points. They initially assume that the relationship between x and y is linear and can therefore be approximated by

$$y = a_0 + a_1 x \tag{3.15}$$

Mathematically speaking, they would like to find the numbers a_0 and a_1 that approximately solve the overdetermined linear system four equations in two unknowns in some "best" sense:

$$\begin{aligned} a_0 + 1a_1 &= 2 \\ a_0 + 2a_1 &= 1 \\ a_0 + 3a_1 &= 3 \\ a_0 + 4a_1 &= 6 \end{aligned} \tag{3.16}$$

The least-squares approach to solving this problem is to try to minimize the sum of the squares of the differences between the right-hand and left-hand sides of these equations. Putting this into algebra we are attempting to make the following function as small as possible:

$$S(a_0, a_1) = (2 - a_0 - a_1)^2 + (1 - a_0 - 2a_1)^2 + (3 - a_0 - 3a_1)^2 + (6 - a_0 - 4a_1)^2 \tag{3.17}$$

From A-level mathematics, one should remember how to find the minimum (or maximum) of a function with one independent variable; you find where the first derivative of that function is zero. For functions with multiple independent variables, the method is no different only that we determine the partial derivative with respect to the independent variables separately. Apart from the variable, we are taking the derivative with respect to, all other independent variables are considered constant. Applying this to the function $S(a_0, a_1)$ and after some rearrangement we obtain

$$\begin{aligned} \frac{\partial S}{\partial a_0} &= 8a_0 + 20a_1 - 24 = 0 \\ \frac{\partial S}{\partial a_1} &= 20a_0 + 60a_1 - 74 = 0 \end{aligned} \tag{3.18}$$

We know that these must give minimums and not maximums because $S(a_0, a_1) \propto a_0^2 + a_1^2$, which has no maximums. Equations (3.18) are called the normal equations and when solved give $a_0 = -0.5$ and $a_1 = 1.4$.

The line that these coefficients describe is plotted in Figure 3.3(a) along with the data points, and it is the line of best fit for a linear model. However, what if the experimenter's initial assumption about the relationship being linear is wrong? Perhaps we ought to add more terms to the approximating polynomial. Adding an extra term to our approximating polynomial gives

$$y = a_0 + a_1 x + a_2 x^2 \qquad (3.19)$$

that when processed via the method above gives us values for the coefficients of best fit of $a_0 = 4.5$, $a_1 = -3.6$, and $a_2 = 1.0$.

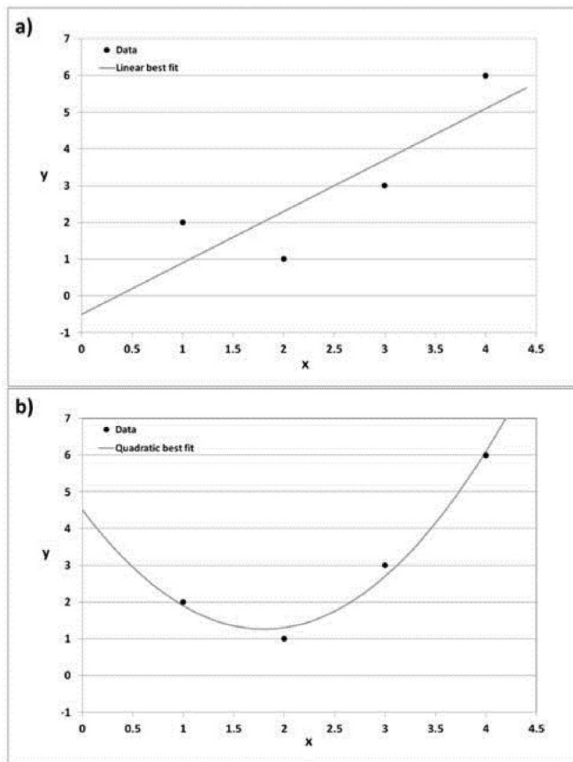

FIGURE 3.3: Linear least squares fit of the data using a linear model (a), and a quadratic model(b).

This curve is plotted with the data points in Figure 3.3(b). So, which is the curve of best fit and thus tells the experimenters the global behavior? Naively you may think the quadratic curve is better, it being much closer to the data points than the linear behavior. However, upon inspection of our approximation, we see that we are producing the Taylor series expansion for the function that passes through those specific data points. Adding more terms to the polynomial is bound to improve the accuracy of the curve passing through the points as we are providing a better approximation to the higher-ordered derivatives of the function. Clearly, the difficulty in interpreting the global behavior of this simple, made-up data set is due to the small number of measurements considered. Could the first data point measured at x=1 be an anomaly or an actual feature? The only way to tell in a real experiment would be to take more measurements.

3.2.2 Linear Least Squares: Matrix Form

When it comes to solving a large system of linear equations it is most convenient to write them in matrix form. The general matrix formula for a system of linear equations is

$$A\underline{x} = \underline{b} \tag{3.20}$$

where A is a matrix of known coefficients (not to be confused with the coefficients of the approximating polynomial), $\underline{x}$ is the vector of unknown variables, and $\underline{b}$ is the vector of known right-hand side values. To illustrate this matrix form for the normal equations of the linear least-squares method consider Equations (3.18). Written in matrix form they give (notice there is a common factor of two)

$$\begin{pmatrix} 8 & 20 \\ 20 & 60 \end{pmatrix}\begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} 24 \\ 74 \end{pmatrix}.$$

We can generalize this matrix form for linear least squares to give

$$A = \begin{pmatrix} n & \sum_{i=1}^{n} x_i & \cdots & \sum_{i=1}^{n} x^m \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 & \cdots & \sum_{i=1}^{n} x^{m+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{n} x_i^m & \sum_{i=1}^{n} x_i^{m+1} & \cdots & \sum_{i=1}^{n} x_i^{2m} \end{pmatrix}, \qquad (3.21)$$

where n is the total number of data points, and m is the order of the approximating polynomial. Here x refers to the independent variable in the measurement set, not the unknown vector x in Equation 3.20, that holds the unknown coefficients (sometimes there just aren't enough characters in the alphabet). We note that matrix A in this case has the nice property that it is symmetrical.

To solve Equation (3.20), that is, find the unknown vector $\underline{x}$ (the coefficients of the approximating polynomial), we must factorize matrix A. If you have solved a set of simultaneous equations before then you have factorized a matrix without realizing it, probably. The Gaussian elimination (GE) method is used to find a multiplier between two equations to remove a variable from their resulting addition. In matrix format, this is equivalent to finding a multiplier between two rows, with the resultant addition zeroing a matrix element. With row and column exchanges the resultant matrix can be made into either a lower or upper triangular matrix and we can solve the entire system from the row containing the single non-zero element on the diagonal. Note that GE is not the only factorization method; others of note include Cholesky, LU decomposition, and QR decomposition.

Cholesky factorization is the method of choice here to solve Equation 3.20 for least linear squares. Andre-Louis Cholesky was a French mathematician who developed his eponymous factorization method (sometimes referred to as decomposition) when solving a geographical survey problem in his home country in the first decade

of the 20th century. The factorization takes a symmetric positive definite matrix A and writes it as

$$A = LL^T$$

where L is a lower triangular matrix with positive diagonal entries. To solve Equation (3.20) you say that

$$\underline{y} = L^T \underline{x}$$

$$L\underline{y} = \underline{b}$$

The latter of these two equations is solvable for y, which in turn means you can solve the former of these equations for x. Cholesky was also a French military officer who served during the First World War and died from battlefield wounds in August of 1918 at the age of 42. His work was published in 1924 posthumously by a fellow officer but received little attention until the latter half of the 20th century.

I have written some of these matrix factorization methods as classes in the C++ library found at the GitHub site. They are defined in *LinearSolvers.h* and implemented in *LinearSolvers.cpp*. I have also written a program to find the least linear squares fit for an order 1 polynomial of the example data using Cholesky factorization, source code in *leastLinearSquares.cpp*. As an aside, if you plan on performing matrix factorization in any serious fashion you should look up C/C++ wrapper libraries for LAPACK and BLAS, or for a totally C++ approach search for the Eigen project.

3.2.3 Realistic Example: Millikan's Experiment

The oil drop experiment, or more famously Millikan's Experiment, was an experiment performed by Robert Millikan and Harvey Fletcher in 1909 that provided one of the first accurate measures of the elementary electric charge (the charge of the electron).

TABLE 1: Some of Millikan's and Fletcher's oil drop data

n	$q_n / 10^{-19} C$	n	$q_n / 10^{-19} C$
4	6.558	12	19.68
5	8.206	13	21.32

n	$q_n / 10^{-19}C$	n	$q_n / 10^{-19}C$
6	9.880	14	22.96
7	11.50	15	24.60
8	13.14	16	26.24
9	14.82	17	27.88
10	16.40	18	29.52
11	18.04		

The experiment involves balancing the gravitational force with the drag and electric forces acting on microscopic, charged droplets of oil suspended between two metal electrodes. The droplet's radii can be measured, and with knowledge of the oil's density, their weight and buoyancy can be calculated. Millikan and Fletcher could use this information with a known electric field to determine the charge on oil droplets in mechanical equilibrium. By repeating the experiment for many droplets, they confirmed that the charges were all multiples of some fundamental value and calculated it to be about 1.5924×10^{-19} C ±0.01%. They proposed that this was the charge of a single electron.

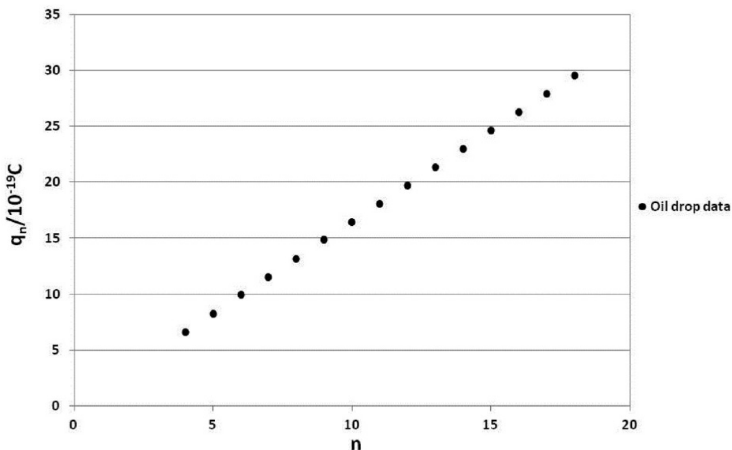

FIGURE 3.4: Plot of Millikan's oil drop data.

Some of the data from Millikan and Fletcher's experiment are shown in Table 1 and plotted in Figure 3.2. At first glance, the data seem to lie perfectly on a straight line that passes through the origin.

Can we show that mathematically that this is the case? The answer is yes otherwise this section would be truly short!

A straight line through these data has the form

$$q_n = ne + \Delta q$$

where the fundamental charge e is the gradient of the line, and n is an integer. We can determine both e and the (estimated) error in the charge Δq from this data by adapting the linear least-squares program to use the data found in the file *millikanData.txt* located in the *resource* sub-directory of the *progs* directory. The resulting output determines $e \approx 1.64 \times 10^{-19}$ C with an estimate for the error bounds as $\Delta q \approx \pm 0.03 \times 10^{-19}$ C. This is in remarkably close agreement with the currently accepted value of $e = 1.602 \times 10^{-19}$ C (3sf). To mathematically test how well the data are fitted by a straight line we can calculate what is called the residual norm. This is the square root of the sum of the squared residuals and it should be a vanishingly small number for the oil drop data presented. The other method we can employ is to increase the order of the approximating polynomial to study the relative sizes of the coefficients. For both the methods exercises are provided at the end of chapter for practice.

As a cautionary note, the arguments above for least-squares fitting assume that there is no error in the measurements of the independent variable x and this assumption is valid in general. In fact, for the oil drop experiment the x values are necessarily integers, being multiples of the fundamental charge, and implicitly have no error. However, in some cases, the error in x will be comparable to the error in the measured value y. In this case, you would have to apply a total least squares approach that somehow minimizes residuals in both the x and y coordinates.

Of course, sometimes you may be faced with data that is nonlinear, for example, data from spectral measurements or resonant phenomenon, where you will be interested in the location of a peak, and probably its width and its height. Non-linear equations are more complicated to deal with but can still be fitted in the least squares sense. The mathematics to deal with non-linear equations are beyond the scope of this book but for a decent introduction to

non-linear least-squares approximations see Chapter 3 of Paul L. DeVries' book *A First Course in Computational Physics*.

EXERCISES

3.1. Write a program that uses the Lagrange interpolation class to interpolate the function sinc(x) using polynomials of increasing order.

3.2. Using a mathematics library, for example, octave, see how well a cubic spline interpolation performs over polynomial interpolations for the same functions using the same data points. Choice of function and number of data points is completely free. Go nuts. Test out any other interpolation routines you may find.

3.3. The code provided for the Linear Least Squares approximation is less than optimal.

- **a.** Add user-defined input so that the order of the fitting polynomial can be chosen by the user. Think about the validation of that input.
- **b.** Modify the code so that the matrix form of the normal equations (A) can be initialized from a general set of data read in from a file. (Hint: How do the matrix indices i and j relate to the power of the independent variable x?).
- **c.** Automate the calculation of the right-hand-side vector b in Equation 3.20 from a general set of data. Use Equations 3.16-18 as a guide, and there is a hint in the source file.
- **d.** Include a calculation for the residual norm

3.4. Test out your modified Linear Least Squares approximation program on the Millikan data for polynomial orders greater than one. Comment on the results.

CHAPTER 4

SEARCHING FOR ROOTS

4.1 FINDING ROOTS

A root-finding algorithm is a numerical method, or algorithm, for finding a value x such that $f(x)=0$, for a given function *f*. Such an x is called a root of the function *f*. This type of problem occurs often in physics and science in general, typically as a starting point or intermediary process of a larger problem, though sometimes it is *the* problem.

Generally, computing the root of a function cannot be done analytically and this is especially true when a function is not represented by a low order polynomial. Closed-form solutions for the roots exist for polynomials up to the fourth order

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

However, no such general solutions exist for order five polynomials or higher. Factorization can be used in finding the roots of a polynomial equation but tends to be viable only for well-chosen coefficients, that is, no messy fractions to deal with. For equations that are not polynomial, analytical solutions are few and far between.

Finding a root of $f(x)-g(x)=0$ is the same as solving the equation $f(x)=g(x)$. Here, x is called the unknown in the equation. Any

equation can take the form $f(x)=0$, so equation solving, that is, finding x, is the same thing as computing a root of a function. One of the first techniques you should have been taught to find the root of $f(x)=g(x)$ is to plot graphs of the two functions on the same coordinate system; the x value of where the two functions intersect is the root. Clearly, this has accuracy limitations stemming from the precision of the human eye and the thickness of pencil lines. Unless one is willing to draw a massive graph, this technique should be reserved for providing a rough estimate of the root, which can be passed as an initial guess to a numerical root-finding algorithm. Root-finding methods, provided with an initial guess, use iteration to produce a sequence of numbers that hopefully converge toward a unique value, the root you wish to find. The methods are recursive in nature, that is they compute subsequent values based on current and/or previous values of x, $f(x)$, and derivatives of $f(x)$ where appropriate.

The behavior of root-finding algorithms is studied in numerical analysis. Algorithms perform best when they take advantage of known characteristics of the given function, and typically you find those specific algorithms perform better for particular functions. To evaluate the usefulness of a particular root-finding method, we should test its robustness in achieving reliable results, its ability to find closely located roots, and its rate of convergence, in that order.

4.1.1 Bisection

The Bisection method is a root-finding algorithm that repeatedly bisects (halves) an interval and then selects a subinterval in which a root must lie for further processing. It is a simple and robust method, but it is also relatively slow. Because of this, it is often used to obtain a rough approximation to a solution which is then used as a starting point for more rapidly converging methods (e.g., the Newton–Raphson method discussed in the next section).

In general, we wish to solve $f(x)=0$ that is defined on an interval $[a,b]$, and $f(a)$ and $f(b)$ have opposite sign. So long as $f(x)$ is continuous along with this interval then the limits of the interval must contain, or bracket, at least one root. We then halve the interval size and retain the bracket that must contain a root, at the limits of the interval the value of the function has the opposite sign. This step

is repeated several times and the limits should approach the value of the root. The first two iterations of this process are illustrated in Figure 4.1. Typically, the method is repeated until some desired accuracy is achieved, or we have performed a particular number of iterations.

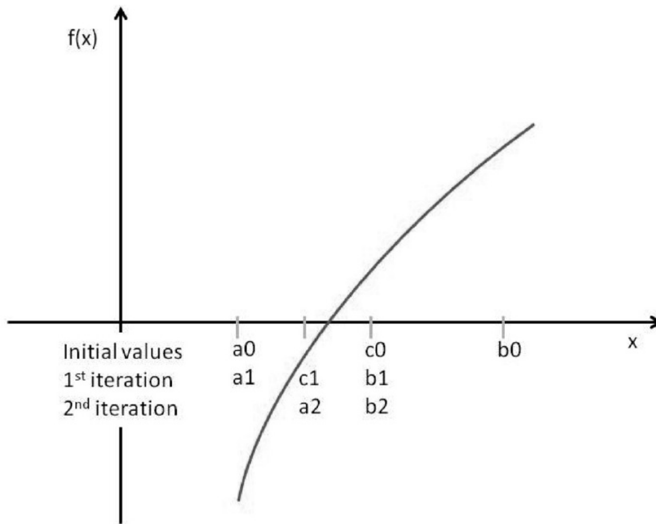

FIGURE 4.1: Illustration of the Bisection method showing the initial values and two subsequent iterations.

Explicitly, we find the midpoint between a and b

$$c = \frac{(a+b)}{2} \tag{4.1}$$

and evaluate the function at the midpoint, $f(c)$. If $f(a)$ and $f(c)$ are of opposite sign, then the method sets c as the new value for b. Else if $f(b)$ and $f(c)$ are of opposite sign the method sets c as the new value for a. In either case, the updated $f(a)$ and $f(b)$ are of opposite sign, so the method is applicable to this smaller interval.If $f(c)=0$ then c is the root and the process stops.

There is a trick to determining whether the function evaluations at the limits of the interval are of the opposite sign without the need to assess them individually. Taking the product of two numbers with the same sign always gives a positive result. Conversely, the product

of two numbers with opposite signs always gives a negative result. Hence, if the product of $f(a)$ with $f(b)$ is negative then they must be of opposite sign, if positive they must have the same sign. We can use this information in a logical condition expression of an **`if`** statement in C++ to determine which half interval to keep and which to discard.

The program *bisection.cpp* performs the Bisection root-finding algorithm on the function $f(x)=\cos(x)-x=0$. This uses the `RootSearch` class to perform the Bisection search, specifically the `Bisection` class. Similar approaches can be found in ref: Pang pp. 62–63, which is written in Java, and in ref: DeVries pp. 41-51 written in Fortran, which provides a nice comparison between the different programming languages. After compiling and running the code in *bisection.cpp* you should find the value for the root to be $x=0.739$ (3sf). The precision of this value is governed by the data member `m_tolerance`. This parameter sets the minimum value of *relative* error that we will tolerate in the solution for the root and can be modified via the relevant setter member function.

As an aside, although the `RootSearch` classes are written to be mathematically correct they are somewhat poorly designed in terms of their interface.

4.1.2 Newton–Raphson

The Newton–Raphson method, named after its creators Isaac Newton and Joseph Raphson, is another method for finding successively better approximations to the roots of a function. Derivation of the method can be done by considering the geometry of a function in the neighborhood of the root. Consider Figure 4.2 that illustrates this point. Using the gradient of the function at an initial guess for the root we can arrive at a better approximation. This is done by tracing the tangent to the function at the initial guess back to the x-axis. Repeating this method using the new value of x we arrive at an even better approximation for the root. By applying the method several times the approximations should converge on the root to some desired accuracy.

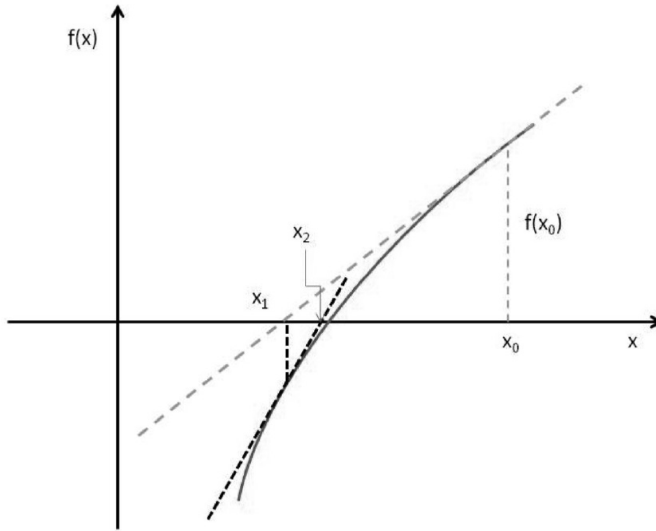

FIGURE 4.2: Sketch of the Newton-Raphson method show the initial value and two subsequent iterations.

Using Figure 4.2, we know that

$$f'(x_0) = \frac{f(x_0)}{(x_0 - x_1)} \tag{4.2}$$

where $f'(x)$ is the gradient of the function at x. Rearranging Equation (4.2) to solve for the improved approximation to the root, x_1, we obtain

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{4.3}$$

where x_0 is the initial guess. We can generalize this for the nth iteration

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \tag{4.4}$$

Note that we must be able to find the first-order derivative for this method to work. This method can also be derived from the Taylor expansion of the function about the root.

The program *newtonRaphson.cpp* implements this method on the same function we looked at with the Bisection method. You can, of course, change the function and its derivative to any you wish to study (so long as you can find the first derivative analytically) without the need to change anything in the class implementation. The initial guess will have to be modified to be close to the root of the equation you choose.

Generally, the Newton–Raphson method is good and if it converges it will converge rapidly. However, be wary that this convergence is very much dependent on the choice of the initial guess. Too far away from the root and the method will fail to converge. In some special cases, the value will not converge at all; try $f(x) = x^3 - 2x + 2$ with an initial guess of one and see if you can figure out what is happening (remember CTRL-c terminates an executing program). Another issue with convergence is its reliance on the first derivative of the function in the neighborhood of the root. What happens if the gradient of the function approaches zero at the root? In general, you will likely be finding roots in functions that do not have a nice and precise derivative, and you will have to approximate it somehow. Next, we discuss a method of root searching that does just that.

4.1.3 Secant

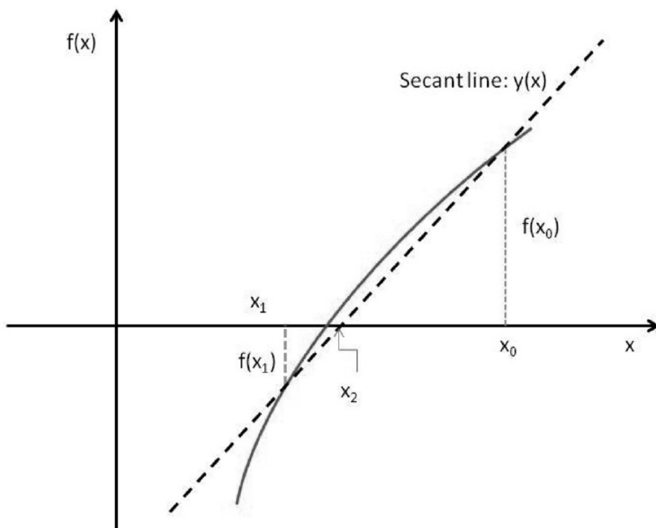

FIGURE 4.3: Sketch of the Secant method with initial values and one subsequent iteration.

The Secant method uses a series of Secant lines (a straight line that cuts a curve in two places) to find better approximations of a root of a function. As with the Newton–Raphson method, we can derive the recursion formula by considering the geometry of a function around the neighborhood of the root. Figure 4.3 illustrates the derivation of the Secant method.

Starting with two points x_0 and x_1 that lie close to the root we draw a line through the points at $f(x_0)$ and $f(x_1)$. The equation for this straight line is given by

$$y(x) = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1) + f(x_1) \tag{4.5}$$

We wish to find the x value at which this line intersects the x-axis, in other words we find the x where $y(x) = 0$. The result is

$$x = x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)} \tag{4.6}$$

This value of x will be a better approximation of the root. We can then use this improved approximation, which we label x_2, with x_1 to perform the same process again to obtain an even better approximation of the root. By repeating the process iteratively, we can find an approximation that lies within some desired accuracy of the actual root. We can generalize Equation (4.6) for the nth iteration giving the recursion formula

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} \tag{4.7}$$

Some readers may have noticed that Equation (4.7) looks like the Newton–Raphson method, but with the first-ordered derivative being replaced with its finite difference approximation; we look at finite difference approximations in Chapter 6. In the limit of the approximations converging on the root the second term in Equation (4.7) does indeed approach the second term of the Newton–Raphson method. The Secant method should therefore be used when we do not have an analytical equation for the first derivative of the function.

Like the Newton–Raphson method, the Secant method will fail to converge if the starting values are not sufficiently close to the root. It should be noted that the Secant method is more rapidly convergent than the Bisection method, but less so than for the Newton–Raphson method. Obviously, you are not just going to take my word for it, are you? The `Secant` subclass can be used similarly to the `Bisection` subclass such that writing a new program using the Secant method should be straightforward.

4.2 HYBRID METHODS

4.2.1 Bisection–Newton–Raphson

From the previous section, we note that the Bisection method is robust to initial guesses but slow to converge. Whereas the Newton–Raphson method will converge rapidly but only if the initial guess is relatively close to the root, and in some circumstances may fail to converge at all. Also, the Newton–Raphson method may fail to converge if the gradient of the function in the neighborhood of the root approaches zero. We would therefore like to combine the reliability of the Bisection method with the rapid convergence of the Newton–Raphson method so that for any general function we can find its roots with relative ease.

We can do this by making a hybrid method that decides whether to take a Newton–Raphson step or a Bisection step. As computers cannot think for themselves, we as programmers must provide some logical criteria to determine the step to take. Crucially, if an NR step takes the next approximation outside of our interval, then we should discard it and apply a Bisection step instead; else we accept the NR step. To do this, let us consider our Bisection interval $[a,b]$ with some best approximation to the root, r, contained within that interval. To accept the NR step the following inequality has to be satisfied

$$a \le r - \frac{f(r)}{f'(r)} \le b\,. \tag{4.8}$$

Now while we could use this inequality as the conditional expression in an **`if`** statement the coding becomes lengthy and rather difficult to read. We can make our lives easier by rearranging the inequality into the form

$$y \geq 0 \geq z. \tag{4.9}$$

In other words, to satisfy the inequality (4.8) such that the NR step is accepted, the left-hand expression, y, must be positive or zero, and the right-hand expression, z, must be negative or zero. We can therefore apply the trick of comparing the product of y and z to zero to determine whether the inequality has been satisfied. To rearrange inequality (4.8) into the form of (4.9) we subtract r, multiply through by $-f'(r)$, and lastly, subtract $f(r)$ resulting in

$$(r-a)f'(r)-f(r) \geq 0 \geq (r-b)f'(r)-f(r). \tag{4.10}$$

If the product of the left-hand side with the right-hand side is negative then the NR step falls within the interval and should be accepted; else the product is positive and a Bisection step is applied instead. Note that a negative product will also be produced if the RHS is positive and the LHS is negative. However, this satisfies the reverse inequality of (4.10) that when tracked back to inequality (4.8) requires that $a \geq b$, which, by *definition,* is false (unless they are equal and in which case you have found the root).

The initial value for the best approximation, r, can be taken as one of the initial interval limits, it really does not matter. In fact, this is what we were trying to achieve; if the initial guess gives a lousy Newton–Raphson step, then the Bisection method is used to improve the initial guess and continues to do so until the NR step falls within the interval. However, it is likely that if we have reasonable guesses for the interval limits of a root, the one with the smallest function evaluation will lie closest to the root and should be taken as the initial best guess.

If we do not have an analytic equation for the first-order derivative then we should replace the Newton–Raphson method with the Secant method, so that we only have function evaluations, no derivatives. Provided are hybrid Bisection–Newton–Raphson and Bisection-Secant `RootSearch` subclasses in the library. They provide an

implementation of the inequality 4.10 and you should satisfy yourself they provide the functionality described. Write programs using these classes to search for roots on appropriate functions.

4.2.2 Brute Force Search

In the previous sections, we have developed solid methods to accurately compute the roots of a function, so long as we know the rough locations of those roots in advance. The problem then is finding those rough locations. One straightforward technique is to graph the functions either by hand or using a plotting program and obtain those bounds by eye. This is recommended when finding the roots of a function is the problem to solve. But what if the root-finding is only one part of a bigger problem? It would be impractical to manually locate the rough location of roots for numerous functions in this case.

Typically, we use an exhaustive root search across a region of interest (ROI) for the function. That is, starting at the minimum value of the ROI we step the value of x by some small amount and check to see if the function has changed significantly within that small step. If it has, we have found the bounds of at least one root, if not we continue the search. This continues until the whole ROI has been covered. How then do we decide on the step size? Too small and we make our rapidly converging root-finding algorithms redundant; too large and we run the risk of stepping over multiple roots (for an even number of roots this means missing them entirely; for an odd number, in essence, only one root is detected). Choosing the step size is an educated guess and is very much dependent on the function under investigation.

The program *rootSearch.cpp* showcases our root searching classes on the Legendre polynomial:

$$P_8 = \frac{6435x^8 - 12012x^6 + 6930x^4 - 1260x^2 + 35}{128} \tag{4.11}$$

In its current state, it only finds a single root of P_8 in [0,1]. It is known that P_8 has four positive roots between 0 and 1. Modify this program to find all the positive roots of this polynomial in the range

[0,1]. The `RootSearch` base class has a member function `find_brackets` that perform a brute force search for roots of the given function, returning **`true`** when (at least) one root is found.

4.3 WHAT'S THE POINT OF ROOT SEARCHING?

For instance, finding the roots to the Legendre polynomials is an important step in determining the evaluation points for Gaussian quadrature; extremely accurate methods for numerically determining the value of an integral. As a more direct example, and one we shall discuss here, finding the roots of an equation can help us calculate the energies of electrons bound in a finite square well.

4.3.1 The Infinite Square Well

This problem is sometimes referred to as the particle in a box model. Classically the motion of the particle is governed by Newton's equations, potential fields put forces on masses causing them to accelerate or change direction. At the quantum level, Newton's equations are replaced by Schrödinger's such that for a particle of mass m moving through a (one-dimensional) potential $V(x)$ we have

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} = E\psi(x) - V(x)\psi(x) \tag{4.12}$$

where $\hbar$ is Planck's constant h over 2π, E is the total energy of the system, and $\psi(x)$ is the wavefunction of the system. Note that this is the time-independent version of the Schrödinger equation; time-dependent versions also exist. Like the Newtonian equations, we solve Equation (4.12) for the unknown, in this case, $\psi(x)$. Although there is still some considerable debate over the nature of the wavefunction, certain observable quantities do depend on its form. For instance, the quantity $\psi^*(x)\psi(x)$ describes its probability function, that is, the chance of finding the quantum particle at a particular location. More precisely the quantity $\psi^*(x)\psi(x)dx$ is the probability of finding the particle in the region x to $x+dx$.

The simplest form of the particle in a box model considers a one-dimensional system. Here, the particle may only move backward and forward along the x-axis with impenetrable barriers at either end. The walls of this one-dimensional box may be visualized as regions of space with an infinitely large potential energy. Conversely, the interior of the box has zero potential energy everywhere. This means that no forces act upon the particle inside the box, and it can move freely in that region; remember that forces are proportional to the negative of the gradient of the potential field that causes them. If the particle touches the sides of the box, it experiences an infinitely large force that pushes it back into the interior of the box; here the gradient of the potential is infinite as we have a discontinuity in the potential itself. As such the potential field is modeled by

$$V(x)=\begin{cases} 0, & 0\le x\le L \\ \infty, & otherwise \end{cases} \tag{4.13}$$

where L is the length of the box and x describes the position of the particle within the box.

We now consider the wavefunction of the system both inside and outside the box. We know $\psi(x)$ must be zero outside the box as the particle is confined by the potential. Inside the box, the potential is zero everywhere thus Schrödinger's equation becomes

$$\frac{d^2\psi}{dx^2}=-\frac{2m}{\hbar^2}E\psi(x). \tag{4.14}$$

The general solution of this differential equation is

$$\psi(x)=A\sin(kx)+B\cos(kx), \tag{4.15}$$

where

$$k=\sqrt{\frac{2mE}{\hbar^2}}, \tag{4.16}$$

and A and B are constants to be determined. As it stands, we currently lack the information to solve this problem. However, physical reasoning comes to our aid. We expect that the probability of finding the particle anywhere within the one-dimensional space be a continuous function. As the probability of finding the particle outside

the box is zero then we require the wavefunction of the particle inside the box to vanish as it approaches the walls of the box. In other words, the physics of the problem has given us the boundary conditions such that

$$\psi(0) = 0 \tag{4.17}$$

and

$$\psi(L) = 0\,. \tag{4.18}$$

Imposing these boundary conditions on the general solution at $x = 0$ we find that

$$\psi(0) = A\sin(0) + B\cos(0) = 0 \tag{4.19}$$

which implies $B = 0$, and at $x = L$ we find that

$$\psi(L) = A\sin(kL) = 0. \tag{4.20}$$

Equation (4.20) is satisfied either if A is zero or if $sin(kL)$ is zero. Setting $A = 0$ is rather an uninteresting case as it sets the wavefunction zero everywhere, which implies that we have no particle in our system. For $sin(kL)$ to be zero then

$$kL = n\pi \tag{4.21}$$

where n is an integer. After substitution of Equation (4.16) and some rearrangement, we find that

$$E_n = \frac{n^2\pi^2\hbar^2}{2mL^2}. \tag{4.22}$$

Hence, we have found a set of discrete energies that will satisfy our physical boundary conditions and the differential equation. These are referred to eigenvalues of the system. The corresponding wavefunctions of these energies are known as the eigenfunctions. Take note that not all energies are permitted; only those that satisfy Equation (4.22) are allowed. If the particle were macroscopic (that is, not quantum) then it could take any value of (kinetic) energy it liked within the confines of the box.

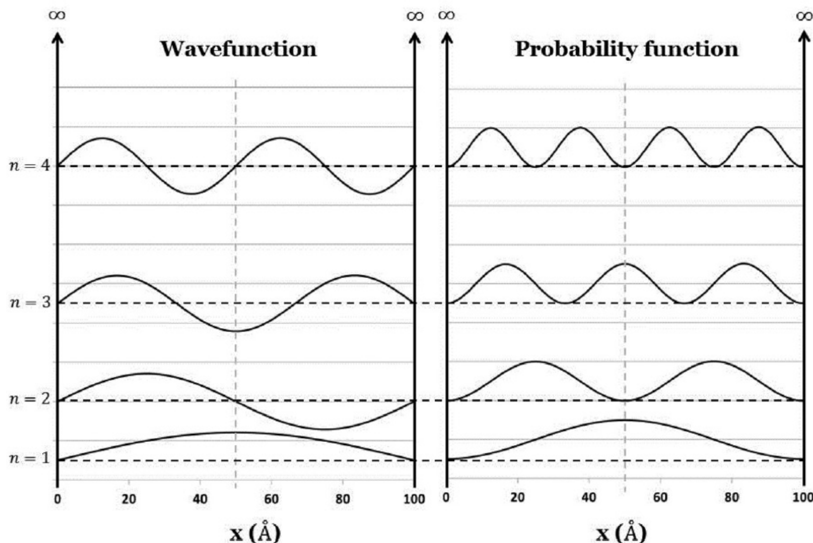

FIGURE 4.4: Wavefunctions and probability functions of the first four energy states of the infinite square well.

Figure 4.4 shows the wavefunctions and probability functions for the first four permitted energies in an infinite quantum well. Typically, we refer to E_1 as the ground state energy, and it is the lowest permitted energy the particle can attain sometimes called the zero-point energy. Subsequent states we call excited states such that energy has been absorbed by the particle to jump from lower states to higher states. Note that these states are standing or stationary waves such that they are formed from two progressive waves traveling in opposite directions. These progressive waves are reflected by the infinite barrier and interact in such a way to produce a standing wave.

Now that we have the energies, we could go back to our functions for $\psi(x)$ and find the coefficients A such that they normalize the wavefunctions, that is,

$$\int_0^L \psi^*(x)\psi(x)dx = 1, \tag{4.23}$$

which is the mathematical statement that the particle must be somewhere within the box. Note that as we only consider real

wavefunctions the integral function reduces to ψ^2. In this case, it is possible to show that $|A| = \sqrt{2/L}$.

The infinite square well is only appropriate for an introduction to quantum physics. It nicely shows the discreteness of bound energy states in the well and can be solved analytically. However, as we have the computer at our disposal, we could solve something a little more difficult.

4.3.2 The Finite Square Well

The finite square well is somewhat more realistic than the infinite square well. We define the potential as

$$V(x) = \begin{cases} V_0, & x < -a \\ 0, & -a \le x \le a \\ V_0, & x > a \end{cases} \tag{4.24}$$

Note that in this case the well is defined symmetrically about the origin of the x-axis, rather than having a barrier at $x = 0$. We now consider the three distinct regions namely the region left of the well, the well itself, and the region right of the well. In Figure 4.5, we label these regions as I, II, and III, respectively, and consider the implications of the potential field on the wavefunction in these three regions.

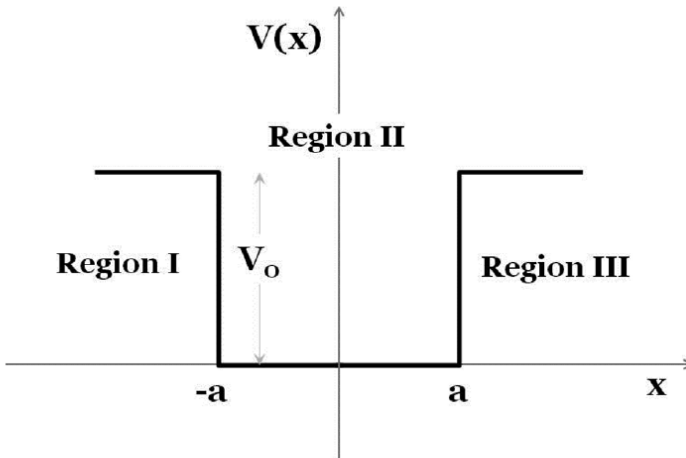

FIGURE 4.5: The finite square well potential.

First, for particles with energy greater than the height of the well V_0 their wavefunctions are unbound, in other words, they can move freely, and have any energy. Interestingly, as the particle moves over the well it loses potential energy, which is transformed into kinetic energy and the particle gains momentum. This shows an increase in the wavenumber of the wavefunction as the particle travels across the well, c.f. de Broglie (pronounced like Troy) momentum. This can also be seen in the differential equation. For a constant potential across x, Equation (4.12) has the form of a simple harmonic oscillator where the $E-V(x)$ term plays the role of the spring constant. As we go from regions I–II, the potential drops from V_0 to zero thus increasing the "spring constant" and the frequency of the oscillations of the particle. The opposite is true as we go from regions II–III. (Strictly speaking, the wave is progressive rather than stationary so we should use the time-dependent version of Equation (4.12) to govern the physics of motion, though the outcome would at least be qualitatively the same. For arguments sake, you can consider the unbound wavefunctions are the bound states of an infinitely wide quantum well.)

We now consider the more interesting case of particles with energy less than V_0. Starting in region I, we can write the Schrödinger equation as

$$\frac{d^2\psi}{dx^2} = \frac{2m}{\hbar^2}(V_0 - E)\psi(x), \tag{4.25}$$

which has the general solution

$$\psi_I = Ce^{\beta x} + De^{-\beta x}, \tag{4.26}$$

where

$$\beta = \sqrt{\frac{2m(V_0 - E)}{\hbar^2}}. \tag{4.27}$$

We know from experiments that the wavefunction of the particle can penetrate the finite barrier; a place where it is forbidden to go according to classical physics. If the barrier in region I had finite width, then there is a probability that the particle would be found to the left of region I; this is known as quantum tunneling. We also know from the experiment that the probability of finding the particle

to the left of region I decrease as the width of the barrier increases and *vanishes to zero* in the limit of the width of the *barrier* going to infinity. For the general solution to satisfy this physical observation D must be zero; remember we are in the negative half of the x-axis thus, as we go deeper into region I, $e^{-\beta x}$ represents a growth function in this direction.

Moving to region II the general solution of the Schrödinger equation is the same as we found for the infinite well case restated here

$$\psi_{II} = A\sin(\alpha x) + B\cos(\alpha x), \tag{4.28}$$

where we have swapped k for α such that

$$\alpha = \sqrt{\frac{2mE}{\hbar^2}}. \tag{4.29}$$

And in region III, which is identical to region II apart from the location on the x-axis, we find that

$$\psi_{III} = Fe^{-\beta x} \tag{4.30}$$

with the same reasoning for dropping the growth term. We expect the wavefunction to be continuous across x. This is again due to physical reasoning that we do not expect a sudden jump in the probability of the particle's whereabouts. In addition to this, we also expect that the derivative of the wavefunction to be continuous across x. In the infinite well case, the discontinuity in the derivative was caused by the infinite nature of the barrier, now we have finite barriers.

To proceed we now consider the boundary conditions of the system. At $x = -a$ we obtain the following relation

$$-A\sin(\alpha a) + B\cos(\alpha a) = Ce^{-\beta a}, \tag{4.31}$$

for the wavefunction and

$$\alpha A\cos(\alpha a) + \alpha B\sin(\alpha a) = \beta Ce^{-\beta a} \tag{4.32}$$

for the derivative. While at $x = a$ we find that

$$A\sin(\alpha a) + B\cos(\alpha a) = Fe^{-\beta a} \tag{4.32}$$

for the wavefunction and

$$\alpha A\cos(\alpha a) - \alpha B\sin(\alpha a) = -\beta F e^{-\beta a} \tag{4.33}$$

for the derivative.

Taking $A = 0$ and $B \neq 0$ such that we have even parity states we find that the following must be true:

$$B\,cos(\alpha a) = Ce^{-\beta a} = Fe^{-\beta a}$$
$$\therefore C = F, \tag{4.34}$$

and

$$\alpha B\sin(\alpha a) = \beta Ce^{-\beta a} = \beta B\cos(\alpha a)$$
$$\therefore \alpha\tan(\alpha a) = \beta. \tag{4.35}$$

For odd parity states where $B = 0$ and $A \neq 0$ we find similar relations:

$$-A\sin(\alpha a) = Ce^{-\beta a} = -Fe^{-\beta a}$$
$$\therefore C = -F \tag{4.36}$$

and

$$\alpha A\cos(\alpha a) = \beta Ce^{-\beta a} = -\beta A\sin(\alpha a)$$
$$\therefore \alpha\cot(\alpha a) = -\beta\,. \tag{4.37}$$

To find the energies and wavefunctions of the finite square well we must find the roots of Equations (4.35) and (4.37). And it just so happens that we have already developed the classes that can do this job.

4.3.3 Programming the Root Finder

Before launching into the code let us just remind ourselves of the nature of the problem we are trying to solve. While units like Joules, kilograms, and meters are all well and good for macroscopic objects, at the quantum level these become extremely cumbersome for quantum objects; especially when performing calculations with a computer. For example, the mass of the electron is roughly 9.1×10^{-31} kg and has a charge of about 1.6×10^{-19} coulombs; these are hard

numbers that will lend themselves well to precise computations. Some advocate the use of dimensionless variables such that we set a particular coefficient to unity to remove any issues of precision. For instance, we could "choose" units that set the value of $\hbar^2 / 2m = 1$; it does not matter what those units are only that the coefficient is one. However, this requires converting the result from the "dimensionless" units back to SI units or any units of choice which has its advantages but can be non-intuitive and confusing for novice programmers. An alternative is to use explicit unit conversions before computing anything and therefore have results that are immediately identifiable in SI units. The unit conversion will be different for different problems, but the common goal is to make the coefficients have an exponent of one. Typically, we can use the natural units of the problem at hand. Case in point, if we use electron masses, Angstroms, and electron volts as our units of mass, length, and energy respectively then

$$\hbar^2 = 7.61996386\, m_e\, eV\, \mathring{A}^2. \tag{4.38}$$

To see how we arrived at this number let us start with the normal definition of $\hbar$ such that

$$\hbar = \frac{h}{2\pi} = \frac{6.62606957 \times 10^{-34}}{2\pi} Js \tag{4.39}$$

and perform some dimensional analysis on the units. In SI base units the units for Planck's constant squared become

$$[J]^2 [s]^2 = [kg]^2 [m]^4 [s]^{-2}, \tag{4.40}$$

which we can rearrange to give

$$[kg]^2 [m]^4 [s]^{-2} = [kg][m]^2 [kg][m]^2 [s]^{-2} = [kg][m]^2 [J]. \tag{4.41}$$

To convert to our computer-friendly units, we note the following conversions:

$$1\, m_e = 9.10938291 \times 10^{-31}\, \text{kg};$$
$$1\, \text{eV} = 1.60217657 \times 10^{-19}\, \text{J};$$

and

$$1\,\text{Å} = 1 \times 10^{-10}\ \text{m}.$$

Combining these to compute $\hbar^2$ in Equation (4.38) we have

$$\hbar^2 = \frac{6.62606957^2}{4\pi^2 \times 9.10938291 \times 1.60217657} \times \frac{10^{-68}}{10^{-31} \times 10^{-19} \times 10^{-20}}$$

$$\therefore \hbar^2 = 0.0761996386 \times 10^2\, m_e\, \text{eVÅ}^2. \tag{4.42}$$

By explicitly setting these unit conversions we know that when defining the well width say we do so in units of Angstroms. Or when defining the potential barrier height or computing the energy of the bound electrons we are using units of electron volts.

Let us imagine we are attempting to find the energy and wave-function of the lowest bound state, the ground state. From the results of the infinite square well case, we would expect this state to be of even parity. Even parity states have the characteristics that $\psi \neq 0$ and $\psi' = 0$ at the middle of the well. Odd parity states have those characteristics reversed. For even parity states we are trying to find the energy E which satisfies the following equation

$$f(E) = \alpha \tan(\alpha a) - \beta = 0 \tag{4.43}$$

where we remind you that

$$\alpha = \sqrt{\frac{2mE}{\hbar^2}}, \tag{4.44}$$

and

$$\beta = \sqrt{\frac{2m(V_0 - E)}{\hbar^2}}. \tag{4.45}$$

Now while Equation (4.43) is perfectly acceptable as a mathematical object note that it contains properties that are abhorrent to a computer. Specifically, the tangent function contains singularities whenever $ka = n\pi$, where n is an integer, due to the cosine function being zero at these points. We can circumvent this issue by rewriting Equation (4.43) in its component terms such that

$$f(E) = \beta \cos(\alpha a) - \alpha \sin(\alpha a) = 0. \tag{4.46}$$

Here we have removed the singularities and the computer thanks us for that.

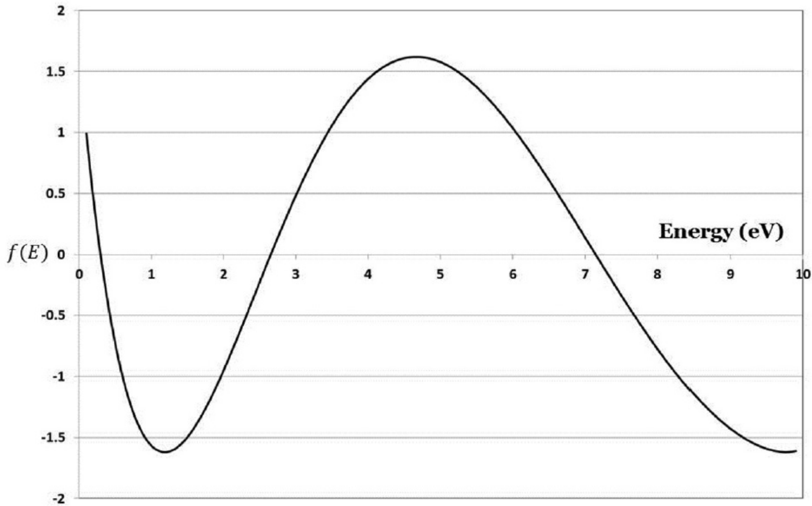

FIGURE 4.6: Function of energy where the roots define the energy eigenvalues for a finite square well of width 10Å and height 10 eV.

The root-finding subroutines we have developed to date demand that a root be bracketed; where do we start looking? We know that our bound wavefunctions must exist (if they exist at all) within the confines of the well. That is, they must exist only for energies between zero and the height of the potential barrier V_0. We could perform an exhaustive search on $f(E)$ but let us see if we can't do a little better by plotting the function on which we wish to perform the root search. Figure 4.6 shows the function $f(E)$ for an electron bound in a well with the parameters $a = 5\text{Å}$, and $V_0 = 10$ eV; remember that the electron can only have energies that are equal to the roots of this function. We can clearly see three roots: the first between 0.0 and 0.5 eV, the second between 2.5 and 3.0 eV, and the third between 7.0 and 7.5 eV. Performing the same procedure for odd parity states we find three roots bracketed between 1.0 and 1.5 eV, 4.5 and 5.0 eV, and 9.5 and 10.0 eV.

There is a slight rub to this plotting argument; we have plotted the function to see roughly where the roots lie and to avoid an exhaustive search where we would have to perform many function evaluations. However, to plot the function we have had to perform function evaluations anyway at equally spaced points and passed that data to an external program for plotting. We have not actually saved ourselves any effort and in fact, have added some. In other words, we may as well perform the exhaustive search. If so desired, we could then store the function evaluations during the exhaustive search for plotting after the program has finished, providing some insight into the validity of our numerical results. As a *rule of thumb,* we should set the search step length no larger than 10^{-2} of our search range to avoid skipping over roots, and no smaller than 10^{-4} of our search range to avoid an excessive number of function evaluations and subverting the intent of the root searching subroutines we have developed, if you consider ten thousand not being an excessive number! That said, the step length for an exhaustive search will very much depend upon the function being investigated. For instance, we can clearly see from Figure 4.6 that a step length of one would find brackets for all three roots.

Once we have the brackets for the roots, they are passed to a root searching algorithm, say our Bisection-Secant hybrid method, for further refinement up to an accuracy specified by a user-defined tolerance. We now have the energies (in eV) of the bound states of our finite square well. All that remains to do is substitute these values back into our equations for the original problem to determine their corresponding wavefunctions.

The *finiteSquareWell.cpp* source file contains the code to implement the discussion above for a finite square well of width 10*Å*, and barrier height of 10 eV. If you have OpenCV installed and have included the visualization module in the library the program will plot the resultant wavefunctions for an electron trapped in this specific well. If not, the program writes to file the data computed for plotting elsewhere.

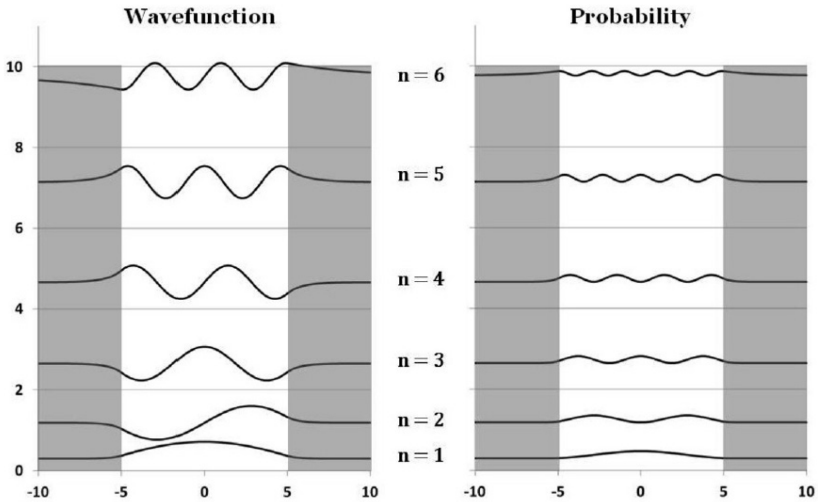

FIGURE 4.7: The wavefunctions and probability functions for a finite square well with the parameter defined in the text (units are Å and eV).

Figure 4.7 plots the results of this code for an electron trapped in the well. Here we plot the *normalized* wavefunction from this computation on the left (see Exercise 5), and the corresponding probability function on the right; the zero baselines are aligned to their matching energy eigenvalue (the code requires additional functionality to obtain the probability functions). This result will come in very handy as a check when we attempt to find the bound states for an arbitrary potential $V(x)$ in Chapter 11 on an advanced ordinary differential equation solver.

EXERCISES

4.1. Run the Bisection, Newton–Raphson, and Secant method programs to determine the number of iterations required to find the root of the equation $f(x) = cos(x) - x$ to an accuracy of eight significant figures. Comment on the dependence of conversion on the choice of initial guesses. (Tip: You may want to include a conditional exit to avoid infinite loops).

4.2. Modify the hybrid method programs so that it prints to screen when it takes a Bisection step and when it takes an NR/Secant step. Using the various initial guess values from the previous exercise perform the same root-finding computation using one of the hybrid methods we have developed. Verify that the hybrid methods are robust to initial guess and comment on how often Bisection is used in comparison with the other method.

4.3. Modify the brute force search so that instead of attempting to find a set number of roots it only searches over a given interval of interest, reporting back the number of roots found in that interval, as well as the bracket for each.

4.4. The Lennard–Jones potential describes the approximate interaction between a neutral pair of atoms and has the form

$$V_{LJ} = 4\varepsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right)$$

where r is the distance between the atoms, and ε and σ are properties of the potential to be determined. At what value of r does the potential V_{LJ} equal zero? The size and nature of the force between the atoms are given by the magnitude and sign of the first-order derivative of the potential with respect to r. At what value of r do the forces balance between the atoms, and what is the value of V_{LJ} at this point? Confirm these results using the root-finding programs we have developed in this chapter. As a bonus question, what is the minimum energy required to tear the atom pair apart according to the Lennard–Jones potential?

4.5. Add code to the finiteSquareWell.cpp source file to compute the probability functions for each wavefunction found (Tip: there is an overloaded operator that allows you to perform elementwise multiplication of two vectors in the code library)

4.6. Is a 10 eV tall, 10 Angstrom wide quantum well physically sensible? Study the effects of varying the well width and well height on the bound states on the bound states in the well. Do we always get at least one state, that is, the ground state?

CHAPTER 5

NUMERICAL QUADRATURE

Numerical integration constitutes a broad family of algorithms for calculating the numerical value of a definite integral. The term is also sometimes used to describe the numerical solution of differential equations that are described in Chapter 6 of this book. This chapter focuses on the calculation of definite integrals. The term numerical quadrature (often abbreviated to just quadrature) is a synonym for numerical integration, especially as applied to one-dimensional integrals.

The basic problem considered by numerical integration is to compute an approximate solution to a definite integral:

$$\int_a^b f(x)\,dx\,. \tag{5.1}$$

If $f(x)$ is a smooth, well-behaved function and the limits of the integration are bounded, there are several methods of approximating the integral using numerical integration to the desired precision.

The first two numerical integration schemes we discuss next should be familiar to you and provide intuitive and illustrative examples of what all numerical integrations schemes are doing regardless of their complexity.

Throughout this chapter, C++ programs that perform the quadrature methods are discussed. There are `Quadrature` classes

found in the library that can be used to shortcut the development. Feel free to use them but ensure you have at least looked at how they are implemented. You will also find other quadrature rules in the library that implement Romberg's method and Gauss type quadrature. We discuss these quadrature methods in Chapter 10 of this book.

5.1 SIMPLE QUADRATURE

5.1.1 The Mid-Ordinate Rule

The mid-ordinate rule computes an approximation to a definite integral, made by finding the area of a collection of rectangles whose heights are determined by the values of the function at certain discrete, evaluation points along the interval.

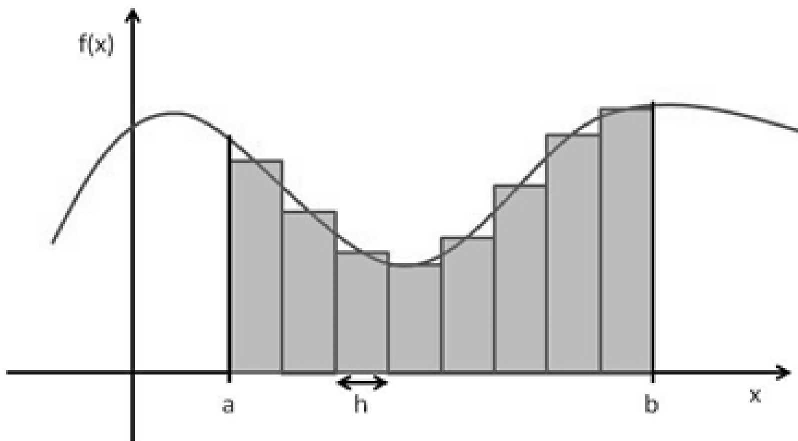

FIGURE 5.1: Illustration of the mid-ordinate rule.

Figure 5.1 illustrates the mid-ordinate method. Specifically, the interval $[a,b]$ over which the function is to be integrated is divided into N equal subintervals of length $h=(b-a)/N$. The height of the rectangle is then determined to be the value of the function found at the mid-point between each subinterval hence the name. The approximation to the integral is then calculated by adding up the

areas (base multiplied by height) of the N rectangles, giving the formula:

$$\int_a^b f(x)dx \approx h\sum_{n=0}^{N-1} f(x_n) \tag{5.2}$$

where $x_n = a + (n + 1/2)h$. As N gets larger, this approximation gets more accurate. As N approaches infinity, h becomes infinitesimally small (approaches dx) and we have the definition of an integral. Why is this impossible to do with a computer?

Write a C++ program that uses the mid-ordinate rule on a function that has an analytic solution. This is so we can confirm the accuracy of the method. The improvement in accuracy of the mid-ordinate method should be on the order of h, written $\mathcal{O}(h)$. In other words, if the subinterval width h is halved, that is, N is doubled, then the error in the numerical approximation of the integral is also halved.

5.1.2 The Trapezoidal Rule

One immediate, and intuitive improvement, we can make to the mid-ordinate rule is to make our subinterval strips approximate the function between the subinterval limits rather than just us the mid-point value. The easiest way to do this is to approximate the function as a straight line between the values for the function at the subinterval limits. In essence, we make the rectangle a trapezoid.

If we consider the entire interval as one strip, as illustrated in Figure 5.2, satisfy yourself that

$$\int_a^b f(x)dx \approx (b-a)\frac{f(a)+f(b)}{2}. \tag{5.3}$$

Equation (5.3) is referred to as the primitive integral. Subdividing this into N strips we obtain

$$\int_a^b f(x)dx \approx h\frac{f(a)+f(b)}{2} + h\sum_{n=1}^{N-1} f(x_n) \tag{5.4}$$

where $h = (b-a)/N$, and $x_n = a + nh$. Equation (5.4) is referred to as the composite (trapezoidal) integral.

The trapezoidal rule should have an error reduction that is proportional to $\mathcal{O}(h^2)$; halve h and you reduce the error by a factor of four. To confirm this, write a C++ program that performs the composite trapezium rule on a function you can integrate analytically.

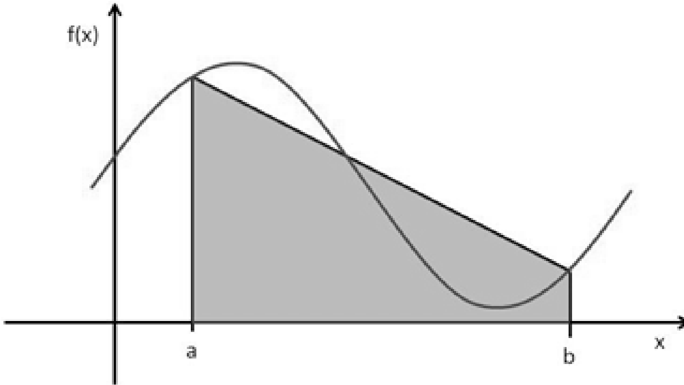

FIGURE 5.2: Illustration of the primitive trapezoidal rule.

5.1.3 Simpson's Rule

As a next step in improving the accuracy of our numerical integration scheme, we might consider approximating the integrand as a piecewise quadratic. This is exactly what Simpson's rule does and is illustrated in Figure 5.3. The derivation of Simpson's rule involves taking a Taylor series expansion about the mid-point of the interval and integrating that expansion. The formula for the primitive Simpson's rule, that is, the whole interval taken as one strip, is

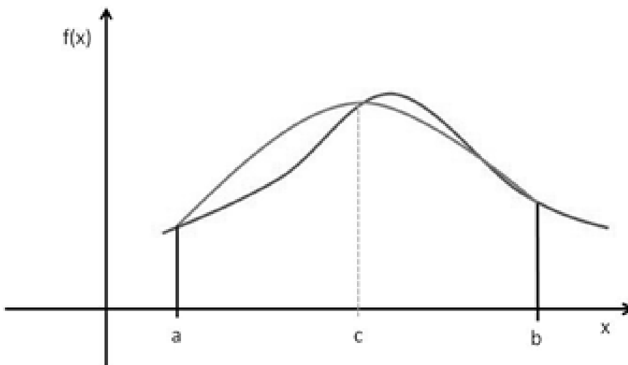

FIGURE 5.3: Illustration of Simpson's rule. Here the definite integral of the function is approximated by area under the quadratic.

$$\int_a^b f(x)dx \approx (b-a)\frac{f(a)+4f(c)+f(b)}{6} \quad (5.5)$$

where c is the mid-point of the interval. We need three function evaluations in this case because we are approximating the function with a quadratic that requires a minimum of three points. The composite Simpson's rule has the form

$$\int_a^b f(x)dx \approx \frac{h}{3}[f(a)+f(b)+4\sum_{n=1}^{N/2} f(x_{2n-1})+2\sum_{n=1}^{(N/2)-1} f(x_{2n})] \quad (5.6)$$

where $h=(b-a)/N$ and $x_n = a+nh$. Again, write a program that performs the composite Simpson's rule for numerical integration and investigate how the error behaves in terms of the strip width. Note that the total number of strips, N, is used as an even number.

5.2. ADVANCED QUADRATURE

5.2.1 Euler–Maclaurin Integration

We could of course keep going with the approximations to the integrand function using higher-ordered polynomials. Indeed, using a cubic polynomial, we are led to Simpson's three-eighths rule, and using a quartic polynomial yields Boole's rule but these soon become very cumbersome to derive and use. The integration scheme is called the Euler–Maclaurin scheme, given by the composite formula

$$\int_a^b f(x)dx = h\left(\frac{f(a)+f(b)}{2}\right)+h\sum_{n=1}^{N-1} f(x_n)+\frac{h^2}{12}\left[f'(a)-f'(b)\right]-$$
$$\frac{h^4}{720}\left[f'''(a)-f'''(b)\right]+\cdots. \quad (5.7)$$

Equation (5.7) can be derived by again considering the Taylor series expansion of the function and its derivatives at the integration limits. See DeVries pp. 153–155 for a neat explanation of the derivation. The first two terms in Equation (5.7) are simply the trapezoid rule, and we can consider the next terms as corrections to that numerical integration scheme. Note that should the first-order

derivatives at the integration limits be near identical, or vanishingly small, the trapezoid rule can give surprisingly accurate results. Can you think of a function whereby its derivatives will be *identical* for a given set of integration limits?

Although the Euler–Maclaurin formula is far superior to any other numerical integration method we have discussed so far it suffers from the drawback that the integrand has to be easily differentiable. If not, we would have to rely on numerical approximations of the derivatives at the integration limits. The accuracy of those derivative approximations should at least match the order of h to which they belong. This quickly becomes impractical.

5.2.2 Adaptive Quadrature

In the previous discussions, it is assumed that the strip width is uniform across the integration interval. To those experienced in numerical integration, this is wasting considerable effort. Typically, we want to determine the value of numerical integration to some predetermined accuracy or tolerance. With our current numerical integration schemes, we can only reduce the error by reducing the size of each strip. For smooth functions this is fine; the contribution of each strip to the total absolute error is roughly the same. However, what if the function is not smooth, or has portions that rapidly change with x, compared to other flat regions. For instance, consider the Lorentzian line-shape function that describes the emission of light from the atoms of an excited gas cloud

$$I(\lambda) = \frac{I_0}{1 + 4(\lambda - \lambda_0)^2 / \Gamma^2} \tag{5.8}$$

where λ is the wavelength of light emitted, λ_0 is the resonant wavelength, I_0 is the peak intensity of emitted light at $\lambda = \lambda_0$, and Γ is a measure of the width of the curve, the full width at half height.

This function is sketched in Figure 5.4; note that a small constant background intensity has been included. Let's say we are performing an experiment to determine how the width of the peak is affected by the pressure of the gas. Being good scientists, we want to ensure that the total number of contained atoms remains constant at each measured pressure level, within some predetermined

tolerance of say 0.1%. One way to do this would be to check the total emitted power of the gas at each pressure, that is, the area under the curve in Figure 5.4. This means integrating the line-shape over some predetermined wavelength range. Let us assume we don't know how to integrate this type of function analytically and so we have to do it numerically (it is actually a standard integral after a simple substitution). The relative error in our numerical approximation should be at least equal to the tolerance we want for our total emitted power measurements and ideally much less, let us say 0.001%. Much of the error in the approximation will be introduced by those strips representing the peak, and we require relatively narrow strips in this region in order to keep the overall error below what we are willing to tolerate. Using a uniform distribution of strips, we would be wasting effort in the relatively flat regions away from the peak; each strip would produce an insignificant portion to the overall error and we could afford to use wider strips in these regions.

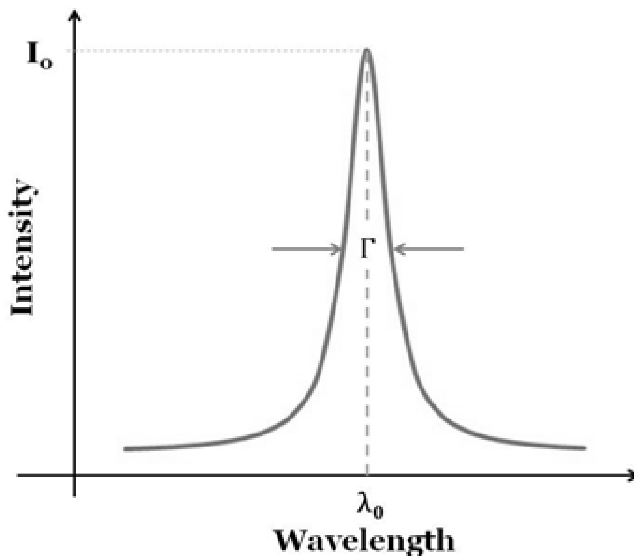

FIGURE 5.4: Sketch of the Lorentzian line shape.

The first approach to this problem might be to manually segment the integration interval into three subintervals: the two flat regions, and the peak. The strips for each subinterval could be chosen so that

the absolute error of each was one-third that required of the total. This approach is perfectly valid but hardly provides a general solution; where do you select the segmentations? What if there is still significant function variation within the selected subinterval? Would greater manual segmentation require more effort?

To provide a more general solution let us consider the behavior of the error of the trapezoid rule. We know that if you halve the strip width then you reduce the error by a factor of four, in other words, the trapezoid rule has an error behavior of $\mathcal{O}(h^2)$. If we denote the trapezoidal approximation using 2^m strips as T_m, then the trapezoidal approximation with twice the number of strips (half the strip width) is given by T_{m+1}. As T_{m+1} has half the strip width its error is reduced by a factor of four compared with T_m. With reference to the Euler–Maclaurin Equation (5.7), we can eliminate the leading error term in our approximation by performing the following calculation

$$T'_{m+1} = \frac{4T_{m+1} - T_m}{3} \tag{5.9}$$

where T'_{m+1} is the improved approximation for 2^{m+1} strips. If it helps you can think of Equation (5.9) as a weighted average between the two approximations T_m and T_{m+1}. As we have eliminated the leading error term from Equation (5.7) the error in our improved approximation is now $\mathcal{O}(h^4)$. It is worth noting here that Equation (5.9) is equivalent to Simpson's rule. We can now estimate the error in the T'_{m+1} by subtracting T_{m+1} giving

$$\varepsilon \approx T'_{m+1} - T_{m+1} = \frac{T_{m+1} - T_m}{3}. \tag{5.10}$$

Note that this is the *absolute* error, **not** a relative error. We can now check this estimated value against the "global" error we want to achieve in our approximation. If this condition is met, we accept the integration, if not then we halve the total integration interval and perform the same process on the two halves. Note that the "global" error needs to be halved for these new subintervals to preserve the global error when they are summed for the entire integration. This procedure is repeated until the desired accuracy is reached upon which we accept the integration for that subinterval, add it to the total, and move on to the next subinterval.

Although not immediately obvious this problem is best suited to a recursive function or subroutine whereby each successive call halves the subinterval and the "global" error that we check against. The recursion is terminated once the error estimate of Equation (5.10) becomes sufficiently small. Comprehending the logic of recursive formulas can be rather difficult, however, often the best way to understand them is to visualize some simple output.

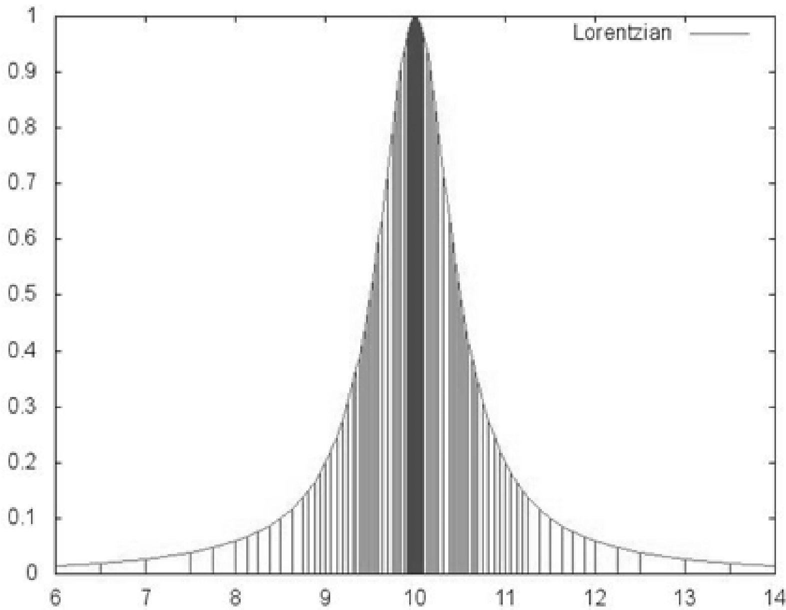

FIGURE 5.5: Adaptive numerical integration of the Lorentzian line-shape function.

The program a*daptiveQuadrature.cpp* performs this recursive action using the trapezoidal rule as default with the improvement technique described above. Figure 5.5 shows the result of applying this program to Equation (5.8) with $\lambda_0 = 10$, $I_0 = 1$, $\Gamma = 1$, and with integration limits of $a = 6$ and $b = 14$. The global error was selected to be 0.01% of the integration. Here we can see that the adaptive quadrature has done its job; in the flat regions away from the peak the strips are wider, whereas the strips covering the peak are much narrower. In order to lend clarity to the figure, the impulse lines plotted belong to the limits of the integration where the segmentation of that strip was accepted; the strips used in the actual calculation of

the integration are half the size of the ones shown. To visualize the data in such a manner the limits of the subintervals used were stored along with their function evaluations and plotted as impulse lines on the same figure as the function using "gnu plot".

As alluded to earlier the fact that the Lorentzian line-shape function can be reformed into a standard integral using a simple substitution. That substitution is

$$x = \frac{2(\lambda - \lambda_0)}{\Gamma} \tag{5.11}$$

which gives the integration of the (normalized) line-function the following form

$$\frac{1}{I_0}\int_a^b I(\lambda)\,d\lambda = \frac{\Gamma}{2}\int_c^d \frac{1}{1+x^2}\,dx \tag{5.12}$$

where c and d are the adjusted integral limits after the substitution of Equation (5.11). Equation (5.12) is a standard integral that has the following exact solution

$$\int \frac{1}{1+x^2}\,dx = \arctan(x)\,. \tag{5.13}$$

This analytical solution allows us to check the validity of our numerical solution and that it satisfies the requirement that the global error is at or less than 0.01%.

There is a fly-in-the-ointment here specifically about the function used to illustrate adaptive quadrature using the trapezoidal rule. If you apply just the composite trapezoidal rule to the Lorentzian line function you can achieve a similar error to the default adaptive scheme in ***fewer*** function evaluations. The reason is to do with the shape of the Lorentzian line function at the integration limits chosen; we refer you back to Section 5.2.1 and the Euler–Maclaurin formula.

5.2.3 Multidimensional Integration

Multidimensional integrations pop up often in physics and generally require much more effort to solve than the one-dimensional case. Take for instance a two-dimensional integral of the form

$$I = \int_a^b \int_c^d f(x,y)\,dxdy \tag{5.14}$$

where $f(x,y)$ is a two-dimensional function, and x and y have their usual Cartesian meaning. Here the integration limits a, b, c, and d specify a region in the x-y plane. With one-dimensional integration, we are finding the *area* between the function curve and the x-axis bounded by the given limits. Similarly, two-dimensional integration finds the *volume* between the function *surface* and the x-y plane bounded by an area or region. In three dimensions, the integration finds a four-dimensional space bounded by a three-dimensional surface. This increase in dimensionality can continue *ad infinitum* (or *ad nauseam* depending on your philosophical bent) in a mathematical sense but typically stops at three when considering most physical phenomena.

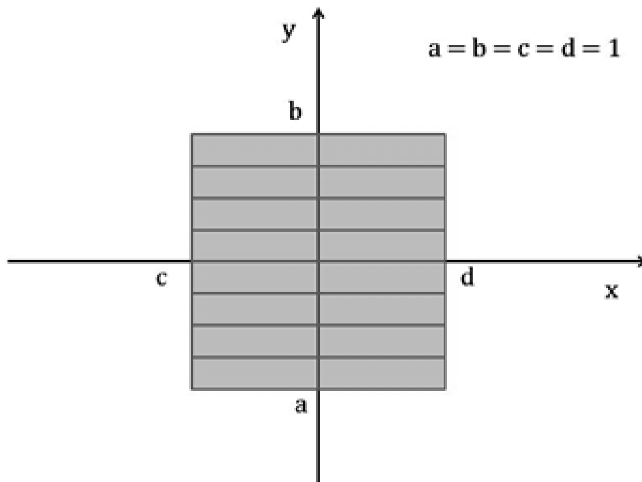

FIGURE 5.6: Square region split into strips running parallel to the x-axis. Here we do the x integration first.

The general strategy in solving a two-dimensional integration numerically is to split it into strips along one of the dimensions and treat each strip as a one-dimensional integration along the other dimension. The total volume of the integral is found by adding together the contributions from each strip. Figure 5.6 illustrates this

point. Here we have a region in the x-y plane that is bounded by the unit square, centered at the origin; the integration limits of Equation (5.14) are constants. The region has been broken into strips running parallel to the x direction. The function axis is pointing out from the plane of the page, and the function itself will form some surface either above, below, or cutting through the plane of the page (the x-y plane).

Mathematically, we have split the two-dimensional integration into two, nested, one-dimensional integrals such that

$$I = \int_a^b F(y)\,dy \tag{5.15}$$

where

$$F(y) = \int_c^d f(x,y)\,dx \,. \tag{5.16}$$

Of course, we can always reverse the order of the integration so that the y variable is integrated first. This would be equivalent to having strips running parallel to the y-direction of the square. When it comes to writing code to perform two-dimensional integration, we can go two ways; (1) have a function with nested loops or (2) have two separate functions to perform the integration in each dimension. As we already have developed classes to deal with one-dimensional integration the second of these choices is easier.

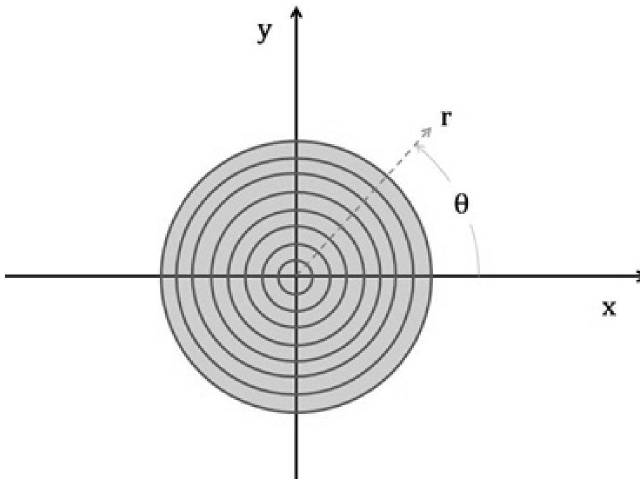

FIGURE 5.7: Segmentation of the (unit) circle region in polar coordinates.

In many real physical systems, the region may not be a square but some other more complicated shape, where the limits of the integral in one dimension are a function of the limits in the other. Take for instance the unit circle located at the origin. This has the equation

$$y = \sqrt{1 - x^2} \tag{5.17}$$

If we integrate over the upper right quadrant of the circle, we can see that the limits in the x integration have to adjust depending on the y value for which we are calculating. However, like many problems in physics we can take advantage of the symmetry of the system and a change to polar coordinates yields constant integration limits, as illustrated in Figure 5.7. The strips become concentric rings centered on the origin. It is likely this change of coordinates will make the integrand function more complex but why should we be concerned? We're performing numerical integration in the first place because the problem was too difficult/impossible to solve analytically.

EXERCISES

5.1. Compare the effort required to find a numerical approximation of the integration

$$\int_0^1 f(x)\,dx = \int_0^1 x(1-x)\,dx$$

to an accuracy of 5 significant figures using the various methods, we have developed in this chapter.

5.2. The period of a pendulum in confined to a single plane without damping has the following formula

$$T = 4\sqrt{\frac{l}{2g}}\int_0^{\theta_0} \frac{d\theta}{\sqrt{\cos\theta - \cos\theta_0}},$$

where l is the length of the pendulum, g is gravitational acceleration, θ is the angle between the pendulum and the vertical, and θ_0 is the initial angle of release. Calculate this integral numerically for various initial angles of release and thus establish what is meant by the small-angle approximation.

5.3. Write a program that performs integration over two dimensions using a method of your choice. The adventurous among you might like to try the adaptive quadrature in two dimensions.

5.4. What constant can you approximate by numerically integrating over the unit quarter circle? Find its value to 8 significant figures.

5.5. Consider a unit square region, centered on the origin, containing a uniform distribution of charge, ρ. The electrostatic potential at a point (x_m, y_m) outside this region is found by integrating over the charged region such that

$$\varphi(x_m, y_m) = \frac{\rho}{4\pi\varepsilon_0}\int_{-1}^{1}\int_{-1}^{1}\frac{dxdy}{\sqrt{(x-x_m)^2+(y-y_m)^2}}.$$

By taking $\rho = 4\pi\varepsilon_0$ and numerically assessing the integral at different points outside the unit square attempt to plot contour lines of isometric potentials. Do you recover Coulomb's law at distances far from the charged region?

5.6. The charge distribution in the previous question does not have to be uniform across the region. Try out different charge distributions with dependencies on x and y to see how they affect the electrostatic potential surrounding the square region.

CHAPTER 6

ORDINARY DIFFERENTIAL EQUATIONS

Physics is mostly concerned with phenomena that are in flux, for instance, things that change either in time or space or both, and many of the laws of physics are most conveniently formulated in terms of differential equations; formulas that relate derivatives to functions. As an example, consider Newton's second law of motion for a particle of mass, m, in one-dimensional motion under a force field $F(x)$:

$$F(x) = m\frac{d^2x}{dt^2}. \tag{6.1}$$

This is a second-order differential equation as we are taking the second derivative of the displacement, x, called the dependent variable, with respect to time, t, called the independent variable. The force field $F(x)$ can be referred to as the derivative function.

Finding the numerical solution of differential equations is one of the most common tasks in computational physics as many of these equations become analytically insoluble (or at least difficult to solve) when you include realistic physical processes.

6.1 CLASSIFICATION OF DIFFERENTIAL EQUATIONS

6.1.1 Types of Differential Equations

Differential equations can be categorized into two major groups, ordinary differential equations (ODE) and partial differential equations (PDE). The difference between the two is that ODEs only have one independent variable (they still can have any number of dependent variables) and PDEs can have any number of independent variables as well as any number of dependent variables. Simple harmonic motion (SHM) in one dimension is an example of an ODE:

$$m\frac{d^2x}{dt^2} = -kx\,, \tag{6.2}$$

where m is the mass of the body in motion, k is the so-called spring constant, and x represents the displacement from some equilibrium position. Here time, t, is the only independent variable and x is the dependent variable that is a function of the independent variable, normally written as $x(t)$. Whereas, the wave equation, which consists of second-order derivatives in both space and time, that is, two independent variables, is an example of a PDE:

$$\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2}, \tag{6.3}$$

where c is the speed of the wave, and u represents some (scalar) property of the wave, for example, displacement, pressure, electric field strength, and so on; u is the dependant function, in this case, normally written as $u(x,t)$. Note the use of the partial derivative symbol, ∂, rather than the usual d.

We can further subdivide the groups into their order. The order refers to the highest derivative appearing in the equations. For example, Equation (6.2) is of order 2, as is Equation (6.3). Order 2 ODEs and PDEs occur frequently in physics. Note that we can separate a second-order ODE into a pair of coupled, first-order ODEs should we so wish but more on that later.

Next, we can classify a differential equation as being linear or non-linear. In a linear ODE, the dependent variable and its

derivatives only appear to the first power and are not cross multiplied. Note that the independent variable can be to any power. For instance

$$f'' = -f + x^3 \tag{6.4}$$

is linear while

$$f'' = -f^2 + x^3 \tag{6.5}$$

and

$$f''f' = -f + x^3 \tag{6.6}$$

are non-linear as they contain the terms f^2 and $f''f'$ respectively. Here I have used the notation that

$$f' = f'(x) \equiv \frac{df(x)}{dx}, \tag{6.7}$$

that is, a dashed derivative is one taken with respect to a spatial variable, in this case, x. A dotted derivative is one taken with respect to a temporal variable, usually the time t, thus

$$\dot{f} = \dot{f}(t) \equiv \frac{df(t)}{dt}. \tag{6.8}$$

This is a generally accepted notation convention within physics, mathematics, and other sciences.

A further classification can be made to distinguish between homogeneous and nonhomogeneous differential equations. A homogeneous equation contains terms that include either the dependant variable or its derivatives, but no other function of the independent variable. The differential equation of the simple harmonic oscillator, Equation (6.2), is an example of a homogeneous equation. Adding a time-dependent driving force, $F(t)$, to this equation gives the nonhomogeneous equation

$$m\frac{d^2x(t)}{dt^2} + kx(t) = F(t), \tag{6.9}$$

as we now have a function of the independent variable, t, on the right-hand side. The actual form of the driving force is unimportant to this discussion but will be particular to the system being described by the differential equation. Note that Equations (6.4)–(6.6) are all

nonhomogeneous due to the addition of the independent variable term, x, on the right-hand side.

6.1.2 Types of Solution and Initial Conditions

When solving differential equations, we draw a difference between the general solution and a particular solution. The general solution refers to all the functions that fit the differential equation, whereas the particular solution is defined by some initial conditions or values. Take for instance Newton's law of cooling (or heating) that states that the rate of change of temperature of a body is in proportion to the temperature difference between it and that of the ambient. It can be written in the form

$$\dot{y} = -ky, \tag{6.10}$$

where y is the temperature *difference* between the body and the (constant) ambient, and k is some constant of proportionality (related to the surface area of the body, the material the body is made from, and so on). The minus sign represents the physics that hot bodies cool and cold bodies warm. This has the general solution

$$y(t) = y_0 e^{-kt}, \tag{6.11}$$

where y_0 is the initial temperature difference, that is, the temperature difference at $t = 0$. Given an initial temperature difference, we could then determine a particular solution for any value of k. Note that y_0 is called an integrating constant that is a mathematical concept related to some initial condition of the system, whereas k is a constant purely related to the physics of the system; the two should not be confused.

Equation (6.10) is an ODE of the first order and only has one integrating constant. As such we only needed to know one initial condition to determine a particular solution, namely the initial temperature difference. In the general case, an n-ordered ODE will produce n integrating constants and we need as many initial conditions to find a particular solution. To convince yourself of this what initial conditions are required to determine a particular solution of the second-order ODE describing SHM?

These are known as initial value problems. Other integration constants may include boundary conditions, that is, the condition or value of the dependant function at or beyond the boundaries or edges of your modeled system. For instance, the physical properties of the potential barriers in a quantum well provide the boundary conditions for solving Schrodinger's wave equation for an electron trapped in the well. Essentially, the wave function and its derivatives decay to zero as it penetrates deeper into the bounding potential barriers.

6.2 SOLVING FIRST-ORDER ODES

6.2.1 Simple Euler Method

Leonhard Euler was an 18th century, Swiss-born mathematician, who we would describe today as a polymath. Euler worked in several areas including optics, astronomy, ship construction, and artillery but was most prolific in his work on mathematics. He contributed much to the fields of number theory, algebra, and calculus, and can be directly attributed to the modern standard usage of the symbols e, π, and i.

Consider again Equation (6.10) which is a linear, homogeneous ODE of the first order. We can write a first-ordered ODE more generally as

$$\dot{y}(t) = f(t, y), \tag{6.12}$$

where f is some function of the independent variable, t, and the dependent variable, y; the form of f determines the classification of the differential equation.

We could attempt to solve Equation (6.12) by taking the Taylor series expansion of the dependent variable about some initial position, t_0, such that

$$y(t) = y(t_0) + (t - t_0)\dot{y}(t_0) + \frac{(t - t_0)^2}{2!}\ddot{y}(t_0) + \cdots. \tag{6.13}$$

Since we know the form of the first-ordered derivative from Equation (6.12) we could calculate the higher-ordered derivatives

by using the partial differentiation of f. However, this soon becomes untenable for all but the simplest expressions for f, and in which cases Equation (6.12) can most likely be solved analytically. We can get rid of those troublesome higher-order derivatives by truncating Equation (6.13) to the first two terms only, leaving

$$y(t) \approx y(t_0) + (t - t_0)\dot{y}(t_0). \tag{6.14}$$

Note that we could have arrived at Equation (6.14) by considering an approximation to the gradient of the dependent function at the initial position

$$\dot{y}(t_0) \approx \frac{y(t) - y(t_0)}{t - t_0}. \tag{6.15}$$

If we now say that $h = (t - t_0)$ where h is a small step, we may now conveniently write Equation (6.14) as an equality

$$y(t_0 + h) = y(t_0) + hf(t_0, y(t_0)) = y_0 + hf_0, \tag{6.16}$$

where we have substituted in the function, f, for the derivative, and used the notation that $y_0 \equiv y(t_0)$ and $f_0 \equiv f(t_0, y_0)$.

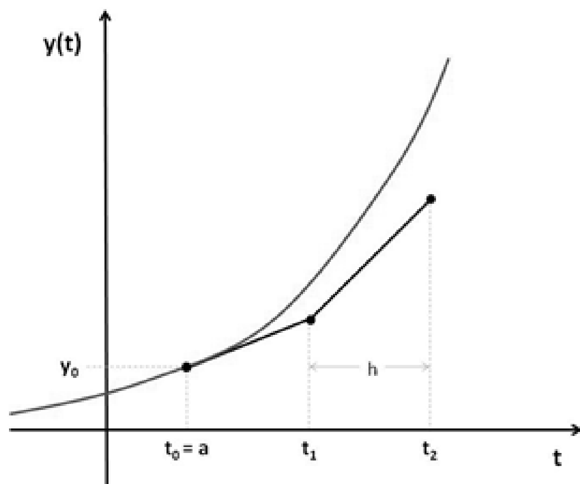

FIGURE 6.1: Sketch of the simple Euler method. We only know the first value of y exactly; the integrated values are approximations to y.

Equation (6.16) is the simple Euler method or Euler's forward approximation. Interpreting this method, we can see that given a

starting position, y_0, we can step to the next position using the derivative f_0 at the start of the step. We can generalize this to the nth step giving the recursion formula

$$y_{n+1} = y_n + hf_n, \tag{6.17}$$

where we define, $t_n \equiv t_0 + nh$, $y_n \equiv y(t_n)$, and $f_n \equiv f(t_n, y_n)$, with $n = 1,2,3\cdots$. Note that this stepping action can be referred to as integrating the solution; we are solving a differential equation and are therefore performing an integration. Figure 6.1 illustrates the simple Euler method in action.

Although we could step *ad infinitum* we typically wish to find a value for y at some predefined value for t, in other words, we have an interval $[a,b]$ over which we wish to step from a to b. The most straightforward way of doing this is to split the interval into N steps of equal size, h, such that $h = (b-a)/N$, and move the solution along one step at a time using Equation (6.17). We can check the accuracy of the method by repeating the integration using smaller and smaller step sizes and seeing if we converge on a solution. Though perhaps a more stringent test is that once we reached our desired value b we integrate *back* toward a and compare our integrated approximation to the initial value for y with which we started.

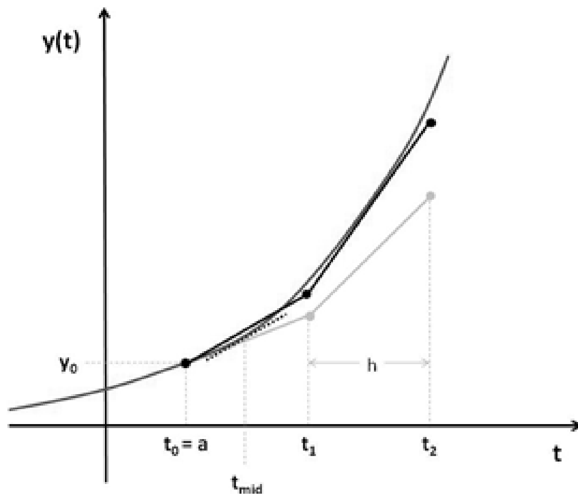

FIGURE 6.2: Sketch of the modified Euler method. The gradient for the entire step is estimated from the derivative at the mid-point. The simple Euler method is shown in grey for comparison.

As defined in Equation (6.17) using a truncated Taylor series expansion behavior of the error can be determined in our approximation to the solution. For a *single step* of the simple Euler method, we know that the approximation given by Equation (6.14) will have an upper bound on the "local" error of $\mathcal{O}(h^2)$ as we kept the first two terms of the Taylor series only. To get from our initial position a to our desired position b we must perform N such steps. Thus, the overall upper bound on the error at b will be given by $N \times \mathcal{O}(h^2)$. As N is inversely proportional to the step size h, we can then estimate the "global" error in the simple Euler method as $\mathcal{O}(h)$. In other words, if you halve the step size (and thus take twice as many steps) you should halve the error in the approximation to the solution at the destination b.

The program *eulerForward.cpp* performs the simple Euler method on the differential equation

$$y' = -xy, \tag{6.18}$$

with the initial condition that $y_0 = 1$, and over the interval $x = [0,2]$. The analytical solution to Equation (6.18) with the given initial condition is

$$y = e^{-0.5x^2}. \tag{6.19}$$

The program uses the `Euler` class found in *ODESolvers.h* and which inherits from the base class `ODESolver`. You should familiarize yourself with the implementation of the `Euler` class in *ODESolvers.cpp* and satisfy yourself that the `solve` and `fullSolve` member functions perform the forward Euler method as expressed by Equation 6.17. Notice that the state of the system, the coordinate pairs of the independent and dependent variables, is encoded by the `state` data structure defined in State.h and implemented in State.cpp. The design of this data structure is discussed when we look at solving ODEs of order 2 in a later section of this chapter. Compiling and running this program should be observed using $\mathcal{O}(h)$ error behavior that is to be predicted.

6.2.2 Modified and Improved Euler Methods

Although the simple Euler method provides an instructive means of introducing the topic of numerically solving differential equations

it should not be used in any serious attempt to find a solution due to its lack of accuracy. The major issue with the simple Euler method is that it assumes the derivative at the start of a step remains constant over that step, see Figure 6.1. This asymmetrical treatment of the step is bound to lead to large inaccuracies of the approximated solution. It would be better if we could use some sort of averaged value to estimate the derivative across the whole step.

The modified Euler method approximates the solution by using the derivative at the mid-point of the step to advance the integration. Obviously, we do not know the value of the derivative at the mid-point, but we can approximate it using the simple Euler method with half the step size such that

$$t_{mid} = t_{n+1/2} = t_n + \frac{h}{2} \tag{6.20}$$

and

$$y_{mid} = y\left(t_{mid}\right) = y_{n+1/2} = y_n + \frac{h}{2} f_n, \tag{6.21}$$

where n is our previous step for which we have values. We can now use the value for y_{mid} to estimate the derivative at the mid-point of the step and thus advance the solution across the whole step as follows

$$y_{n+1} = y_n + hf\left(t_{mid}, y_{mid}\right). \tag{6.22}$$

Equation (6.22) is the modified Euler method and is illustrated in Figure 6.2. From a cursory look at the figure, you can see that the modified Euler method appears to do a much better job at approximating the solution than the simple Euler method. Note that the function sketched is somewhat arbitrary, but it should be able to show that the modified Euler method is better by writing a program.

Another way of obtaining an average value that best approximates the derivative across the step is to take a mean of the derivative at the start of the step with the derivative at the end of the step. Again, we use the simple Euler method but this time to estimate the value of the derivative at the *end* of the step. Using this estimate with the derivative at the start of the step, which we have previously computed, we can take the mean of these two values to advance the integration of one whole step. Mathematically written as

$$y_{n+1} = y_n + \frac{h}{2}\left[f_n + f\left(t_n + h, y_n + hf_n\right)\right]. \tag{6.23}$$

Equation (6.23) is the improved Euler method and is illustrated in Figure 6.3. A cursory study of the figure suggests that the improved Euler method is better than the simple Euler method. But which is better between the improved Euler method or the modified Euler method?

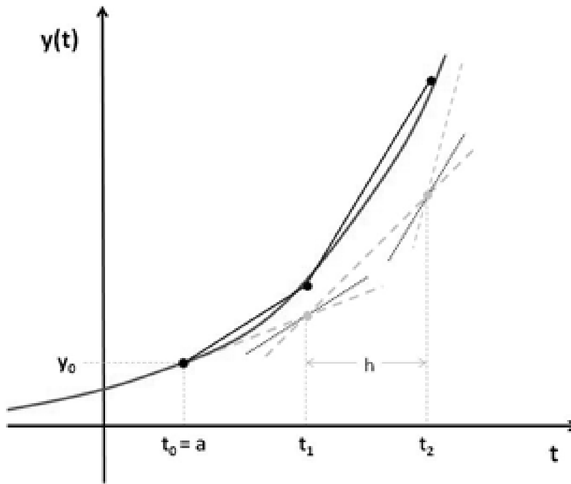

FIGURE 6.3: Sketch of the improved Euler method. The simple Euler method is used to estimate the derivative at the end of the step, which combined with the derivative at the start of the step gives a mean for the entire step.

Use the simple Euler method program provided to write a code for the modified, and improved Euler methods; Equations (6.22) and (6.23), respectively. Using these programs one could determine how the error behaves with step size for these two methods?

It was the German Mathematician Karl Heun who first developed the modified Euler and improved Euler methods, which in part helped develop the more accurate Runge–Kutta methods that will be discussed subsequently.

6.2.3 The Runge–Kutta Method

Carl Runge and the Polish-born Martin Kutta were both German mathematicians and physicists who lived and worked around

the latter part of the 19th century and into the first half of the 20th century. In 1901, they co-developed the Runge–Kutta method(s), used to solve ODE numerically.

The Runge–Kutta methods are characterized by expressing the numerical approximation in terms of the derivative function evaluated at intermediary points between step values. Euler's methods can actually be classed as low-ordered general Runge–Kutta methods; the Euler method being a one-step Runge–Kutta method, and the modified, and improved methods are both two-step Runge–Kutta methods. The popularity of the Runge–Kutta methods in numerically solving ODEs is due in part to their (relative) ease of implementation within computer programs, and the accuracy they achieve. Of the most popular devised is the fourth-ordered Runge–Kutta (RK4), or simply *the* Runge–Kutta method, defined as

$$y_{n+1} = y_n + \frac{h}{6}\left(k_0 + 2k_1 + 2k_2 + k_3\right), \tag{6.24}$$

where

$$\begin{aligned} k_0 &= f\left(t_n, y_n\right), \\ k_1 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_0\right), \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right), \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + hk_2\right). \end{aligned} \tag{6.25}$$

The derivation of Equations (6.24) and (6.25) is a little tricky but involves considering a general form for Euler's methods such that

$$y_{n+1} = y_n + h\left[\alpha f_n + \beta f\left(t_n + \gamma h, y_n + \delta h f_n\right)\right] \tag{6.26}$$

and choosing the coefficients $(\alpha, \beta, \gamma, \delta)$ such that they agree with the Taylor series expansion of the term involving β, up to h^4. Indeed, the modified and improved Euler methods can be found in this way by matching terms up to h^2. Feel free to have a go at deriving these equations yourselves using this method but be warned that taking a

Taylor series expansion of a function of two variables gets somewhat complex for anything more than the degree one terms.

A slightly easier but less general way of deriving Equations (6.24) and (6.25) is to consider the direct integration of Equation (6.12) for the special case that derivative function is a function of the independent variable alone, that is, $f(t)$. We can then write for a single step

$$y(t_n + h) = y(t_n) + \int_{t_n}^{t_n+h} f(t)\,dt, \tag{6.27}$$

and by solving the integration term by Simpson's rule we obtain the results of Equations (6.24) and (6.25).

Interpreting the RK4 approximation, Equation (6.24), we see that the next value in the integration is calculated as the present value plus the weighted average of four increments. The increments are determined from estimates of the slope at intermediary points on the step specified by the derivative function f, multiplied by the step size h. These increments can be described as follows:

- k_0 is the estimate of the slope at the beginning of the step;
- k_1 is the estimate of the slope at the midpoint, using k_0;
- k_2 is the estimate of the slope at the midpoint, but now using k_1; and
- k_3 is the estimate of the slope at the end of the interval, using k_2.

In averaging the four increments, greater weight is given to the increments at the midpoint reflecting the fact that the function's slope is better approximated by the tangent to the curve at the midpoint of the interval rather than its bounds.

The RK4 method is a fourth-order method, meaning that the local error behaves as $\mathcal{O}(h^5)$, while the global error behaves as $\mathcal{O}(h^4)$. This means that halving the step size will reduce the overall error by a factor of 16, hence why the method is so popular.

The class `RK4` found in the ODESolvers module performs the fourth-ordered Runge–Kutta method. Write a program that uses

this class to solve the first-ordered differential equation defined by Equation (6.18) found earlier in this chapter.

Confirm that the Runge–Kutta method provides an accuracy of $\mathcal{O}\left(h^4\right)$ and is superior to the Euler methods we discussed earlier in this chapter.

6.2.4 Adaptive Runge–Kutta

Generally, we wish to find the numerical solution to an ODE to some predefined (global) error or tolerance. Using a fixed step size, we are somewhat constrained to use one sufficiently small across the entire interval to provide the required local accuracy at each step of the integration. If the nature of the solution changes across that interval, that is, becomes increasingly rapid in its variation with the independent variable, then we would waste considerable effort over the "flat" regions of the solution. Therefore, we would like to be able to change the size of the steps taken in the numerical approximation in accordance with the local nature of the solution, that is, allow them to adapt.

By far the most straightforward way to do this is to perform a single step of the integration with step sizes h and $h/2$ and compare the result immediately. More precisely, we perform a single step with step size h, then halve its size and perform two steps with the new step size to reach the same point in the solution. We can estimate the error in the numerical approximation by computing the difference between our two solutions. By comparing this difference to our predefined tolerance, we can either accept the step if the difference is smaller, otherwise, we use the halved step size to repeat the process. However, this is *not* the whole picture. Here we have *only* taken account of the solution starting in a flat region and advancing into a rapidly changing one.

If we start in a rapidly changing region then we merrily halve our step size until it produces a solution that is within the prescribed accuracy tolerance, and we advance with that small step. If the solution now flattens then we simply maintain that small step as it will produce a solution that is (very much) within the accuracy tolerance. This is *not* what we were after; we want a step size that adapts to the local nature of the solution, that is, can increase as well as decrease.

The answer to this issue is to assume that when we accept a step, that is, we are within the accuracy tolerance, the step size is too small and should be increased for the next step; maybe we should double the step size to $2h$, is there a problem with this strategy?

By using an integration method of known error order, we can eliminate the leading error term in the integration step using the two solutions. For example, the *fourth-ordered* Runge–Kutta method reduces the error in the solution by a factor of 16 when we halve the step size. To eliminate the leading error term for this Runge–Kutta method we compute the integration for a particular (accepted) step as $\hat{y} = \left(16y_2 - y_1\right)/15$.

This is like the method used in Chapter 5 to develop an adaptive quadrature using the knowledge of how the error in the trapezoidal rule behaves with strip width. Generally, this method of manipulating the solution based on error behavior is referred to as Richardson's extrapolation that we explore further in the advanced section of this book.

Using the method outlined above write a program that uses the fourth-ordered Runge–Kutta algorithm to adaptively integrate a differential function of your choice. My advice would be to use a simple differential equation that can be solved analytically for comparison to your adaptive routine. You should confirm that your routine is adapting to the local nature of the differential.

6.3 SOLVING SECOND-ORDERED ODES

6.3.1 Coupled 1st Order ODEs

It has been noted before that second-order ODEs occur most frequently in physics as they model many real physical systems. In general, we write a second-order ODE as

$$\ddot{y} = f\left(t, y, \dot{y}\right). \tag{6.27}$$

Note that the function f has three variables namely the independent variable, the dependent variable, and the first derivative of the dependent variable.

Although methods exist to solve higher-ordered differential equations, for example, finite difference method, it is far simpler to reduce the equation into a set of coupled first-order differential equations; the term coupled will become apparent shortly. We can do this by introducing secondary dependent functions such that $y_1 = y$ and $y_2 = \dot{y}$ and thus we can rewrite Equation (6.27) as a pair of coupled, first-order ODEs:

$$\dot{y}_1 = y_2;$$
$$\dot{y}_2 = f\left(t, y_1, y_2\right). \tag{6.28}$$

They are coupled because the rate of change of variable y_1 is dependent on the variable y_2, and the rate of change of variable y_2 is dependent on the variable y_1 contained in the function f. However, if we define $f_1 \equiv y_2$ and $f_2 \equiv f\left(t, y_1, y_2\right)$ then Equations (6.28) can be rewritten in vector form

$$\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \end{bmatrix} = \begin{bmatrix} y_2 \\ f\left(t, y_1, y_2\right) \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}, \tag{6.29}$$

or

$$\underline{\dot{y}} = \underline{f}, \tag{6.30}$$

where $\underline{\dot{y}}$ and $\underline{f}$ represent two-component vectors. Comparison of Equation (6.30) with Equation (6.18) shows that the problem of solving second-ordered ODEs is not primarily different from the first-ordered ODEs for which we have been developing solutions, only that now we have extra components.

To illustrate this point, we can write Equation (6.1), which describes Newton's second law of motion, as a pair of coupled first-order differential equations by introducing momentum as a secondary dependent variable. The momentum of a body of mass m in one dimension is defined as

$$p(t) \equiv mv(t) = m\dot{x}, \tag{6.31}$$

where $v(t)$ is the velocity of the body at time t, and x represents the (one dimensional) displacement of the body in some coordinate system. Thus Equation (6.1) can be rewritten in terms of the momentum as follows

$$\dot{x} = \frac{p}{m}, \tag{6.32}$$

$$\dot{p} = F\left(t, x, p / m\right), \tag{6.33}$$

where time, position (displacement), and velocity have been included in the force term for completeness (here we assume that mass is a constant of motion). We can check these equations make sense by assessing the situation when no net force acts on the body, that is, $F = 0$. This implies a constant momentum that in turn implies an unchanging velocity, and thus we recover Newton's first law of motion. Just to restate and reinforce our nomenclature, here we call time t the independent variable, and the position x and the velocity p / m (or just the momentum p) are the (coupled) dependent variables.

6.3.2 Oscillatory Motion

At the beginning of this chapter, we briefly discussed the second-order differential equation describing SHM. Using Equations (6.32) and (6.33), we can rewrite this as a pair of coupled first-order ODEs

$$\dot{x} = \frac{p}{m} \tag{6.34}$$

and

$$\dot{p} = -kx. \tag{6.35}$$

We can make life easier for ourselves by rewriting these equations in terms of velocity, v, instead of momentum, p, such that

$$\dot{x} = v \tag{6.36}$$

and

$$\dot{v} = -\frac{k}{m}x. \tag{6.37}$$

We can make this change as, in this case, the mass is assumed to be a constant of motion and so we are not changing the physics of the system only our notation. Note that in this form Equation (6.37) has no multiplicative constant in front of the derivative. This is the general strategy you should employ when solving differential equations (numerically or analytically), ensuring all physical constants appear with the derivative function where possible.

In the source file, *ODESolvers.cpp*, one will find two member functions of the `ODESolver` base class, `deriv` and `deriv_B`, that implement Equation 6.30. In our current specific discussion of one-dimensional SHM the `deriv` member function encodes Equation 6.36, and the `deriv_B` member function encodes Equation 6.37, with an appropriately defined differential function. These two member functions are used by all derived classes of the `ODESolver` base class. They are written in such a way as to avoid conditional branching when using these classes to solve differential equations of a different order (though only order 1 or order 2 differential equations are supported). They are linked to the design of the `state` data structure, and how the dependent variable(s), and derivative(s) (for second-ordered differentials) are represented. For a first-ordered differential equation, the C++ vector `y` in the `state` structure represents the dependent variable for each dimension in the system. For a second-ordered differential equation, the vector `y` can be thought of containing consecutive pairs of values, the dependent variable, and its derivative for each dimension in the problem. In this way, the dependant variable is found at even indices in `y` and the corresponding derivative is found at the corresponding odd index, for example, `y[2]` is the dependent variable of the second dimension and `y[3]` is the corresponding derivative variable of the second dimension (assuming the problem has at least two dimensions).

After writing a program that uses the `Euler` class to integrate the differential equation for SHM (Equation 6.37), and using the initial conditions $x(0)=1$, $v(0)=0$, with $k/m=1$, and 100 steps, we obtain the result plotted in Figure 6.4 up to a time of $t=15$ s.

As the world's energy demand has yet to be satisfied by a mass-on-a-spring system what has gone wrong? Have we made a mistake with the physics or the implementation of the simple Euler method? To answer these questions let's examine the analytical solution of the ODE describing SHM. From your A-level or equivalent physics course, you will know that the solution of the spring equation for displacement is a sinusoidal function in time. The phase of that solution, that is, whether it is a sine function, cosine function, or somewhere in-between, is dependent on the initial conditions. In the case above we obtain the particular solution

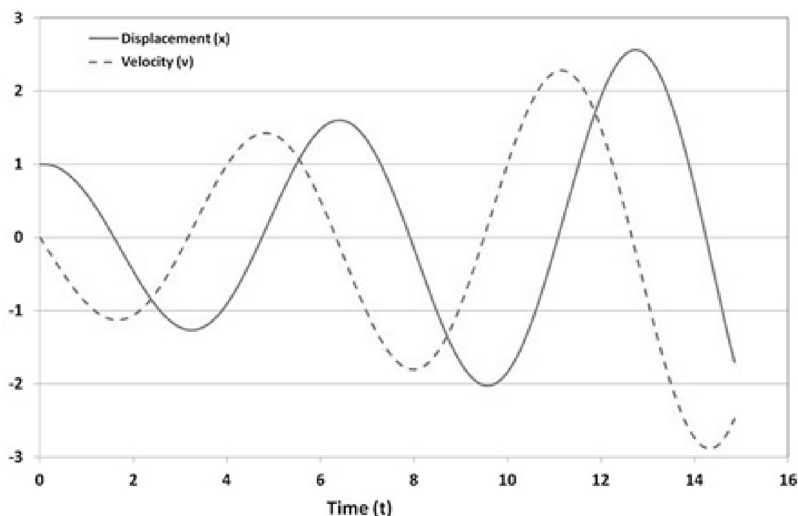

FIGURE 6.4: Numerical solution of SHM using the simple Euler method.

$$x = \cos(\omega t) \tag{6.38}$$

where ω is the angular frequency of the oscillations and is given by

$$\omega = \sqrt{\frac{k}{m}}. \tag{6.39}$$

Thus, we know that the solution is a cosine function with a (time) period of 2π. This is encouraging as our numerical solution, despite increasing in amplitude, has these properties, so it is a safe bet that the physics and the implementation are sound.

From our previous discussions of the Euler method, it is obvious that the instability of the numerical solution is down to the truncation error of the Taylor series. We could of course reduce the step size to help with the stability of the numerical solution but that would be wasting computational effort; we have already developed more accurate numerical solvers. Write a program that uses the `RK4` class to integrate the SHM equation. As an aside, the results plotted in Figure 6.4 should make a strong case as to why the simple Euler method should not be used as a serious attempt to solve ODEs

describing real, physical systems. Indeed, for our next discussion, we require something with a bit more accuracy.

Once you have a program up and running you should be able to reproduce the results plotted in Figure 6.5. Here we start with the same initial conditions and parameters as before but have allowed the integration to run up to time $t = 60$ s and have changed the number of steps N to 600, that is, a step length of 0.1 s.

Notice that the solution is stable for (at least) the first nine periods of oscillation. To stringently test the stability of the Runge–Kutta solution, we should allow the integration to run over several thousand periods of oscillation, maintaining the same step length, and monitor the amplitude of the oscillations produced. Even more stringently, we should integrate backward from the endpoint to the start and compare the values of the displacement and velocity to the initial conditions. However, the range of the stability shown in Figure 6.5 will be sufficient for the following discussion.

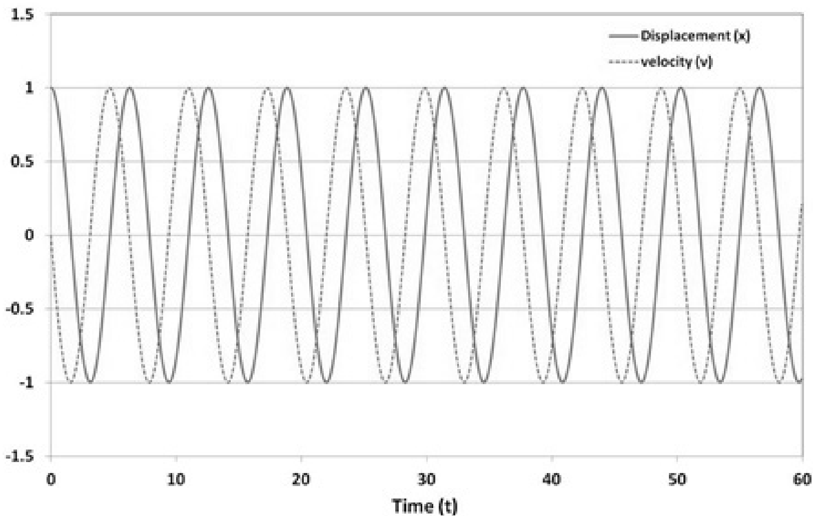

FIGURE 6.5: Fixed step Runge–Kutta solution of SHM, encompassing nine periods of oscillation.

In real physical systems, oscillatory motion is usually damped. We know this because after we put say, a mass on a spring in motion it will lose the initial amplitude it was given and eventually come to

rest. This is due to mainly resistive losses as the mass moves through the air. We can model this damping effect by assuming the drag force acting on the oscillating mass is proportional and opposite to the velocity of the mass. The second-order ODE describing damped oscillatory motion now becomes

$$m\ddot{x} = -kx - D\dot{x}, \tag{6.40}$$

where D is the constant of proportionality for the drag force, and we have taken the mass term to the left-hand side for clarity. Note that Equation (6.40) remains a linear, homogeneous ODE of the second order. Separating Equation (6.40) into a pair of coupled first-order ODEs is straightforward. All we must do is modify Equation (6.37) to include the additional, drag force term such that

$$m\dot{v} = -kx - Dv \tag{6.41}$$

and Equation (6.36) remains unchanged. This appears deceptively simple and studying physics you will probably be developing a healthy mistrust of anything that appears simple, but in this case, it is that straightforward.

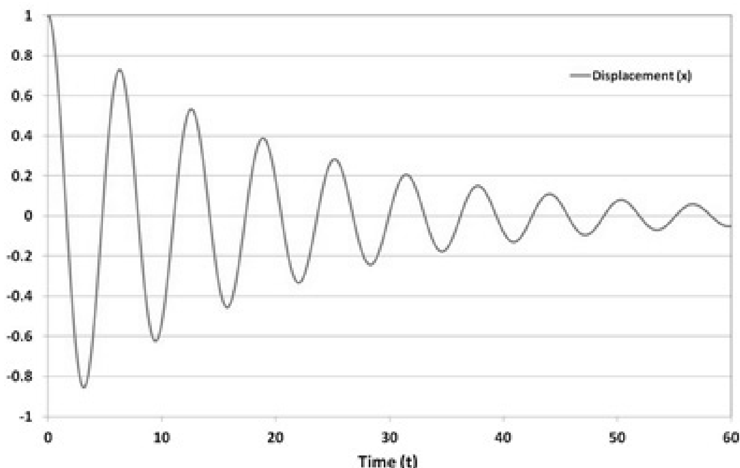

FIGURE 6.6: Damped oscillatory motion integrated using a fixed step Runge–Kutta method.

After making the appropriate modification to the derivative function in your code you should be able to reproduce the results plotted in Figure 6.6. Here I have taken the parameters to be $k/m = 1$ and $D/m = 0.1$. We could now modify the model of resistive drag with relative ease, the numerical algorithm simply processes the numbers.

As a last discussion to this section let us consider driven oscillatory motion. At the beginning of this chapter, we identified a nonhomogeneous ODE as one having a function of the independent variable extra to the dependent variable and its derivatives. The driven oscillatory motion is used as an example in Equation (6.9) and it is repeated here with an addition of the drag term

$$m\dot{v} = -kx - Dv + F(t), \tag{6.42}$$

where $F(t)$ is the driving force. The form of the driving force will be dependent on the physics of the system Equation (6.42) describes. For instance, a child being pushed on a swing (pendulum system rather than a mass-on-a-spring) will have a driving force that would be well suited to be modeled by an impulse acting at a particular point in the oscillation. Whereas the driving force describing a car's suspension system as it travels over a cobbled street, say, could be modeled by some sort of sinusoidal function.

For the sake of this discussion and simplicity, let us assume the driving force is a straightforward sine function, thus

$$F(t) = A\sin(\omega_0 t + \varphi), \tag{6.43}$$

where A is the amplitude or maximum force supplied by F, ω_0 is the angular frequency of the driving force (the subscript distinguishes it from the angular frequency of the solution), and ϕ is a phase shift added for generality.

Make further modifications to your derivative function to include the driving force term as described in Equation (6.43). For now, assume that the phase shift is zero. Keeping everything else the same and using the parameters $A = 0.2$ and $\omega_0 = 1$ you should be able to reproduce the results plotted in Figure 6.7.

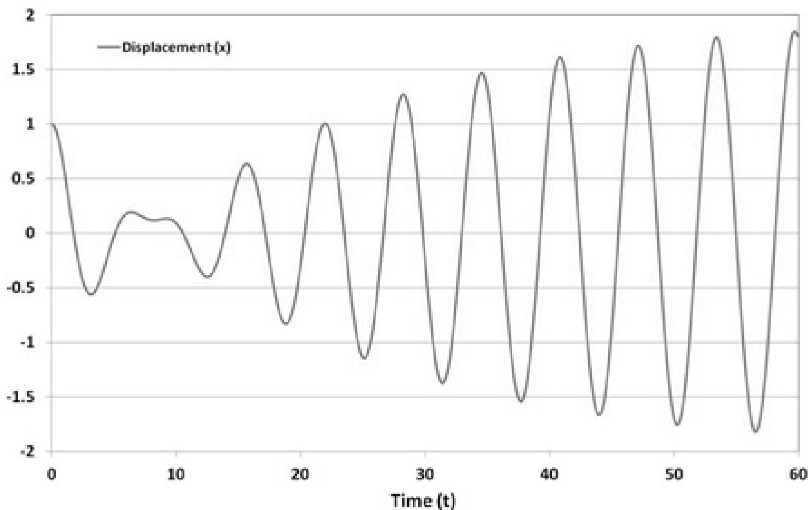

FIGURE 6.7: Driven oscillator. There are two regions of the solution: the transient region, and the steady-state region.

Notice that in the solution of the driven oscillator there are two regions. First is the transient region where the nature of the solution changes with the independent variable. Then the steady-state region where the oscillations follow the form of the driving force. Figure 6.7 shows the case where we have a situation close to resonance; though not actually at resonance—even though the driving frequency equals the natural frequency, the damping term effects the frequency at which the oscillator shows a maximum response to the driving force (see Exercise 6).

6.3.3 More Than One Dimension

The code that we have developed to solve a second-ordered ODE by transforming them into a pair of coupled first-ordered ODEs have currently only considered motion in one-dimensional space. To solve for the motion, we required two dependent variables, namely the displacement and the velocity (or momentum). These methods can be extended to cover problems involving several *dependent* variables that may describe motion in three-dimensional space.

For example, if we were to describe the motion of the Earth in orbit about the Sun, which we know is planar, we would need the

Earth's x and y coordinates as well as it is velocities v_x and v_y, at a particular time. That is, we require four dependent variables to fully describe the Earth's orbit. For motion that is not planar, we would require six dependent variables to fully describe a bodies motion in four-dimensional space-time.

The `state` data structure and `ODESolver` classes can deal with any number of dimensions. To set up a multi-dimensional system for a first-ordered differential equation you provide a C++ vector containing the initial value of each dependent variable to the `state` constructor. For a second-ordered differential equation, you provide an additional vector containing the initial derivative values for each dimension in the system. The implementation of the `ODESolver` classes automatically handles the extra dimensions. As stated in the opening chapter, this is my design, and you should **NOT** take it as gospel. If you think you can redesign the code to make it more user friendly or perform better, then try it out. That's one of the beauties of programming and open-source software.

EXERCISES

6.1. One stringent accuracy test of a numerical integration scheme is to have it step backward from the value of the final step to the starting position and see how close we come to the initial value we supplied. Apply this test to the methods we have developed in this chapter and comment on the outcome for different step sizes.

6.2. Using Equation (6.18) and its solution (6.19) plot, on an appropriate graph, the error produced by the Euler methods and fixed step Runge–Kutta method for step sizes in the range $h = 0.1$ down to $h = 10^{-15}$. Comment on what you find.

6.3. Consider one-dimensional projectile motion with air resistance we can write

$$m\dot{v} = mg - Dv^2$$

where m is the mass of the projectile, v is its velocity, g is the acceleration due to gravity, and D is the drag coefficient. For a sphere of mass $m = 10^{-1}$ kg the drag coefficient was found to be $D = 10^{-3}$ kg/m. Using one of the numerical solvers we have developed find the terminal velocity of the sphere dropped close to the surface of the Earth. Does this agree with theory?

6.4. Modify your Runge–Kutta program for SHM without any damping or driving force terms to check the stability of the method over many periods (tens of thousands). How might you monitor the accuracy of the numerical solution?

6.5. Using your Runge–Kutta program for SHM with damping only, check for critical damping and assess when it occurs in terms of the relative values of k, m, and D. Does this agree with the theory of critical damping?

6.6. Using the numerical solvers at hand, how does damping affect the resonance phenomena (resonant frequency and maximum response of the oscillator) of driven oscillations? Does this agree with real observations?

6.7. Newton's gravitational force of attraction between two objects is given by,

$$\underline{F} = -\frac{GMm}{r^3}\underline{r}$$

where G is the universal gravitational constant, M and m are the masses of the two bodies, and r is their separation distance. Using either the fixed step or adaptive step Runge–Kutta method we've developed an attempt to compute the mass of the Sun knowing that Earth requires one year to make the orbit. The distance between the Earth and the Sun is one astronomical unit (1 AU). Assume Earth's orbit is circular, that there is no influence from any other galactic body, and that the coordinates of the Sun are fixed at the origin. Try to get your answer to within four significant figures of precision and check your result for accuracy.

CHAPTER 7

FOURIER ANALYSIS

Fourier analysis, also known as spectral analysis, is a powerful tool for the experimental scientist. It can help to establish a clear physical picture of an experimental system than just from the raw data on its own. Fourier analysis can also be used to help extract significant information from particularly noisy or complicated signal or waveform that may have otherwise been missed or lost. For instance, Fourier analysis can be used to: reconstruct a crystal structure from its X-ray diffraction pattern; determine the mass of ions exhibiting cyclotron motion in a magnetic field; reconstruct the 3D image from a series of X-ray images in computerized tomography scan; produce bandpass filters in electronic circuits; improve digital radio reception; clean up noisy digital images; and the list goes on. In general, all these techniques rely on finding the Fourier transform of the measured, raw data. To do this, we must first talk about how to represent or approximate a function using a Fourier series. As a starting point, let us return to the Taylor series expansion of a function and discuss its limitations.

As described in Chapter 2, the Taylor series expansion is a powerful tool when it comes to approximating functions. However, the truncated Taylor series expansion of the sine function can only approximate a function reasonably accurately about the (unique) point it was taken. While this may not cause a significant limitation to most continuous functions, periodic functions are not well suited to Taylor series expansions; the period is simply not considered.

In addition to this limitation with periodic functions the Taylor series expansion requires that a function and all its derivatives exist everywhere. In other words, the Taylor series expansion cannot be used to approximate functions with discontinuities (jumps) either in the function or in the derivatives of the function.

7.1 THE FOURIER SERIES

Jean Baptiste Joseph Fourier was a French mathematician and physicist born in Auxerre in 1768. He is best known for starting the investigation of the now eponymous Fourier series, that he applied to the then unsolved (general) problems of the propagation of heat and vibrations. It was Fourier who first pointed out that an arbitrary periodic function $f(t)$, with a period T, can be separated into a summation of simple trigonometric terms such that

$$f(t)=\frac{a_0}{2}+\sum_{n=1}^{\infty}\left(a_n\cos(n\omega_0 t)+b_n\sin(n\omega_0 t)\right), \tag{7.1}$$

where the a_n and b_n are the so-called Fourier coefficients, and $\omega_0=2\pi/T$ is the natural frequency of the function. Note that *every* periodic function has a natural frequency, but only harmonic oscillators behave as pure sinusoidal waves. Interpreting the Fourier series, we see that the function, which may represent some audio signal or EM radiation or whatever, is composed of the superposition of many harmonic tones of the natural frequency. A harmonic tone is a sinusoidal function with a period equal to an integer multiple of the natural frequency. The coefficients a_n and b_n thus providing a measure of the contribution to the signal from the cosine and sine harmonics, respectively. More precisely, the intensity or power at each harmonic frequency is proportional to $a_n^2+b_n^2$; this is referred to as the Fourier (power) spectrum.

The coefficients of the Fourier series are given by

$$a_n=\frac{2}{T}\int_0^T f(t)\cos(n\omega_0 t)dt \tag{7.2}$$

and

$$b_n = \frac{2}{T}\int_0^T f(t)\sin(n\omega_0 t)dt. \tag{7.3}$$

Equations (7.2) and (7.3) can be derived directly from the Fourier series which is given as an exercise for the reader to perform. Note that we integrate over one period.

Equation (7.1) need not be restricted to periodic functions as any general function may be described by an infinite sum of its Fourier components (this is its Fourier transform which will be discussed in the next section). Moreover, as this series does not require the derivatives of the function to exist it can be used to describe functions that are discontinuous or contain discontinuous derivatives. The Fourier series will provide a "best-fit" to the function (or signal) in the least-squares sense and it generally converges to the average behavior of the function. At discontinuities, it converges to the mean value of the function just on either side of the jump, and at sharp corners, that is, where there are discontinuities in the function's derivative(s), it overshoots the function.

So far, this discussion has been somewhat abstract so let us go through an illustrative example. A square wave can be thought of as (periodic) repetition of a step function. A step function over a period T is given by

$$f(t) = \begin{cases} -A, & -\frac{T}{2} < t < 0 \\ A, & 0 < t < \frac{T}{2} \end{cases}, \tag{7.4}$$

where A is the amplitude of the square wave. Given this definition the square wave is an odd function, that is, $f(-t) = -f(t)$, and all the a_n must be zero; remember that the cosine is an even function, that is, $f(-t) = f(t)$, such that the integration of Equation (7.2) goes to zero over one period. Our job then is to find the b_n as follows

$$
\begin{aligned}
b_n &= \frac{2}{T}\int_{-\frac{T}{2}}^{0} -A\sin(n\omega_0 t)\,dt + \int_{0}^{\frac{T}{2}} A\sin(n\omega_0 t)\,dt \\
&= \frac{4A}{T}\int_{0}^{T/2} \sin(n\omega_0 t)\,dt \\
&= \frac{4A}{n\omega_0 T}\left[-\cos(n\omega_o t)\right]_0^{T/2} \\
&= \frac{2A}{n\pi}(1-\cos(n\pi)) \\
&= \begin{cases} 0, n = 2,4,6,\ldots \\ \dfrac{4A}{n\pi}, n = 1,3,5,\ldots \end{cases}
\end{aligned}
$$

where the fact is that the function is used as odd, and the natural frequency is defined as $\omega_0 \equiv 2\pi / T$. Substituting these values into Equation (7.1) and summing to infinity we would end up with the square wave. In practice, we typically only retain the first few significant terms from the Fourier series.

Figure 7.1 plots the result of performing the Fourier series for a square wave with an amplitude $A = 1$ and a period $T = 4$ (seconds). The plot shows the first three non-zero Fourier terms of the series namely $n = 1$, 3, and 5. As more terms are added, we can see that the series does an increasingly better job of approximating the function. Where the square wave is a constant, the Fourier series oscillates around the function value with decreasing amplitude as we increase the number of terms in the series. As the Fourier series passes through the discontinuity in the function it converges on the average value of the function limits either side of the jump (in this case zero) and misses the function entirely just passed the jump. This is the *overshoot* that was mentioned previously. Unlike the oscillations about the constant function value (more generally the continuous parts of the function), the overshoot does not improve as rapidly as we increase the number of terms in the series.

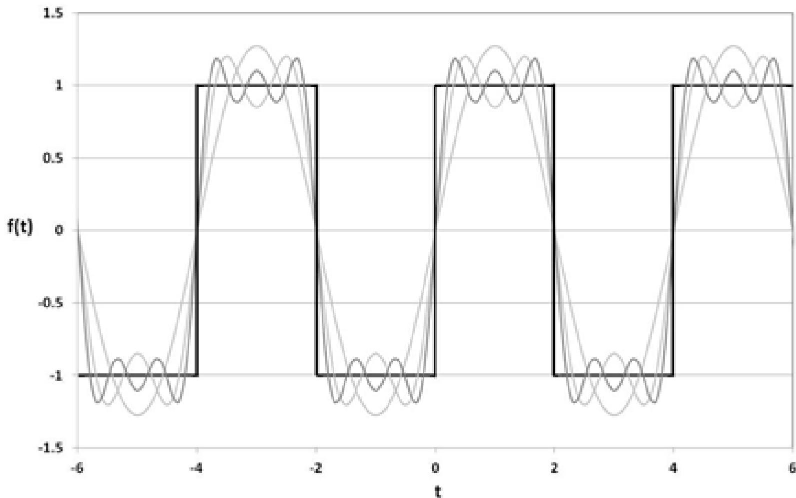

FIGURE 7.1: Approximation of the square wave using a Fourier series, keeping the first, second, and third non-zero terms in the series.

Most of the problems and functions that we will deal with will involve real numbers, that is to say, not complex numbers. However, it is sometimes convenient to express the Fourier series in terms of complex numbers. Returning briefly to Euler who derived the following complex identities

$$e^{i\theta} = \cos(\theta) + i\sin(\theta) \tag{7.5}$$

and

$$e^{-i\theta} = \cos(\theta) - i\sin(\theta) \tag{7.6}$$

where $i = \sqrt{-1}$ is the so-called imaginary number, and the Fourier series can be rewritten as

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{in\omega_0 t}. \tag{7.7}$$

The coefficients are now represented by the c_n which can be calculated using

$$c_n = \frac{1}{T}\int_0^T f(t)e^{-in\omega_0 t}dt. \tag{7.8}$$

It could be inferred from the above discussion that some functions of the Fourier series may not have a Fourier series representation. The Fourier series approximation may not converge on the function and in fact, it may not even converge at all. The Dirichlet's theorem defines the sufficient mathematical conditions of a function for its Fourier series representation to converge so that you can research them for yourselves.

7.2 FOURIER TRANSFORMS

In the preceding section, we have mentioned that any general, that is, not necessarily periodic, function can represent as an infinite sum of its Fourier components. To do this mathematically, the Fourier series is tweaked into Fourier integrals for it to deal with a non-periodic function. The basic idea is that a non-periodic function can be thought of as periodic with its period extending toward infinity, that is, the period becomes infinitely large but not actually infinity. This means that the natural frequency reduces toward zero, that is, it becomes infinitesimally small but not actually zero. By applying this mental manipulation, we can write Equation (7.7) as

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{in\Delta\omega_0 t} \tag{7.9}$$

where the coefficients are given by

$$c_n = \frac{\Delta\omega_0}{2\pi} \int_{-\infty}^{\infty} f(t) e^{-in\Delta\omega_0 t} dt, \tag{7.10}$$

and $\Delta\omega_0$ is our infinitesimal natural frequency. Note the change in the integral limits for the coefficients to reflect the idea that the period extends toward infinity and thus can also be shifted to extend to minus infinity. As the discrete values $n\Delta\omega_0$ are summed over infinity, it could be mapped on to a continuous variable that, for consistency, we shall simply call ω. Due to this modification, the infinite sum over n in Equation (7.9) becomes an integration over ω, on the infinite interval. Thus, the Fourier *integral* gives

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \left(\int_{-\infty}^{\infty} f(t) e^{-i\omega t} \, dt \right) e^{i\omega t} d\omega \tag{7.11}$$

where the traditional symbol is used for the infinitesimal element of the integration over ω, and define some function of ω as

$$g(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt \tag{7.12}$$

then Equation (7.11) becomes

$$f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(\omega) e^{i\omega t} d\omega. \tag{7.13}$$

Equations (7.13) and (7.12) define an integral transform and its inverse, respectively. These are commonly known as the Fourier transform and the inverse Fourier transform. The multiplicative factor in both these integrals can be chosen to be anything, so long as their product equals $1/2\pi$; the form shown is called symmetrical for obvious reasons. By knowing the Fourier transform and its inverse, it allows us to map a function (or data) from one domain to another where perhaps a mathematical operation on the function is easier in the transformed domain. After applying the operation, the modified function can be transformed back (inverse transform) to the original domain. Harmonic analysis is an example of where this kind of technique is used.

For convenience and shorthand, the Fourier transform, and its inverse can be written as

$$f(t) = \mathcal{F}\{g(\omega)\} \tag{7.14}$$

and

$$g(\omega) = \mathcal{F}^{-1}\{f(t)\}. \tag{7.15}$$

Note that instead we could have started with the time variable and transformed that into the frequency variable and, in which case, we would have to reverse our definitions (Equations (7.12) and (7.13)). So long as we are consistent with what is the transform and

what is the inverse it does not matter what our original variable was to begin with.

The choice of our variables, that is, time and frequency, in deriving these transforms were for instructive purposes only and Fourier transforms need not be restricted to them. They can be applied to other types of variables including those described by vectors (e.g., three-dimensional space). For instance, if we considered the variable λ that represents the wavelength of some quantum particle in one or more dimensions then its Fourier transform would be the wavenumber (or vector) k. This has important applications in solid-state physics where the use of k-space or momentum-space is beneficial in understanding several electronic and optical properties of matter. As an aside, the reason why it is called momentum-space is due to De' Broglie (pronounced like Troy); hk, where h is Planck's constant, gives the momentum of a quantum particle.

A Fourier transform pair have several significant properties not least among them that the operation is linear. That is to say, if $f_1(t)$ has a Fourier transform $g_1(\omega)$, and similarly $f_2(t)$ has a transform $g_2(\omega)$, then the Fourier transform of $f_1(t)+f_2(t)$ is simply $g_1(\omega)+g_2(\omega)$.

Another property is the scaling relation that has an interesting physical interpretation. It can be shown that

$$\mathcal{F}\{f(a\,t)\} = \frac{1}{|a|}g\left(\frac{\omega}{a}\right) \tag{7.16}$$

where α is a scaling factor that can be positive or negative. Equation (7.16) shows that if we squeeze the $f(t)$ along the t axis, that is, $|\alpha|>1$, then its corresponding Fourier transform broadens along the ω axis and also reduces in height by a factor of $|\alpha|$. Conversely, if $|\alpha|<1$ then we broaden $f(t)$ and squeeze $g(\omega)$, this time increasing its height. In other words, the more localized the function is in time, say, the more delocalized it is in frequency, and vice versa. Remember that we are not restricted to time and frequency variables, we could just as correctly use 3-D spatial variables and momentum-space variables as a Fourier transform pair. In this case, the more accurately we know the position of a particle, the less accurately we know its momentum. If you have not come across a

chapter called Heisenberg yet and his uncertainty principle, then you soon will.

Other properties exist for shifting relations (moving the coordinate system) and the symmetries (odd or even functions) and complexities (real and/or imaginary functions).

7.3 THE DISCRETE FOURIER TRANSFORM

As with all numerical procedures, we first must find a way of representing a continuous variable as a discrete set of points. Note that in doing so we will not be computing the true Fourier transform but we intend to find a reasonable approximation to the transform.

Let us consider $f(t)$ as a time-dependent physical quantity obtained from actual measurements such that we have N data points taken at equidistant increments of Δt. In other words, we have the data points $(\Delta t, f(m\Delta t))$, $m = 0,1,2 \ldots N-1$. If we have sufficient data points to adequately describe the behavior of the function over a given length of time T, and that the function is periodic beyond this region, we may then use the notion that

$$\Delta\omega = \frac{2\pi}{T} = \frac{2\pi}{N\Delta t}. \tag{7.17}$$

Under these conditions, we can write the Discrete Fourier transform (DFT) and its inverse as

$$f(m\Delta t) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} g(n\Delta\omega) e^{i2\pi mn/N} \tag{7.18}$$

and

$$g(n\Delta\omega) = \frac{1}{\sqrt{N}} \sum_{m=0}^{N-1} f(m\Delta t) e^{-i2\pi mn/N} \tag{7.19}$$

where we have kept the symmetric form; in this case, the product of the factors must equal 1/N. For convenient notation let us now drop the Δt and $\Delta\omega$ in the function arguments and use the corresponding integer multiple as a subscript instead, that is, $f(m\Delta t) \rightarrow f_m$ and $g(n\Delta w) \rightarrow g_n$.

To implement Equations (7.18) and (7.19) into a computer program it is convenient (but not necessary) to separate the functions into their real and imaginary parts. This makes the coding somewhat more intuitive and means that we only deal with real numbers (imaginary numbers are essentially a real number multiplied by $i = \sqrt{-1}$, which we can drop in a computer program). In separating the real and imaginary parts we obtain the following:

$$\mathrm{Re}(f_m) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} \left[\mathrm{Re}(g_n)\cos(\theta) - Im(g_n)\sin(\theta)\right]; \tag{7.20}$$

$$Im(f_m) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} \left[Im(g_n)\cos(\theta) + \mathrm{Re}(g_n)\sin(\theta)\right]; \tag{7.21}$$

$$\mathrm{Re}(g_n) = \frac{1}{\sqrt{N}} \sum_{m=0}^{N-1} \left[\mathrm{Re}(f_m)\cos(\theta) + Im(f_m)\sin(\theta)\right]; \tag{7.22}$$

and

$$Im(g_n) = \frac{1}{\sqrt{N}} \sum_{m=0}^{N-1} \left[Im(f_m)\cos(\theta) - \mathrm{Re}(f_m)\sin(\theta)\right] \tag{7.23}$$

where $\theta = 2\pi nm / N$.

The code contained in the file *DFT_bellcurve.cpp* performs the DFT (in one dimension) on the (normal) Gaussian distribution function with the parameters specified, then performs the inverse transform to check the correctness of the programming. The output is somewhat uninteresting in the sense that we have mapped the function back on to itself but at least it shows we have coded the DFT correctly. Note that in our implementation we have condensed the factors $1/\sqrt{N}$ into a single factor of $1/N$, which can either multiply the transform or the inverse but not both. By comparing the implementation of the DFT function (*Fourier.cpp*) to Equations 7.20-7.23 satisfy yourself that the reversal of sign of the imaginary part of the transform is required (complex conjugate).

The DFT though straightforward to program is not efficient in terms of computational effort. Each component of the transform requires that we sum over the N data points of the signal, and there

are N such components. This leads to an operation count that is proportional to N^2; you can see this in the function code where we have the nested **for** loops. This situation only gets worse as you increase the number of dimensions in your data. How do we get around this limitation?

7.4 THE FAST FOURIER TRANSFORM

7.4.1 Brief History and Development

The fast Fourier transform (FFT) has been independently discovered and rediscovered by various people, the earliest version appearing in the literature being attributed to Gauss in 1866. It appeared as an unpublished manuscript in his collected works. The actual date Gauss wrote this manuscript is presumed to be around 1805, which predates Fourier's original work by 2 years. For whatever reasons Gauss's idea was largely ignored by the scientific community and no one connected it to the use of modern computation. In 1965, the American mathematicians James William Cooley and John Wilder Tukey published an article that discussed in detail the use of a machine algorithm to calculate complex Fourier series. This is largely credited as the first formal use of the FFT on a "modern" computer. However, more than 20 years before a pair of physicists Cornelius Lanczos and Gordon C. Danielson gave a particularly lucid description of the FFT derivation in their 1942 publication on practical Fourier analysis of X-rays scattered from liquids.

Let us assume N is an even number. We can then write the DFT as a summation over the even-numbered points and a summation over the odd-numbered points. Mathematically this is written as

$$\begin{aligned} g_n &= \sum_{m=0}^{(N/2)-1} f_{2m} e^{-i2\pi n(2m)/N} + \sum_{m=0}^{(N/2)-1} f_{2m+1} e^{-i2\pi n(2m+1)/N} \\ &= g_n^{(even)} + g_n^{(odd)} e^{-i2\pi n/N} \end{aligned} \tag{7.24}$$

where we define

$$g_n^{(even)} = \sum_{m=0}^{(N/2)-1} f_{2m} e^{-i2\pi nm/(N/2)} \tag{7.25}$$

as the even-numbered points, and

$$g_n^{(odd)} = \sum_{m=0}^{(N/2)-1} f_{2m+1} e^{-i2\pi nm/(N/2)} \tag{7.26}$$

as the odd-numbered points. Note that we have ignored the $1/\sqrt{N}$ factor here, which can be easily reintroduced at a later stage. Let us take stock of what we have just done. By splitting the DFT into even and odd summations we have essentially produced two new DFTs, Equations (7.25) and (7.26), with half the number of points of the original transform. Hence the number of operations required is now proportional to $2\times(N/2)^2$, that is, half the original. The beauty of this algorithm is that we can keep going and further split those new DFTs into their even- and odd-numbered points, and so on until we reach the level where there is only one component to find in the summation. However, this requires that the number of points at each subdivided level contained within the summation remains even. This can easily be insured by specifying that N is an integer power of two. For instance, let $N = 2^k$ then after k subdivisions, there will be N DFTs to compute each with only one component to find. In other words, instead of the operation count being proportional to N^2 it is now proportional to Nk or more generally $N\log_2 N$.

7.4.2 Implementation and Sampling

The reason why Cooley and Tukey are generally credited with the discovery of the FFT as applied to modern computing was their clever way of interweaving the summation pairs at the lowest level of the algorithm. This interweaving is just an exercise in bookkeeping which is rather tedious and can make the coding somewhat complicated. Rather than discussing the interweaving strategy at length, the function FFT is provided in the file *Fourier.cpp*, which contains an FFT algorithm, for your use. For interested readers, Landau's book, *A Survey of Computational Physics*, 2008, pp. 256–263 for an in-depth discussion of the interweaving strategy is recommended.

Before we continue, let's explore briefly how to interpret the spectrum resulting from the FFT function. It assumes that the (time) data passed to it is periodic on the interval for which it is defined, which in turn implies the resulting transform is also periodic. Figure 7.2 illustrates this point. Here we imagine a sketch of the frequency spectrum of some arbitrary harmonic (time) signal with the natural frequency ω_0 calculated using the FFT function (The broadening of the peak is for illustrative purposes but can be caused by actual properties of the data and the sampling). The solid curve on the positive frequency portion of the plot is the complete vector output from the function. Assuming the vector produced by the FFT has length N that is some integer power of two then we can say the following:

- the zero frequency is located at index zero;
- positive frequencies correspond to indexes 1 -> N/2 - 1;
- negative frequencies correspond to indexes N/2 + 1 -> N - 1 (most negative to least negative); and
- and index N/2 gives the Nyquist critical frequency (either positive or negative).

We will discuss the meaning of the Nyquist critical frequency in due course. Therefore, the algorithm is considered as computing normal, forward time, and time-*reversed* frequencies; essentially a mathematical quirk of the FFT algorithm. The zero frequency is countes as a positive frequency. To demonstrate we can write

$$\cos(\omega_0 t) = 0.5\cos(\omega_0 t) + 0.5\cos(-\omega_0 t) \tag{7.27}$$

as cosine is an even function. Conversely

$$\sin(\omega_0 t) = 0.5\sin(\omega_0 t) - 0.5\sin(-\omega_0 t) \tag{7.28}$$

as sine is an odd function. As the spectrum is shared between the positive and negative frequencies its intensity (the Fourier coefficient value) is half what we would expect if we just considered the "physically" significant positive frequencies. The upshot of all this is that when recording the spectrum data, we could only store the first $N/2$ points and multiply their values by two in order to obtain the

"physically" correct power spectrum. More precisely, the intensity (or power) of the spectrum is given by the integration of the spectrum over the entire range of the transformed domain.

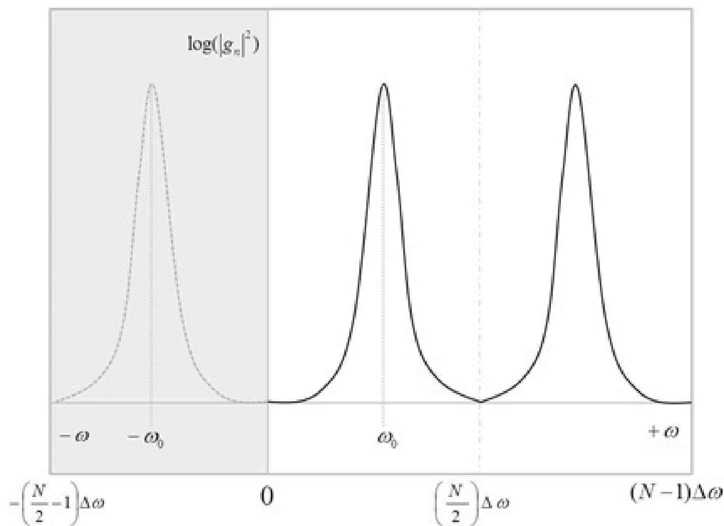

FIGURE 7.2: Sketch of a frequency spectrum of some harmonic oscillator with natural frequency ω_0. We have deliberately included peak broadening to clearly demonstrate the interpretation of the spectrum.

Modify the program you have written to analyze the spectrum of the following function

$$f(t) = \cos(5\pi t), \tag{7.29}$$

sampled once per second for 32 s, then twice per second for 16 s, and so on up to 16 times a second for 2 s. Here, we keep N constant at 32. You will need to compute the equivalent discrete frequencies of the array indices; essentially the index is divided by the time domain range, scaled by 2π to get the angular frequency. To view the spectrum, we should plot the magnitude of the transform values against the discrete frequencies. In other words, the spectrum can be considered a histogram with the width of each (frequency) bin given by $\Delta\omega$ and its height given by the square root of

$$|g_n|^2 = \mathrm{Re}(g_n)^2 + Im(g_n)^2. \tag{7.30}$$

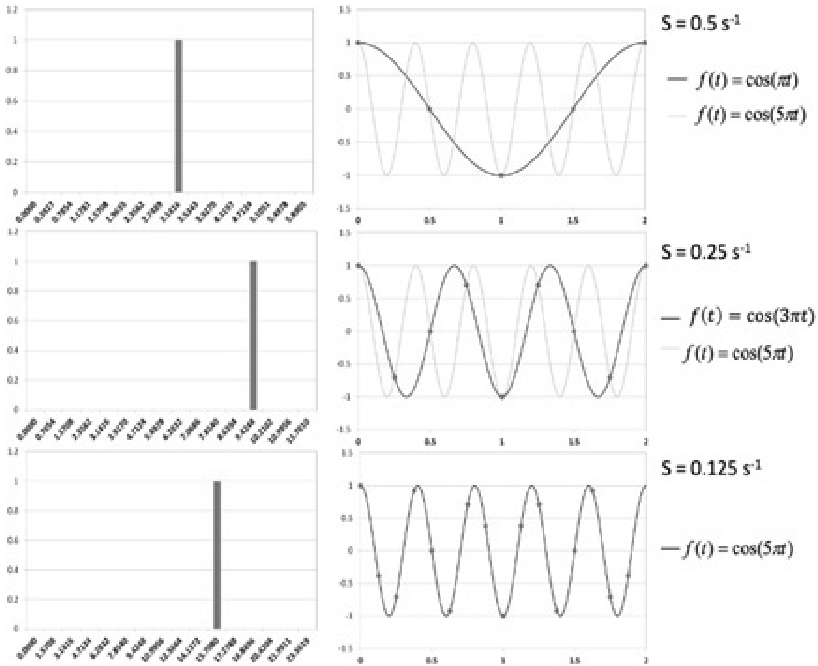

FIGURE 7.3: Left - the frequency spectrums of Equation (7.29) sampled at twice, four times, and eight times per second respectively. Right - the time sampled function with the detected frequency harmonic shown.

The code library uses the std::complex class (using a template argument of double) to represent complex numbers. There are several functions that act on objects of std::complex type that perform the expected mathematical operations such as conjugation and finding the magnitude.

After performing the FFT on the given function at different sampling rates, the results were obtained as depicted in Figure 7.3. We know that the (angular) frequency of the function described by Equation (7.27) must be $\omega = 5\pi \approx 15.7$. Why then do we see a frequency of π in the spectrum when sampling at a rate of twice a second, and indeed a frequency of 3π when sampling at four times a second? The answer lies in the plot of the time sampled function overlaid on the actual function as shown in the right-hand column of Figure 7.23. When sampling at twice per second (top) we see that the sampled data resemble a triangular waveform with a period of 2 seconds, equivalent to a harmonic frequency of π. Similarly, when

sampling at 0.25 per second (middle) the curve $f(t) = \cos(\pi t)$ pass through the sampled points leading to the erroneous frequency spectrum. This is called *aliasing*; the higher frequency "signal" has been aliased by lower frequency harmonics. This happens because, in these two cases, we are *under-sampling* the function; our sample rate is not sufficiently high to capture the true waveform. As a rule, you should sample the signal at a rate at least twice the highest frequency component contained in the signal. This is the Nyquist critical frequency. In our example, the angular frequency of our waveform is 5π equivalent to a frequency of 2.5 Hz. Thus, to avoid under-sampling we should take data at time intervals at least 0.2 s apart or less. Indeed, when we sample at intervals of 0.125 s, and we recover the correct frequency spectrum.

What then happens when we sample at a rate of 32 times per second for one second? The FFT spectrum appears to distort across all the discrete frequency bins. This problem is known as *leakage* and occurs when there is a lack of frequency resolution such that the actual frequency of the data does not match one of the frequency bins. In this case, the FFT tries to compensate by distributing the transform across nearby frequencies, in other words, it leaks. To alleviate this problem, we can increase the total observation time, that is we increase N but without changing the sampling rate. Try sampling at the same rate of 32 per second but for 2 s, that is, increase N to 64, and see if we get a better outcome.

The leakage problem can be attributed to wherein the time domain we finish sampling. Remember that the FFT assumes the data you pass to it is periodic on the observation interval for which it is defined. If the sampling finishes mid-period, then the FFT "sees" a discontinuity in the function. As we know from the Fourier series a discontinuity is better approximated by increasing the number of terms in the sequence. Comparatively, the FFT increases the number of frequencies detected in the spectrum to deal with the discontinuity. It is therefore advantageous to use a sampling rate that is proportionate with the period of the function.

EXERCISES

7.1. Calculate the Fourier series for a Saw-tooth waveform. Plot the results to get a clear picture of how the series converges to the function.

7.2. Investigate the overshoot in the square waveform as the number of terms in the Fourier series increases. Calculate the error between the series approximation and the function, and hence determine the behavior of the overshoot as we retain more terms in the series—this is called the Gibbs phenomenon.

7.3. Derive the Fourier coefficients of Equations (7.2) and (7.3). Hint: sine and cosine functions are orthogonal.

7.4. Derive the scaling property for the Fourier transform; Equation (7.16). Then derive the similar property for the inverse transform.

7.5. Find a way to time the operation of a program in Fortran then evaluates the runtimes of the DFT algorithm versus the FFT algorithm for the same set of data. Check the statements that the DFT algorithm operation count is proportional to N^2 and the FFT operation count is proportional to $N\log_2 N$. Also, check that the output from each algorithm is the same for the same input (within unit round-off error precision).

7.6. What is the shape of the Fourier transform of the rectangle function? How does this relate to the diffraction of a wave through a single slit?

7.7. Use the FFT subroutine to obtain the spectrum of the function $f(t)=\sin(5t)$. Use a sampling rate that is sufficiently rapid to avoid under-sampling. Can you derive a sampling rate that avoids the problem of leakage?

7.8. Consider the function $f(t)=\cos\left[(1+\alpha)t\right]+2\cos\left[(2-\alpha)t\right]$, for α in the range $[0,1]$. Investigate how the sampling rate and overall observation time affects the resolution of the frequency peaks.

CHAPTER 8

MONTE CARLO METHODS

Monte Carlo methods (or Monte Carlo experiments) are a broad class of computational algorithms that rely on repeated random sampling to obtain a numerical result. They are often used in physical and mathematical problems when it is impossible to obtain an analytical solution, and the application of a direct algorithm is infeasible. Monte Carlo methods are mainly used in three distinct problems: numerical integration, simulation, and optimization. The first two in this list and how they relate to physics problems are discussed in this chapter.

8.1 MONTE CARLO INTEGRATION

8.1.1 Dart Throwing

"Hit and miss" integration, also known as the shooting method, is arguably the most intuitive type of Monte Carlo method to understand. To demonstrate the application of this approach, let us discuss a novel way of approximating the value for π (see Figure 8.1). It shows the upper right quadrant of a circle of unit radius circumscribed by a unit square. Imagine throwing darts randomly at this board (some of you may have had a similar experience already in the student's union bar). Of the total number of darts that hit within

the square, the fraction of those that land within the circle will be approximately equal to the ratio area of the circle contained by the square. Mathematically, we write

$$\frac{A_{circle}}{A_{square}} \approx \frac{N_{circle}}{N_{thrown}}. \tag{8.1}$$

Here we have the constraint that darts cannot be thrown outside of the square and A_{circle} is the area contained in the unit square.

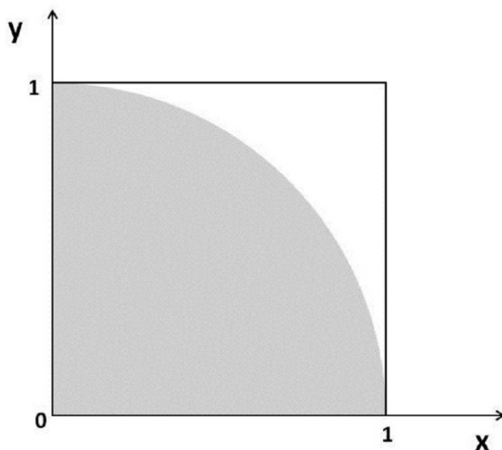

FIGURE 8.1: The Monte Carlo "dart board" used to approximate π.

Remembering your geometry basics, we can substitute and rearrange the equation above to give an approximation formula for π, such that

$$\pi \approx \frac{4N_{circle}}{N_{thrown}} \tag{8.2}$$

In other words, the probability that a dart will hit the shaded area is equivalent to one-quarter of the value of π. Despite the fun you can have in trying to make the dart-throwing random, attempting to physically perform this experiment soon becomes tedious as you need a large number of thrown darts to get a reasonably accurate approximation for π. Instead, we make a computer simulate the dart-throwing by having it generate random numbers.

Now before anyone gets militant on my personage computers do not generate true random numbers as they are deterministic machines. That said, on some modern systems, there are hardware devices that can provide true randomness via the stochastic processes involved in their operation. Hardware aside, computers can generate what is known as pseudorandom numbers via a recursion formula; given a starting point, generally referred to as the random number seed, the generator produces a sequence of "random" numbers by performing mathematical operations on the previous "random" number. Rigorous statistical tests can be applied to the outputs of these generators to check that the numbers are random in relation to one another. As a cautionary note, a random number generator will produce the identical random number sequence for the same seed. Hence, for multiple trials, different seeds must be found to produce different random number sequences. Typically, this is done by using the system's clock. C++ has a number of built-in classes that perform pseudorandom number generation on different distributions that will be adequate for our purposes.

For each random throw, we generate two random numbers, x and y, that represent the displacement from the origin to where the dart hit in the horizontal and vertical directions, respectively. Using the Pythagorean Theorem, the distance from the origin can be calculated and thus it could be determined whether the dart landed within the circle. That is, if the distance is greater than one unit it missed, less than or equal to one unit it hit. By keeping count of the total number of darts thrown that is, the number of random (x,y) coordinates generated, and the number that hit the circle we can approximate π using Equation (8.2).

The file *piMonte.cpp* contains a program to perform this experiment. The code generates a pair of (uniformly distributed) random numbers, both on the interval $[0,1]$, to simulate where the thrown dart lands within the unit square. After computing the distance from the origin, we either add one to the counter if it is a hit or do nothing if it is a miss. After every 10 darts thrown, we estimate π using Equation (8.2) and store both that value and the current number of darts thrown for plotting after the processing loop has completed.

Notice that we save the random numbers generated to check the randomness of the throws.

Figure 8.2 shows the results from running this program on four separate occasions, using a total of 1000 dart throws. Here we can see that our random number generator has done an adequate job; the four different runs have produced four *different* results as we should expect if we had physically performed the experiment on four separate occasions.

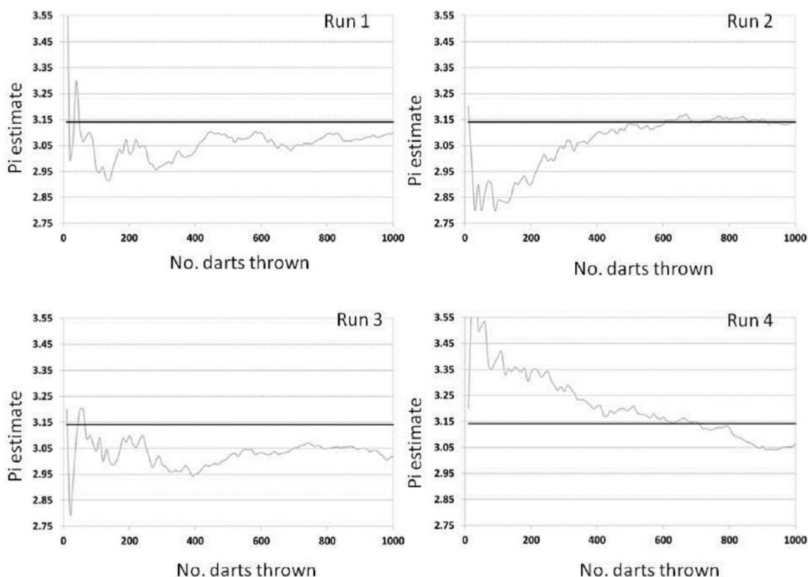

FIGURE 8.2: Results of estimating π from a Monte Carlo integration for four separate runs. The black line represents π.

The black line in each of these plots represents the actual value of π. The figure suggests that although we are not guaranteed to converge on the actual value of the integration, the approximation does, to some extent, stabilize as we increase the number of throws. However, remember that the results are *accumulated*. Thus, as the number of throws increases the influence of the next throw is reduced, and the variation in the estimate from one throw to the next necessarily diminishes. In other words, the results at the end of the experiment are very much influenced by the outcome of the throws at the start of the experiment.

If you're looking at the results plotted in Figure 8.2 and wondering why we would bother with Monte Carlo integration at all remember that this is an illustrative (and simple) example. If the integration can be done easily by other means, then the Monte Carlo method should *not* be used. The Monte Carlo integration comes into its own when other numerical techniques are difficult, if not impossible to implement.

Figure 8.3 plots histograms of performing 10,000 dart-throwing integrations with 100 darts per integration in the top panel and 1000 darts per integration in the bottom panel. Note that both plots are over the same range, but they have different bin widths; 0.1 for the top plot, and 0.01 for the bottom plot, which is related to the number of darts thrown per integration.

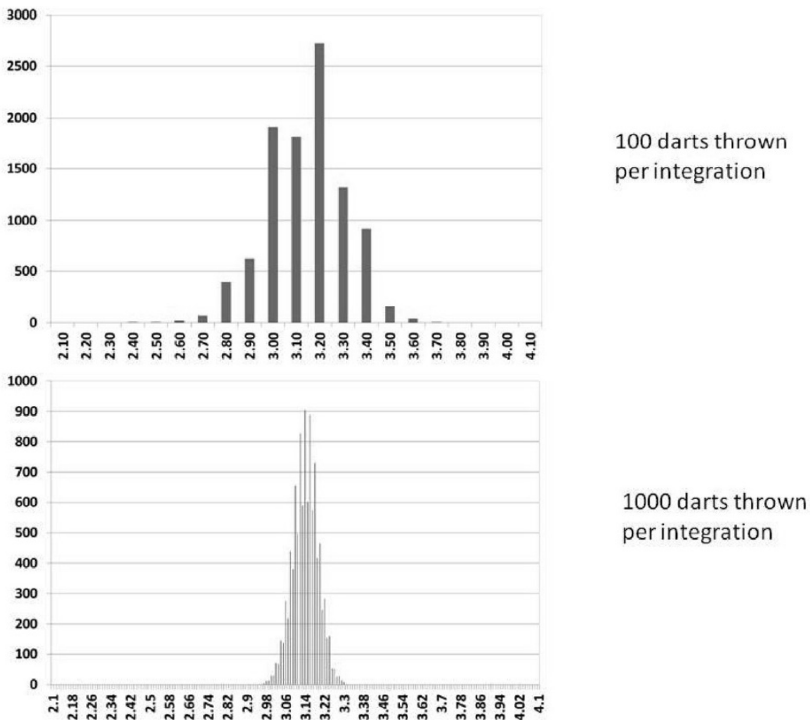

FIGURE 8.3: Histograms of 10,000 Monte Carlo integrations of π using 100 darts per integration (top) and 1000 darts per integration (bottom).

The curves are being the bell shape of a normal (or Gaussian) distribution; in fact, with an increasing number of integrations, the distributions would become much smoother and would approach the ideal bell shape. As can be seen from the plots as we increase the number of darts thrown per integration the distribution of estimates for π narrows about the true value. In other words, the mean value of the distribution becomes a more accurate value for π.

From Figure (8.3), the width of the distribution at half height for the 1000 darts per integration case is roughly one-third of that for the case of 100 darts per integration. If you remember your probability theory, you should recall that the width of a normal distribution of estimates of a value is proportional to one over the square root of the total number of points used to compute each estimate. In other words, the factor difference between the widths of these two distributions should be equal to $1/\sqrt{10}$, which is what we find.

Moreover, the standard deviation of the mean, which is a measure of the width of the distribution, can be itself estimated from a *single integration* using

$$\sigma_N = \sqrt{\frac{\frac{1}{N}\sum f_i^2 - \left(\frac{1}{N}\sum f_i\right)^2}{N-1}} \tag{8.3}$$

where f_i is the estimate of the value (π in our case), after the ith point is sampled (dart is thrown), and N is the number of points sampled (darts thrown) in total. For large N, we can drop the minus one in the denominator. It is of note that Equation (8.3) can be updated after each new random point is sampled; in which case N becomes equal to the value of i we have reached. This means we can monitor the confidence we have in the estimate of the integrated value as we increase N. Remember that the estimate lies within σ of the precise average to a 68.3% degree of confidence; within 2σ to a 95.4% degree of confidence; within 3σ to a 99.7% degree of confidence; and so on. To decrease σ and therefore improve the accuracy in the estimate we merely sample more random points. The drawback to this method is that the improvement can only go as the square root of N. This takes us back to the point

made previously that there is a law of diminishing returns due to the accumulation of data; when we have already sampled 1000 points, say, one more sampled point makes little difference to the outcome, but another 1000 would.

8.1.2 General Integration Using Monte Carlo

In our dart-throwing method example above to estimate π we have, in a round-about fashion, approximated the integral

$$\int_a^b f(x)\,dx = \int_0^1 \sqrt{(1-x^2)}\,dx = \frac{\pi}{4}. \tag{8.4}$$

A slightly more direct method of using the Monte Carlo integration would be to sample (uniformly distributed) random values of x on the interval $[a,b]$, and finding the average of the function evaluations, $f(x)$. In general, for a one-dimensional integration, we are using the notion that

$$\int_a^b f(x)\,dx = (b-a)f \tag{8.5}$$

where f is the precise mean average of the function on the interval $[a,b]$. This has a very straightforward geometrical interpretation as depicted in Figure 8.4.

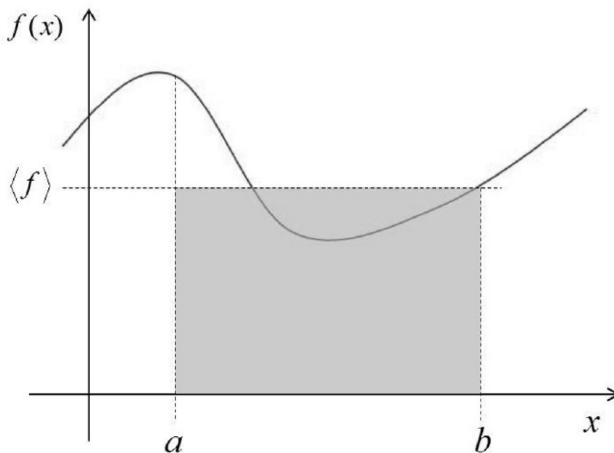

FIGURE 8.4: Geometrical interpretation of Monte Carlo integration.

The integration, which is the area under function defined on the interval, is equal to the area of the shaded rectangle. The Monte Carlo method is an attempt to estimate the precise function average, f. Notice that because of this interpretation $\pi/4$ must be the precise function average of the unit, quarter circle.

Formally we write the Monte Carlo integration estimate as

$$\int_a^b f(x)dx \approx \frac{(b-a)}{N}\sum_{i=1}^{N} f(x_i) \tag{8.6}$$

where N is the total number of randomly sampled points.

In the file *Monte_Carlo.cpp* you will find a function that performs the Monte Carlo integration according to Equation (8.6). You can check that it works by choosing an easily analytical integral and seeing if we obtain the same result using the Monte Carlo method; see *monte_carlo_integration.cpp* for an example. We could also use this information to check how well the estimate for the standard deviation models the actual error in the integration.

The Monte Carlo method of integration is most effectively used in the computation of multidimensional integrations where the application of more direct methods is either infeasible or impossible. To perform a multidimensional integration via the Monte Carlo method we simply find random numbers for all the variables involved, find the value of the function at those coordinates, then update the sum. For instance, a two-dimensional integration can be written as

$$\int_c^d \int_a^b f(x,y)dxdy = \frac{(d-c)(b-a)}{N}\sum_{i=1}^{N} f(x_i, y_i). \tag{8.7}$$

As we add more dimensions, we generate more random numbers for the additional dimensions and multiply by the relevant integration interval; the sum over f divided by N still provides an estimate of the precise function average defined with the integration region. Notice that because of this simplicity the Monte Carlo method of integration has some inherent advantages over more direct numerical techniques.

The error in the direct methods for numerical integration (trapezoidal rule, Simpson's rule, etc.) stems from the number of terms retained in the Taylor series approximation of the integrand function.

For example, the trapezoidal rule approximates the integrand with a linear polynomial, which, as we have seen, has an error that is $\mathcal{O}\left(h^2\right)$, where h is the strip width across the integration interval. If we wish to halve the error in our numerical approximation of the integration in one dimension then we decrease h by a factor of $\sqrt{2}$; this is equivalent to increasing N (the number of function evaluations) by the same factor. For a two-dimensional integration to halve the error we must apply this modification in both dimensions such that N increases by a factor of 2 in total. For three dimensions N must be increased by a factor of $2^{3/2}$. In general, for a d dimensional numerical integration to halve the error in our approximation we would have to increase N by a factor of $2^{d/m}$, where m represents the order accuracy of the numerical integration method used to compute the approximation (for the trapezoidal rule $m = 2$; Simpson's rule $m = 4$; and so on).

The error in the Monte Carlo method is different. As we have just discussed the error produced by a Monte Carlo computation is probabilistic in nature; we can say that the approximation calculated is within one standard deviation of the "true" value 68.7% of the time. To improve the approximation, that is to reduce the standard deviation and thus make the average value converge on the "true" value, we increase the number of random points sampled, N. As this is a probabilistic process, we know that the error will reduce as $\sqrt{N}$. Thus, to halve the error we increase N by a factor of 4. *This is independent of the dimensionality of the integration*! To explain, we perform a multidimensional integration via the Monte Carlo method by computing as many random numbers as there are dimensions, then evaluating the function at the coordinates specified by those random numbers and updating the sum. Note that this is a true scattershot approach; none of the dimension variables are held constant, we just keep "shooting" and evaluating a single value for the function at those coordinates randomly generated.

To demonstrate, let us imagine a four-dimensional integration that we can perform either by the trapezoidal rule or the Monte Carlo method. After obtaining the approximation from both methods we would like to halve the error in each. From our discussions, we can see that both require the number of function evaluation

points to be increased by a factor of 4. In other words, the rate of convergence for the two methods is comparable when performed on a four-dimensional integration. To avoid confusion, here we are talking about the rate of convergence of an approximation to the "true" value, not the absolute value of the error. It is likely that the trapezoidal rule is more accurate (has a less absolute error) than the Monte Carlo method to start with. That said, for dimensions higher than four the Monte Carlo method will converge more rapidly than the trapezoidal rule. Indeed, for integrals of sufficiently high dimensionality, the Monte Carlo method will converge more rapidly than any direct method that has been discussed earlier, dependent on the method's order of accuracy.

A secondary advantage to the Monte Carlo method is the number of function evaluations that must be performed in order to gain an approximation to the integral. To illustrate, imagine a 10-dimensional integration that we are computing via the Monte Carlo method. Let us say we evaluate the integrand function 10 times, that is, we generate 10 random numbers for each function evaluation, that is, 100 points in total. Now we want to halve the error in our approximation so as stated we increase N by a factor of 4, that is, we have to generate 400 random numbers. If we evaluated the same integration using the composite trapezoidal rule, again with 10 function evaluations per dimension, then it would have $10^{10} = 10$ billion function evaluations to perform. To halve the error, we would have to increase N by a factor of 2^5; we would now have to perform 320 billion function evaluations. Thus, the trapezoidal rule will probably give a more accurate result than the Monte Carlo method, and it would certainly take more time (an infeasible amount) to obtain. Even though the Monte Carlo estimate will be crude, the method does give a quantifiable measure of the error, and the knowledge that we can improve this error by taking just a few more randomly sampled points.

8.1.3 Importance Sampling

Before completing the discussion about Monte Carlo integration, the author discusses the technique that can help improve the accuracy of the method called importance sampling. Importance sampling uses information about the function to place more randomly sampled

points where the function is largest, meaning that the approximation is more accurate for the same number of sampling points. To do so we find a function g(x) that approximates the integrand function f(x) over the integration interval so that we can write

$$\int_a^b f(x)dx = \int_a^b \frac{f(x)}{g(x)} g(x)dx = \int_{y^{-1}(a)}^{y^{-1}(b)} \frac{f(y^{-1})}{g(y^{-1})} dy \tag{8.8}$$

where

$$y = \int^x g(t)dt. \tag{8.9}$$

Interpreting Equation (8.8), we see that instead of integrating $f(x)$ over x, we integrate the ratio $f(x)/g(x)$ over y. This has the geometrical effect of flattening the integrand over the integration interval, thus making the Monte Carlo method more accurate. Remember we are attempting to approximate the precise function average over the integration interval. Having a "flat" function makes this easier.

For example, consider the integral

$$I = \int_0^{\pi/2} \sin(x)dx. \tag{8.10}$$

This has an analytical value of 1. We can approximate sine using the first term of its Taylor series expansion $\sin(x) \approx x$, such that the integral becomes

$$\int_0^{\pi/2} \sin(x)dx = \int_0^{\pi^2/8} \frac{\sin(\sqrt{2y})}{\sqrt{2y}} dy, \tag{8.11}$$

where

$$y = \int^x t dt = \frac{x^2}{2} \tag{8.12}$$

and

$$x = \sqrt{2y}. \tag{8.13}$$

Without importance sampling, the Monte Carlo method using 100 sampled points, produces a result with $|3\sigma| \approx 0.15$. For the same number of sampled points but with importance sampling, we obtain

a result with $|3\sigma| \approx 0.04$. Note that the approximation function $g(x)$ should be reasonably good across the entire integration interval; otherwise, the result is likely to be worse (see Exercise 1).

8.2 MONTE CARLO SIMULATIONS

One of the first uses of a Monte Carlo method was in determining the thickness of the shielding required to stop neutrons from leaking from a nuclear reactor. Developed in the 1940s by Stanislaw Ulam when he worked on the Manhattan project it coincided with the birth of modern computing. Indeed, Jon Von Neumann was the first to successfully program a Monte Carlo method on ENIAC (Electronic Numerical Integrator and Computer) in the late 1940s and into the 1950s.

The process of a neutron traveling through metal is very much a random one; the neutron collides with the metal atoms and is scattered in a random direction. Any influence of the neutron's previous motion is lost in the (random) scattering process i.e., there is no correlation between the results of a particular collision and the neutron's initial motion. This kind of random process is called stochastic, and they frequently occur in physics; molecular diffusion; percolation of atoms on (growth) surfaces; and radioactive decay, to name but a few.

8.2.1 Random Walk

Let us begin our discussion of Monte Carlo simulations with a simple drunken walk. As some of you may have already found out, a drunken walk can be described as a random experience. Mathematically speaking a random walk is one in which you are equally likely to step in any direction. The question normally posed is how far you will travel in a given number of steps.

To answer this question, we simulate the walk using random numbers, and, of course, some assumptions and constraints. The first assumption is that each of your strides is equal in length and that you only walk along with the cardinal directions. In other words, you are

on a two-dimensional unit square grid moving from point to point. After reaching each point you have an equal probability of going north, south, east, or west. This can be simulated by generating a random number on the interval [0,1], with equal intervals defining the direction taken, for example, 0-0.25 walk north, 0.25-0.5 walk east, and so on (the same outcome could be achieved using the integers 1-4). As the simulation runs, we keep track of the x and y distances traveled from our starting point (origin) and calculate the distance traveled using Pythagoras, either at the end of the run or updated after each step. As this is a stochastic process, we do not gain much insight from one random walk, and so we should run the simulation many times over to obtain statistically valid results.

After writing such a program we obtain the results presented in Figure 8.5. Here we have results for $N = 10$ up to $N = 1000$ in increments of 10 where each value plotted is the average of 1000 simulations. Although this drunken walk scenario may seem somewhat oversimplified can you think of any real physical situations to where it might be applied? (Hint: the grid need not be a square grid but a regular grid of some other shape that may have more possible directions of travel.)

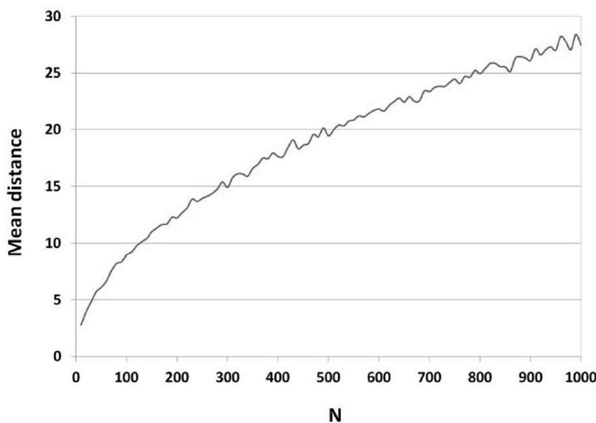

FIGURE 8.5: Mean distance travelled versus number of steps taken in a drunken walk.

We can relax the constraint that we only walk along with the cardinal directions and allow any direction on the two-dimensional surface, maintaining a stride length of one. To do this we consider

the angle that governs the direction of our next step. In other words, we randomly sample the angle φ on the interval $[0,2\pi]$ and use trigonometry to determine the x and y distances moved per step, that is, $x=\cos(\varphi)$ and $y=\sin(\varphi)$. Figure 8.6 shows the effect of the modification to the results of the mean distance traveled against the number of steps taken under the same conditions as shown in Figure 8.5.

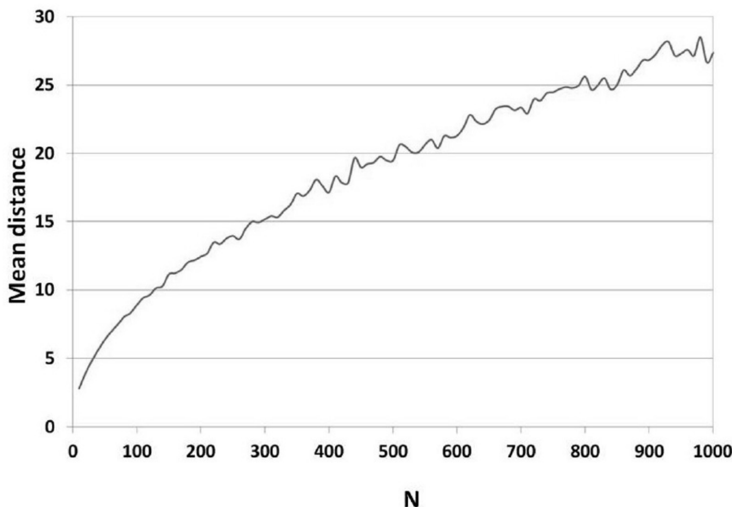

FIGURE 8.6: Mean distance travelled versus number of steps taken for a random walk on a 2d surface where any direction is possible.

We can also remove the assumption that the stride length is a constant by instead of randomly sampling the angle of travel, we randomly sample two numbers per step on the interval $[-1,1]$ that represent the x and y components of a displacement vector. Note that we could also do this by uniformly sampling theta on $[0,2\pi]$ and uniformly sampling the stride length, r, on the interval $[0,1]$. However, these two methods are subtly different; can you spot why, and does this affect our results in any way?

What your results should have told you is that the qualitative result is unaffected by the size of the strides. In other words, we have scale invariance, and we can simply set stride length to unity. The total mean distance traveled is then measured in units of the stride length. In technical parlance, our stride length is what is known as the mean free path.

Now that we have covered a random walk over a two-dimensional surface let us try to extend that to a random walk in a three-dimensional volume. In this case, it is useful to think of a perfume molecule diffusing through the air; the randomness, or stochastic process, is introduced by the perfume molecule randomly scattering from collisions with the molecules that make up the air. If we assume that the perfume molecule is equally likely to scatter in any direction, then we have spherical symmetry.

Figure 8.7 shows the spherical coordinate system (r,θ,φ), where r is the length of some position vector, θ is the polar or inclination angle, that is the angle between the z-axis and the position vector, and φ is the azimuth angle, that is the angle between the x axis and the projection of the position vector on to the x-y plane. As we have discussed, the mean free path can be set to a constant that is equal to one, that is, $|r| = 1$. We, therefore, need to uniformly sample random numbers on the surface of the unit sphere.

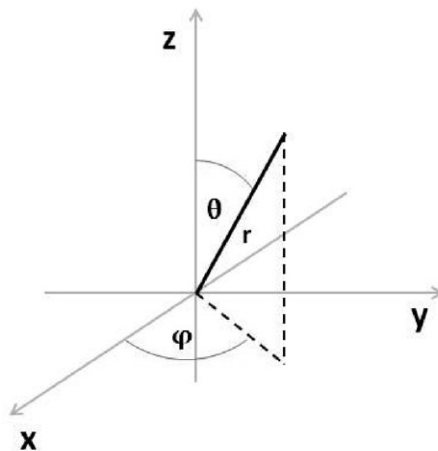

FIGURE 8.7: Spherical coordinate system.

If we naively sampled θ from $[0,\pi]$ and sampled φ over $[0,2\pi]$ both with uniform distributions, we would run into problems. The issue stems from the fact that the surface of a sphere is curved and can explain as follows. In our two-dimensional representation of the problem the azimuth angle, φ, can be sampled uniformly on the interval $[0,2\pi]$ as this leads to a uniform distribution of points on the

circumference of the (unit) circle; see Figure 8.8(a). Now consider Figure 8.8(b). This shows a sphere cut by the x-y plane such that a circle can be drawn around the sphere, which defines its equator. If we now lift the x-y plane up through the sphere, as if decreasing the polar angle θ by uniform increments, the circle defining where the x-y plane cuts the sphere necessarily contracts. If there were a uniform distribution of points on the circle, then as the circle moves up the sphere those points also contract. In other words, the distribution of points on the *surface* of the sphere would be not uniform and in fact, would bunch at the poles.

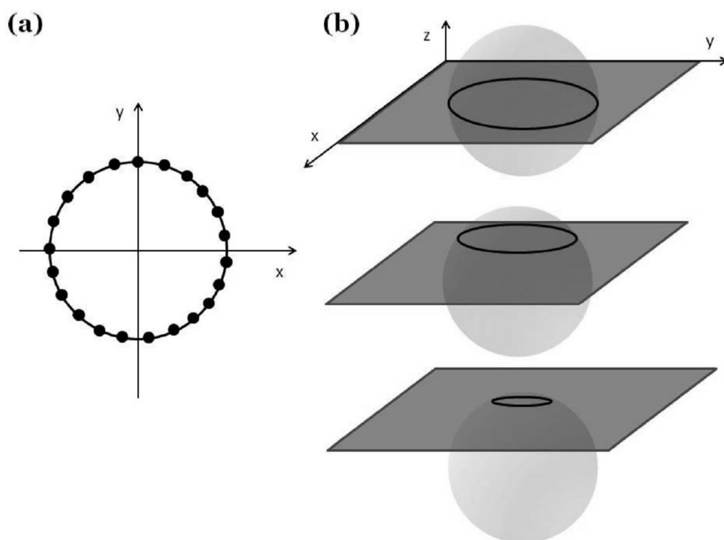

FIGURE 8.8: (a) Points on the unit circle defined by a random sampling of φ on the uniform distribution $[0,2\pi]$. (b) Unit sphere cut by the x-y plane; the circle defining the intersection deminishes as the plane is pulled upwards.

The way around this problem is not to randomly sample numbers for θ on a uniform distribution but to sample points on a $\sin(\theta)$ distribution. We can see this from the figure; we need less points at the poles where $\theta = 0$ and π, and more points around the equator where $\theta = \pi/2$.

More formally, we consider the solid angle element

$$d\Omega = \sin(\theta) d\theta \, d\phi. \tag{8.14}$$

We would like to uniformly sample on the element $d\Omega$, which means that we need to uniformly sample both $\sin(\theta)d\theta$ and $d\varphi$. If we let

$$dg = \sin(\theta)d\theta, \tag{8.15}$$

then we can write

$$g(\theta) = \cos(\theta). \tag{8.16}$$

We now have the means to uniformly sample the surface of the (unit) sphere. We select φ from a uniform distribution on the interval $[0, 2\pi]$ and select g from a uniform distribution on the interval $[-1, 1]$, where θ is then calculated using

$$\theta = \cos^{-1}(g). \tag{8.17}$$

Modify your program so that it can perform this three-dimensional simulation of a molecule diffusing through the air; you will have to calculate the x, y, and z coordinates of the displacement vector using trigonometry. Again, we should be able to show a relationship between the mean distance traveled and the number of collisions taken, so long as these results are statistically valid. Once you have the plot it should be the same as those we have previously presented. How might we show that the mean distance traveled is a power of N, and how might we discern that power?

These data *suggest* that we have stumbled upon a principal property of nature, that the average distance traveled of a particle undergoing random scattering events is proportional to the square root of the total number of scattering events to which it has been subjected. Mathematically, we write

$$\frac{R}{\lambda} \approx \sqrt{N}. \tag{8.18}$$

Note the use of the operative word suggest. We have not proved anything only that we have statistically valid results. What our computations have given us is a significant insight into the physics of diffusion, and a strong suggestion that the relationship described by Equation (8.18) probably is real.

This outlines the importance of Monte Caro simulations; they can provide qualitatively valid results for physical systems that may be difficult or impossible to solve using more direct methods (both analytical and numerical).

That said there is more information we can extract from our simulations that lends credibility to the results we obtained. By simulating many molecular paths, we have actually modeled the situation whereby a bottle of perfume has been opened and the many molecules of the aroma have diffused into the air (under the assumption that they all emerged from a point source located at the origin). In which case, we should be able to visualize the distribution of aromatic molecules as a function of distance from the perfume bottle after a particular number of collisions. Essentially, this represents the density of aromatic molecules in the air as we move away from the perfume bottle at a particular moment in time. Before you calculate the distribution how might you expect it to look, and how might you expect it to evolve with time (number of collisions)?

8.2.2 Radioactive Decay

Radioactive decay occurs when an unstable atom (or particle) releases some form of radiation (alpha, beta, and or gamma) and decays into other particles. This is also referred to a spontaneous decay, in that it requires no external stimulation to occur. Each unstable atom has the same probability to decay in any given period, but when this specifically happens is random. As the total number of unstable atoms decreases the number that decays in a particular period also decreases. In the limit of the period going to zero, we can say that the rate of decay is proportional to the number of unstable atoms that still exist. Thus, when there are many unstable atoms in a sample spontaneous decay is well modeled by an exponential decay. Essentially, this is a continuous or large number approximation to the actual process of the discrete decay events. As the number of unstable atoms inevitably decreases this approximation begins to fail and the process becomes increasingly stochastic (subject to chance).

In the limits of $N \to \infty$ and $\Delta t \to 0$ we write

$$\frac{\Delta N(t)}{\Delta t} \to \frac{dN(t)}{dt} = -\lambda N(t), \tag{8.19}$$

where λ is the so-called decay constant. This is related to the half-life of the unstable atoms by

$$T_{1/2} = \frac{\ln(2)}{\lambda}, \tag{8.20}$$

and λ can be described as the activity of the unstable atoms i.e., the probability that an atom decays within a given period of Δt.

We can use Monte Carlo simulation to model when this change in behavior occurs. More precisely, we can determine the minimum number of unstable atoms required for the large number approximation to hold true. To do this we increase time in discrete steps, and for each of these steps, we count the number of decay events that occurred during that interval. By keeping track of the number of atoms left in our simulation we can quit once they have all decayed. To simulate a decay event, we generate a random number, and if that number is less than λ then a decay event occurs, and we drop our atom count by one.

Unless we are comparing our computational results to actual experimental data, we can ignore any time scale that may be required. For instance, if $\lambda = 0.7 \times 10^4 s^{-1}$ then we should set our time intervals to be equal to $10^{-4} s$ so that we can set $\lambda = 0.7$ in our program and are able to use random numbers in the range $[0,1]$. Otherwise, we keep the value of λ as is and scale our random numbers accordingly; $\lambda = 0.7 \times 10^4 s^{-1}$ and the random numbers are scaled to $\left[0,10^4\right]$. In our time scale-free program, the increments in time can be equal to one, and λ is chosen somewhere between zero and one.

The file *nuclearDecay_ocv.cpp* contains a program that performs this simulation. The parameters in this file should be self-explanatory. Make sure the code is understood as it is used in one of the exercises that follow.

EXERCISES

8.1. Evaluate the integral

$$I = \int_0^{\pi} \sin(x)dx$$

using the Monte Carlo method with importance sampling, where $g(x) = x$. Compare the results to the same integration without this important sampling. What went wrong? Retain an additional term from the Taylor series expansion of sine and try again. Rather than retaining more terms in the Taylor series is there any other way to approximate sine over this interval?

8.2. The true period of a pendulum that is, with no small-angle approximation, of length l in a gravitational field of strength g is given by

$$T = 4\sqrt{\frac{l}{2g}} \int_0^{\theta_0} \left(\cos(\theta) - \cos(\theta_0)\right)^{-0.5} d\theta$$

where θ is the angle with the vertical, and θ_0 is the initial angle of release. Using a Monte Carlo method of integration, determine for what angles the small-angle approximation is valid. (Hint: You will need to recall/research the period for the small-angle approximation and decide what is accepted as valid.)

8.3. Confirm the random walk plots presented in the "Simulation" section. Can the noise in these results be reduced? Are there any analytical results that the mean free path of a particle, undergoing random scattering, is proportional to the square root of the number of scattering events?

8.4. Using the *nuclearDecay* program:

a. For large N (> 1000) check that N is proportional to the actual decay rate i.e., the number of decay events per time step.

b. Produce a plot (hint: logarithm) that shows that spontaneous decay looks initially exponential-like but as N decreases look increasingly stochastic in behavior. Approximately determine where the behavior changes. As a check, is the slope equal to λ?

c. Is this change in behavior independent of the initial number of atoms, and the decay constant used?

CHAPTER 9

PARTIAL DIFFERENTIAL EQUATIONS

Most of the interesting equations in physics are partial differential equations (PDE). Nearly all measurable quantities in the universe vary both in space and time; a fact that is reflected by the abundance of second-order PDEs in physics that have both space and time as independent variables. In general, we call functions whose values vary in both space and time (or with any other independent variable) a field. Name any topic in physics and it will likely have a PDE describing its phenomena; examples include but are not limited to Poisson's equation, the diffusion equation, the wave equation, the Helmholtz equation, the continuity equation, the Navier–Stokes equation, and the Schrödinger wave equation.

To solve PDEs analytically we must employ specific techniques such as the separation of variables or through the application of the Fourier Series. Where this is difficult or not even possible, the problem might be simplified, or special cases considered, whereby the equations can be reduced to an ordinary form. However, the most general approach to solving PDEs is by numerical methods.

This chapter refers to the Fortran code written for an earlier version of this book. It can be found on GitHub at: *github.com/DJWalker42/ComputationalPhysicsFortran*. The README explains how to compile Fortran programs.

To develop these numerical methods for solving PDEs, we must first return to ODEs and show how to write these as *finite difference equations*.

9.1 CLASSES, BOUNDARY VALUES, AND INITIAL CONDITIONS

For second-order PDEs, these names are analogous to the conic sections of the same name and are about the properties of their solutions. Figure 9.1 illustrates the conic sections.

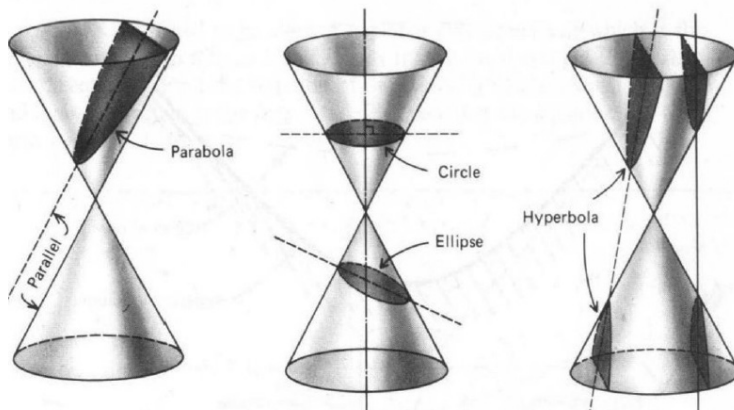

FIGURE 9.1: Conic sections: Parabolic, circular, elliptical, and hyperbolic. [Image copied from: http://www.andrews.edu/~calkins/math/webtexts/numb19.htm].

Just as an *ellipse* is a smooth, rounded shape, solutions to *elliptic* PDEs also tend to be smooth and rounded. Elliptic PDEs generally arise from physical problems that involve diffusion processes that have reached some equilibrium, for example, a steady-state temperature distribution. The *hyperbola* is the disconnected conic section. By analogy, *hyperbolic* PDEs can deal with discontinuities in the solution, for example, a shock wave or pulse, or some instantaneous increase in temperature. Hyperbolic PDEs usually occur in relation to mechanical oscillations, such as a vibrating string. Mathematically, *parabolic* PDEs serve as a shift from the hyperbolic PDEs to the elliptic PDEs. Physically, parabolic PDEs crop up in time-dependent

diffusion problems, such as the transient flow of heat in a conductor, say, before it reaches a steady-state.

Consider the most general form for a second-order PDE with two independent variables

$$A\frac{\partial^2 U}{\partial x^2}+2B\frac{\partial^2 U}{\partial x\partial y}+C\frac{\partial^2 U}{\partial y^2}+D\frac{\partial U}{\partial x}+E\frac{\partial U}{\partial y}+FU=G, \tag{9.1}$$

where A through G are arbitrary functions (that can be constant) of the variables x and y, and U is some physical field. Note that the second term has a factor of 2 as for any partial derivative

$$\frac{\partial^2 U}{\partial x\partial y}=\frac{\partial^2 U}{\partial y\partial x}, \tag{9.2}$$

where the left-hand term is the second-ordered derivative taken with respect to x first then y, and the right-hand term is the second-ordered derivative taken with respect to y first then x. Remember that with partial differentiation all other independent variables are considered constant when performing the operation with respect to a particular variable.

We can define the discriminant of Equation (9.1) as the following

$$d \equiv B^2 - AC. \tag{9.3}$$

When $d<0$ the PDE is elliptic, $d=0$ the PDE is parabolic, and $d>0$ the PDE is hyperbolic.

Poisson's equation in two dimensions is given by

$$\nabla^2\phi\left(x,y\right)=-4\pi\rho\left(x,y\right), \tag{9.4}$$

where $\nabla^2 \equiv \Delta$ is the Laplace operator that has the form

$$\Delta=\sum_i \frac{\partial^2}{\partial x_i^2} \tag{9.5}$$

with x_i representing the spatial coordinates. Here φ is a (scalar) electrical potential field and ρ is a charge density. Comparing Equations (9.4) with (9.1), we see that $A=C=1$ and $B=0$, thus the discriminant is negative, and Poisson's equation is elliptic.

The heat equation (or more generally the diffusion equation), in one spatial dimension, is given by

$$\alpha \frac{\partial^2 T(x,t)}{\partial x^2} = \frac{\partial T(x,t)}{\partial t}, \tag{9.6}$$

where T is some temperature field and α is known as the diffusion constant. In the case of heat flow $\alpha = K / C\rho$, where K is the thermal conductivity, C is the specific heat, and ρ is the density of the material through which the heat flows. Here $A = \alpha$ and $B = C = 0$ thus the discriminant is zero and the heat equation is parabolic.

The wave equation, in one spatial dimension, is given by

$$c^2 \frac{\partial^2 \psi(x,t)}{\partial x^2} = \frac{\partial^2 \psi(x,t)}{\partial t^2}, \tag{9.7}$$

where ψ represents the displacement of the wave from some equilibrium, say, and c is the speed of the wave. Here $A = c^2$, $C = -1$, and $B = 0$ making the discriminant positive and thus the wave equation is hyperbolic.

As with ODEs we need to know some initial values in order to determine a particular or unique solution to the PDE we are studying. For example, to obtain a particular solution of the (second-order) ODE governing simple harmonic motion we needed to know the initial position and the initial velocity of the body. For second-order PDEs, we still require two pieces of information, but in this case, they are the initial state or condition of the entire system, and the behavior of the system at its boundaries. To illustrate this concept, imagine a metal rod that is held at a constant $0°C$ at one end, and $100°C$ at the other, these are the boundary values. Do we have sufficient information to determine the temperature distribution of the rod as it evolves with time? We can certainly guess the steady-state distribution, it will be linear over the length of the rod increasing from $0°C$ to $100°C$. However, to determine the transient behavior we need to know the initial temperature distribution of the rod that is, its initial condition. Generally, we write the initial condition as

$$U(x_i, t = 0) = U_0(x_i), \tag{9.8}$$

where U is some physical quantity, x_i represent spatial coordinates, t represents time, and it follows that U_0 is the initial state of the system.

The type of *boundary conditions* that we have used in the example above is known as *Dirichlet* conditions; the *value of the function* (temperature in our example) at the boundaries of the system domain (the ends of the rod) is known. Mathematically, we write Dirichlet conditions as

$$U(x_i = \Omega_i, t) = f, \tag{9.9}$$

where f represents the value of the quantity U on the boundary Ω_i. There are other types of boundary conditions. *Neumann* boundary conditions are when we know the *normal derivative of the function* at the boundaries. In the example of heat conduction through a rod, Neumann boundary conditions relate to the flux or heat flow across the ends of the rod. Mathematically we write Neumann conditions as

$$\frac{\partial U(x_i, t)}{\partial x_i} \Big| x_i = \Omega_i = g, \tag{9.10}$$

where g represents the flux or flow of quantity U across some boundary $x_i = \Omega_i$. Note that in the most general cases f and g can be known functions of x_i and t but to keep things relatively simple we will only consider the case where they are constants. For example, if heat flows across one end of our metal rod, then physics tells us there must be a temperature gradient across that boundary, that is, g is some nominal, constant value with units of $°C$ / m. Additionally, the sign of g tells us whether the heat flows into or out of the rod.

Cauchy boundary conditions are when we know both the value of the function and its normal derivative on the *same* boundary. For some problems, a Cauchy boundary condition will *over-specify* the PDE, and no unique solution will exist. As a rule of thumb Cauchy conditions are typically associated with hyperbolic PDEs (like the wave equation), whereas Dirichlet and Neumann conditions are more appropriate for elliptic and parabolic PDEs. Note, however, that the precise boundary conditions will depend on the physics being modeled, and PDE we are solving. Sometimes this requires that we have

mixed boundary conditions; *different boundary conditions* are used on *different parts* of the boundary of the system. For example, if our metal rod is held at a constant temperature at one end, a Dirichlet condition, but is insulated at the other end, a Neumann condition (g = 0), then we have mixed conditions.

Before we leave this discussion of the generalities of PDEs it is worth mentioning some convenient shorthand notation for expressing partial derivatives. We make the following adjustments

$$\frac{\partial U(x_i,t)}{\partial x_i} = U_{x_i}, \tag{9.11}$$

$$\frac{\partial U(x_i,t)}{\partial t} = U_t, \tag{9.12}$$

$$\frac{\partial^2 U(x_i,t)}{\partial x_i^2} = U_{x_i x_i} \tag{9.13}$$

and so on. Although we rarely come across the mixed, second-ordered derivative, for completeness

$$\frac{\partial^2 U(x_i,t)}{\partial x \partial t} = U_{x_i t} \tag{9.14}$$

and remember that $U_{xt} = U_{tx}$.

Thus, Equation (9.1) at the start of this section can be rewritten as

$$AU_{xx} + 2BU_{xy} + CU_{yy} + DU_x + EU_y + FU = G. \tag{9.15}$$

9.2 FINITE DIFFERENCE METHODS

As stated in Chapter 5, an approach to solving second-order ODEs involved generating an auxiliary variable and separating the ODE into a pair of, coupled first-order ODEs. While this perfectly good method for many equations found in physics it can be difficult, if not impossible, to apply to PDEs.

This section refers to LAPACK (Linear Algebra PACKage) which is a library of subroutines, written in Fortran, that solve linear algebra problems. Interfacing, C++ (or other high-level languages) with Fortran is possible but contains many pitfalls. Alternatively,

Eigen is an entirely C++ template library for linear algebra and can be used instead of LAPACK. Notice that the matrix factorizations developed in the C++ code written for this book do not deal with tridiagonal matrices and will be relatively slow at solving the problems discussed throughout this chapter.

9.2.1 Difference Formulas

Another approach to solving boundary value problems is to approximate the differential equation with a difference equation. For now, we will only consider differential equations with one independent variable, that is, ODEs, and will later show how to extend this to PDEs. We can obtain the difference equation by considering the definition of the differential equation, which is

$$\frac{\Delta f(x)}{\Delta x} = \lim_{h \to 0}\left(\frac{f(x+h)-f(x)}{h}\right) = \frac{df(x)}{dx}. \tag{9.16}$$

Thus, an appropriate approximation to the derivative is

$$f'(x) \approx \frac{f(x+h)-f(x)}{h}, \tag{9.17}$$

where h is some small but finite value.

To determine the error behavior of this approximation we could compare it to the analytical solution of some known ODE. However, we can do a much better job by using the mathematical tools at our disposal. As some of you may have already guessed, we can obtain Equation (9.17) using the Taylor series expansion of a function at $f(x+h)$ about $f(x)$, that is,

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x). \tag{9.18}$$

This can be rearranged for the first-order derivative such that

$$f'(x) = \frac{1}{h}\left[f(x+h)-f(x)-\frac{h^2}{2!}f''(x)-\ldots\right]. \tag{9.19}$$

Notice the equivalency, Equation (9.19) is not an approximation but an exact formula for the first-ordered derivative assuming an infinite number of terms. Comparing Equation (9.17) with Equation (9.19), we see that the approximation for the derivative is the

Taylor series expansion truncated after the second term. Therefore, the error in the approximation is $\mathcal{O}(h)$ (note the multiplicative factor of $1/h$) and we can write the equivalency

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h). \tag{9.20}$$

This is called the forward difference approximation to the derivative as we are using a point that is forward one step, $f(x+h)$, to approximate the derivative at the current position, $f(x)$.

A backward difference is derived similarly such that

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \mathcal{O}(h), \tag{9.21}$$

where we use a step behind, $f(x-h)$, to approximate the derivative at the current position, $f(x)$. This also has $\mathcal{O}(h)$ accuracy. For convenient notation, let us make the following changes: if our current position is given by $f(x) \to f(x_i) \to f_i$, the forward position is then given by $f(x+h) \to f_{i+1}$, and the backward position is given by $f(x-h) \to f_{i-1}$. Both the forward and backward difference approximations are two-point formulas.

An advantage of deriving these expressions from their respective Taylor series is that we notice the leading error term in both has the same magnitude but opposite sign. Thus, if we add Equations (9.20) and (9.21) this leading error term will cancel, and we should obtain a more accurate approximation.

After performing the necessary steps, we find that

$$f_i' = \frac{f_{i+1} - f_{i-1}}{2h} + \mathcal{O}\left(h^2\right). \tag{9.22}$$

This is called the central difference formula and is a three-point formula as we use both the forward, $i+1$, and backward, $i-1$, values to approximate the derivative at our current value, i. You can think of the formula as containing an f_i term but with a zero coefficient. Note that this formula takes a symmetrical "picture" at the local neighborhood of the point of interest, whereas the forward and backward formulas only use the information to one side of the point of interest.

We could keep improving the error behavior by taking more points around our point of interest. For example, a five-point formula can be derived in a similar manner to the three-point formula above yielding

$$f_i' = \frac{1}{12h}\left[f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}\right] + \mathcal{O}\left(h^4\right). \tag{9.23}$$

Notice that we have an error behavior of $\mathcal{O}\left(h^4\right)$. In general, an n point central difference formula will be $\mathcal{O}\left(h^{n-1}\right)$ accurate. While we could just keep increasing the number of points in the approximation for the derivative to improve its accuracy this does have a practical limit. These approximations require that we know values for the function both ahead and behind our current position. Unlike the Euler or Runge–Kutta methods that are self-starting formulas, multi-step methods require that we initiate them in some way. Typically, this involves using a Runge–Kutta method, say, to provide the required number of points from the boundary conditions to start the multi-step formula. As any error incurred during this initiation stage will be propagated throughout the integration, the initiation method needs to be at least the same order of accuracy as the multi-step formula it is starting. As the values of the function and its derivative(s) on and close to the boundaries of a real physical system are usually important (if not crucial) to the outcome of the integration, they must be calculated accurately.

Note that should we be approximating the derivatives of a *known function* on a particular interval we can use an interpolation method to *extrapolate* the derivatives at the limits (boundaries) of the interval. For example, let us imagine we have some function, $f(x)$, stored in an array of size N such that we have function values at equidistance increments, h, over x. We then approximate the first-ordered derivative using the five-point formula, say. Necessarily, we require two extra points both at the start and the end of the x line-space. To obtain the missing values for the derivative we extrapolate from those we have calculated in the main body of the array. As we have used a five-point finite difference method we should use an interpolation method that matches the accuracy order, for example, a four-point (third ordered) Lagrange polynomial interpolation scheme which has a quartic error behavior.

This is not the only method we could use. As we have computed function values in the interior of the array, we could apply a forward and a backward difference formula at the start and at the end of our array, respectively, to approximate the derivative(s) at those points. Keep in mind that these methods to deal with edge or boundary values must match the error behavior of the method applied to the interior values of the function.

The more practical (and obvious) way of increasing the accuracy of our approximation formulas is to decrease the step size h. Of course, this requires more computational effort but as with most things, there is always a cost-reward trade-off. And, as we shall see shortly, knowing how the error behaves as the step size decreases can be used to our benefit. That said what happens if we let $h \to 0$ in a computer program that calculates *finite differences*?

Higher-order differential equations can also be approximated with a difference equation. Again, they are derived using the Taylor series expansion of various points about the point of interest. The central difference formula for the second-ordered derivative is

$$f_i'' = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} + \mathcal{O}\left(h^2\right), \tag{9.24}$$

which is a three-point formula. The five-point central difference formula for the second-ordered derivative is given by

$$f_i'' = \frac{1}{12h^2}\left[-f_{i-2} + 16f_{i-1} - 30f_i + 16f_{i+1} - f_{i+2}\right] + \mathcal{O}\left(h^4\right). \tag{9.25}$$

9.2.2 Application of Difference Formulas

How do we go about applying these formulas to a particular differential equation? To illustrate, consider the following general, second-order ODE

$$\alpha f'' + \beta f' + \gamma f = \delta x, \tag{9.26}$$

where α through δ are constants and is subject to the boundary conditions

$$f(a) = c,\ f(b) = d, \tag{9.27}$$

where $[a,b]$ defines the computational domain (integration interval), and c and d are the values of the function at the boundaries.

The first step is to partition our computational domain into a grid or mesh of discrete points. For simplicity, we make this grid uniform by defining

$$h = \frac{b-a}{N-1}, \tag{9.28}$$

where N is the *total* number of grid points; we include both the boundaries in our grid. We then approximate the continuous, differential equation with a discrete, difference equation to find the solution on the grid we have just imposed. Substituting the three-point central difference formulas into Equation (9.26), we arrive at the following finite difference equation for some arbitrary interior point x_i

$$\frac{\alpha}{h^2}\left(f_{i+1} - 2f_i + f_{i-1}\right) + \frac{\beta}{2h}\left(f_{i+1} - f_{i-1}\right) + \gamma f_i = \delta x_i. \tag{9.29}$$

We then solve Equation (9.29) on the $N-2$ interior grid points; the boundary values are fixed, that is, $f(a) \equiv f_1 = c$ and $f(b) \equiv f_N = d$. This means we have a system of $N-2$ linear equations that can be solved simultaneously for the $N-2$ unknown interior grid points. Note that the solution on the discrete grid will only approximate the solution of the original, continuous problem. That, of course, is the point; if we had access to an exact, analytical solution to the problem we should not need to approximate the problem in the first place.

There are now two ways to proceed with the solution of Equation (9.29). We either go for the direct method or the indirect method. The direct method involves recasting Equation (9.29) in matrix form and solving the system by Gaussian elimination, say. To do this, we gather like terms such that

$$\varphi f_{i+1} + \theta f_i + \psi f_{i-1} = \delta x_i, \tag{9.30}$$

where

$$\varphi = \frac{\alpha}{h^2} + \frac{\beta}{2h},$$

$$\theta = \gamma - \frac{2\alpha}{h^2}$$

and

$$\psi = \frac{\alpha}{h^2} - \frac{\beta}{2h}.$$

Perhaps not immediately obvious but Equation (9.30) has the matrix form of

$$A\underline{f} = \delta\,\underline{x}, \tag{9.31}$$

where A is an $(N-2)$-by-$(N-2)$ tridiagonal matrix given by

$$A = \begin{bmatrix} \theta & \varphi & 0 & \cdots & 0 \\ \psi & \theta & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \theta & \varphi \\ 0 & \cdots & 0 & \psi & \theta \end{bmatrix}; \tag{9.32}$$

$\underline{f}$ is an $N-2$ vector representing the unknown function values at the interior grid points, explicitly

$$\underline{f} = \begin{bmatrix} f_2 \\ f_3 \\ \vdots \\ f_{N-2} \\ f_{N-1} \end{bmatrix} \tag{9.33}$$

and $\underline{x}$ is an $N-2$ vector representing the known, discrete grid of points on the independent variable x, explicitly

$$\underline{x} = \begin{bmatrix} x_2 \\ x_3 \\ \vdots \\ x_{N-2} \\ x_{N-3} \end{bmatrix}, \tag{9.34}$$

where $x_i = a + (i-1)h$ for $i = 2, \ldots, N-1$.

Equation (9.31) can be readily solved by the LAPACK subroutine "DGTSV," say, which solves a general tridiagonal system of linear equations by Gaussian elimination with partial row pivoting.

The indirect method of solution is to find an equation for f_i in terms of its neighboring points and find an approximation that satisfies the difference equation at the grid points using an iterative technique. That is, we start with an initial guess for f at all grid points and using the appropriate equation we iterate to a more accurate approximation.

We express our iterative scheme by rewriting Equation (9.30) to solve for f_i such that

$$f_i^{(n)} = -\frac{1}{\theta}\left(\varphi f_{i+1}^{(n-1)} + \psi f_{i-1}^{(n-1)} - \delta x_i\right), \tag{9.35}$$

where n is an index that represents the level of iteration, not to be confused with as a power. Thus, our initial guess is given by the index $n = 0$ and we iterate to the next level, $n = 1$, through the application of Equation (9.35) at all interior grid points, remembering that the boundaries are fixed. Note that we need to have two storage arrays; one to store the grid point values at the current level of iteration, and one to store the grid point values at the next level of iteration. Note also that the scheme can be described as "red-black" in reference to a chessboard pattern of the same colors. To explain this, take an even value for the grid point index i. Note that the next level of iteration depends only on the adjacent, odd values of the current iteration level (the grid points x_i remain constant throughout the iteration scheme). This means that, for each iteration level, the computations for even and odd grid points can be done independently. Thus, if we think of the iteration scheme as a red and black chessboard, where the squares represent the grid points and the columns represent the iteration level, then red squares only influence other red squares, and black squares only influence other black squares. This property lends itself well to parallel computing, but more on this in a later chapter. The iteration method we have just outlined is called the *Jacobi* scheme and will converge to the exact solution.

Parallel computing aside, we can apply a little thought to the Jacobi scheme and come up with a similar method but with quicker

convergence. Imagine we have just determined $f_i^{(n)}$ and are ready to compute the next grid point $f_{i+1}^{(n)}$. The Jacobi scheme would have you use the value $f_i^{(n-1)}$ at the previous iteration level despite the fact we have just calculated an improved value for that grid point. Instead, let us use that improved value. Equation (9.35) then becomes

$$f_i^{(n)} = -\frac{1}{\theta}\left(\varphi f_{i+1}^{(n-1)} + \psi f_{i-1}^{(n)} - \delta x_i\right). \tag{9.36}$$

Note that this formula represents moving through the array with ascending *i* values. For descending *i* values the iteration index levels on the right-hand side swap accordingly. This iteration method is called the *Gauss–Seidel* scheme and it will converge to the exact solution more quickly than the Jacobi scheme. However, note that we have now lost the red-black property of the Jacobi scheme, making it more of a challenge to write the Gauss–Seidel scheme in parallel code.

Iterative methods are generally inferior to direct methods when applied to ODEs. By this, we mean that direct methods can supply us with a solution with far less computational effort than that of an iterative scheme to the same level of accuracy. Iterative methods come into their own when applied to physical problems that involve several independent variables, in other words, PDEs. This is in part due to the differences in the propagation of error between the two methods. Direct methods rely on matrix factorization, typically Gaussian elimination, with the solution found by back substitution. Any error in one value is passed to all values that follow it in the substitution, leading to a non-uniform distribution of error in the numerical solution. As we add more independent variables this accumulation of error tends to worsen. For iterative methods, the error tends to get smeared out across the entire computational domain (across each independent variable) leading to a more uniform distribution. Additionally, the error in an iterative method can always be improved by simply iterating further, whereas the error in a direct method cannot be improved unless applying Richardson extrapolation (see the next section), for example.

When writing code for an iteration scheme we must bear in mind that *all* grid points must reach a specified level of accuracy before

we can say the iteration has converged. That is, we check that the difference between the same grid points at different iteration levels is less than some tolerance, for all the grid points in the computational domain. This is easily accomplished by using a logical variable which is set to true (or false) at the start of each iteration, then set to false (or true) should *any* of the grid points fail the accuracy test, that is, it only takes one to fail for the convergence check to fail. If the convergence check has failed, then we go to the next level of iteration. The tolerance itself should be set relatively large (depends on the problem) as the solution we obtain is only relevant to the discrete, finite difference approximation we made of the continuous, differential equation. Of course, the accuracy of our approximate problem can be improved by making the grid finer, in which case we could also apply Richardson extrapolation.

Though convergence is (pretty much) guaranteed it can be slow, thus we should provide a count of the number of iterations and have the program exit should we go beyond some maximum; this is true of any iteration method.

The file *gauss-seidel.f90* contains the program code to implement the iteration method of the same name. You may find it useful to add some code to this program to have it print out some measure of the error between iterations should the method fail to converge before reaching the maximum iteration count. This may give you some feel for how close the method was to convergence and allow you to adjust either the tolerance or maximum iteration count appropriately.

Both the Jacobi and Gauss–Seidel schemes are what is known as relaxation techniques. That is, we derive the finite difference approximation to the differential equation, guess at a solution, and the method relaxes that guess to the exact value. How quickly the method relaxes the solution to the "exact" value depends upon how good the initial guess was in the first place. However, no matter how good or bad the initial guess the method will eventually converge on a significantly accurate approximation and the relaxation becomes what is called monotonic. When a function is monotonic it only ever increases or decreases with its independent variable; there are no oscillations, inflections, or stationary points. In this case, the

relaxation takes the solution closer to the true value at all grid points after each iteration loop. This suggests that if we take a weighted average between the iterated value we have just calculated and the previous iterated value that we should obtain, a more accurate solution for that grid point. Mathematically, this is

$$f_i^{(n)} = \omega \overline{f}_i^{(n)} + (1-\omega) f_i^{(n-1)}, \tag{9.37}$$

where ω is the weighting factor (sometimes referred to as the extrapolation or relaxation factor) and lies on the interval $[0,2]$, and $\overline{f}_i^{(n)}$ is the Gauss–Seidel value. Equation (9.37) is called *over-relaxation* and as we apply it successively at each iteration level the entire method is referred to as *Successive Over-Relaxation*.

For each individual differential equation, and hence derived finite difference approximation, there will be an optimal value for ω that gives the most rapid convergence. This optimal value can be calculated in some cases and estimated in others.

Currently, we have only developed finite difference approximations to ODEs but the extension to PDEs is not so difficult.

9.3 RICHARDSON EXTRAPOLATION

Richardson extrapolation is stated previously in the sections on developing adaptive step numerical integration (quadrature) methods, the adaptive step ODE solvers, and most recently in the section above. Here, we discuss the general technique and how it relates to computations of all sorts.

Richardson extrapolation is what is known as a sequence acceleration method. It is named after Lewis Fry Richardson, who introduced the technique in the 1920s for use in predicting the weather via numerical techniques. In essence, the technique uses a single formula with known error behavior, which is computed using different step sizes, and the results are combined to eliminate the leading error term in the sequence. To explain how it works in detail let us first apply it to the method of numerical differentiation and then generalize for any method where the error behavior is known.

We know from the Taylor series expansion that the first-ordered derivative of any function can be given by

$$f_i' = \frac{f_{i+1} - f_i}{h} - \frac{h}{2!} f_i'' - \ldots, \tag{9.38}$$

so that the approximation to the derivative is given by

$$A_0(h) = \frac{f_{i+1} - f_i}{h}, \tag{9.39}$$

with an $\mathcal{O}(h)$ error behavior. The subscript index on A refers to the extrapolation level; this will be explained shortly. Here we are using the specific example of the central difference formula. We could use any numerical technique that we have discussed thus far so long as we know its error behavior.

If we halve the step size, h, then the new approximation will have a leading term error that is half that of the previous approximation. Thus, we can eliminate the leading error term by subtracting the approximation using h from twice the approximation using $h/2$. The extrapolated approximation is then given by

$$A_1(h/2) = 2A_0(h/2) - A_0(h), \tag{9.40}$$

which has an $\mathcal{O}(h^2)$ error behavior as we eliminated the leading $\mathcal{O}(h)$ error term. If we halve the step size again and calculate $A_0(h/4)$ then we can apply the same technique to find

$$A_1(h/4) = 2A_0(h/4) - A_0(h/2), \tag{9.41}$$

with the same error behavior as before. As we know how the error behaves in the extrapolated approximations, $\mathcal{O}(h^2)$, and we have two measures of that error with differing step size we can eliminate the next leading error term by performing

$$A_2(h/4) = \frac{4A_1(h/4) - A_1(h/2)}{3}. \tag{9.42}$$

Here we have used the fact that the leading error term in $A_1(h/4)$ must be four times smaller than the leading error term in $A_1(h/2)$. Therefore, four times $A_1(h/4)$ minus $A_1(h/2)$ must

leave three times the exact answer plus the remaining error terms. The error behavior in Equation (9.42) is then $\mathcal{O}(h^3)$.

This process can continue indefinitely with a general recurrence relation

$$A_{m+1,l+1}(h) = \frac{t^{k_m} A_{m+1,l} - A_{m,l}}{t^{k_m} - 1} \tag{9.43}$$

where $m \leq l$,

$$A_{m,l} = A_l(h / t^m), \tag{9.44}$$

l is the extrapolation level, t is the factor we use to reduce the step size h (for practical purposes this is nearly always two), and k_m is an integer related to the step size reduction level m and the nature of the sequence we are extrapolating. For instance, the central difference formula, Equation (9.22), has only error terms involving even power terms of h, thus $k_m = 2m$. Conversely, if we had a method that involved only odd power terms of h then $k_m = 2m+1$. In the most general cases k_m need not be an integer but we would not consider those.

As Equation (9.43) is somewhat abstract let us apply it to an actual example. Consider the function $f(x) = \sin(x)$ with its first-ordered derivative approximated by the three-point central difference formula, Equation (9.22). We can put the approximations into matrix format with the step size reduction level, m, defining the rows and the extrapolation level, l, defining the columns; note that we start the level numbering at zero. Thus, we gain the following approximation matrix for the first two levels of extrapolation:

$$A = \begin{bmatrix} 0.52600907 & 0.00000000 & 0.00000000 \\ 0.53670749 & 0.54027363 & 0.00000000 \\ 0.53940225 & 0.54030051 & 0.54030230 \end{bmatrix}. \tag{9.45}$$

Here we assessed the derivative at $x = 1$, with an initial $h = 0.4$, $t = 2$, and we use the fact that $k_m = 2m$. The way this matrix equation fills is row by row. In other words, we compute a new level of extrapolation as soon as we have sufficient information to do so. For example, we first compute $A_{0,0}$ and $A_{1,0}$, then use those to calculate

$A_{1,1}$, before returning to the zeroth level of extrapolation to calculate $A_{2,0}$, and the procedure continues. In this way, the bottom right-most entry of A is the most accurate approximation to the derivative. The matrix A is called a lower triangular matrix for obvious reasons.

The first question you should be asking yourselves is "are we actually saving ourselves computational effort?". If say instead of approximating a derivative via a finite difference we were performing a numerical quadrature, such as the composite trapezoidal rule, then yes, we do save ourselves computational effort. To explain, the first column of A would contain the approximations from the trapezoidal rule calculated at the different step sizes (slice width). If the first approximation took N function evaluations, then to calculate the first column of A up to the second level of step size reduction would require $N + 2N + 4N = 7N$ function evaluations. Using these approximations, we can obtain a more accurate value up to the second level of extrapolation. At this level, the error behaves as $\mathcal{O}\left(h^6\right)$; the trapezoidal rule has no odd powers of h in its error terms. Hence, the error in the approximation at this level will be roughly a factor of $1/4^6$ times smaller than the first entry. To obtain this accuracy order by only reducing the step size would require us to use at least $1/64$ times the original step size. In other words, $64N$ function evaluations versus 7N function evaluations. The further we extrapolate the bigger (and the better) the difference between these two numbers.

However, for any finite difference formula, we only ever need to use a set number of evaluations to compute an approximation to the derivative at a given point. For instance, the three-point central difference formula requires two function evaluations, one at f_{i+1} and one at f_{i-1}, regardless of the step size. The extrapolation would appear to be wasting computational effort. However, if we have an unknown function taken from some measurement, say, and we wish to find its derivative we may not have enough points to analyze the absolute error in our approximation using different step sizes of the central difference formula alone. Richardson extrapolation would (hopefully) converge more rapidly on a precise approximation and give us means to estimate the absolute error in the numerical derivative. Secondary to this point is that as we reduce the step size the

function evaluations for the finite difference come closer together in size. Eventually, their difference becomes comparable to the machine precision, and we introduce round-off error. We can see this by applying the central difference formula to a known function and calculating the absolute error in the approximation for decreasing step size. The error reduction initially behaves as predicted but will eventually slow down and even begin to increase for sufficiently small step sizes due to the round-off error. By using Richardson extrapolation, we can avoid round-off error issues.

Typically, the extrapolation does not go much beyond the third or fourth level as by then we usually have an approximation that is within the desired tolerance level. One way to ensure the significant figure of the accuracy of the approximation is to subtract the previous level of extrapolation from the current level (at the same step size reduction level, i.e., the same row) and test it against the desired tolerance. For instance, in our example above subtracting element $A_{2,1}$ from $A_{2,2}$ we would see that $A_{2,2}$ is at least five significant figures accurate, that is, within the tolerance that we used of 10^{-4}. In fact, the approximation shown is 6 significant figures accurate.

The file *richardsonExtrap.f90* contains the code to perform the extrapolation using a matrix array A and a three-point central difference formula to approximate the first-ordered derivative of some user-defined function.

9.4 NUMERICAL METHODS TO SOLVE PDEs

To begin this discussion let us start with arguably the most intuitive of the PDE.

9.4.1 The Heat Equation with Dirichlet Boundaries

Consider the normalized (the diffusion constant is unity) heat equation in one dimension across a metal rod

$$U_t = U_{xx}, \tag{9.46}$$

where U is the temperature field, x is the spatial coordinate, and t is time. Let us impose the Dirichlet boundary conditions

$$U(0,t) = U(L,t) = c, \tag{9.47}$$

where L is the length of the rod, and c is a constant temperature. This is the model for a metal rod held at a steady temperature at either end. The initial condition of the rod is given by

$$U(x,0) = U_0(x). \tag{9.48}$$

One way to solve this equation numerically is to approximate all the derivatives by finite differences. We partition the domain (the length of the rod) in space using a grid or mesh $x_1,\ldots,x_M$ (hence M spatial grid points in total) and in time using a grid $t_1,\ldots,t_N$ (hence N time grid points in total). We assume a uniform partition both in space and in time such that the difference between two consecutive space points can be given by h, and between two consecutive time points can be given by k. The points

$$u(x_m, t_n) = u_m^n \tag{9.49}$$

represent the numerical approximation to the exact solution at the grid point (m,n). To avoid confusion a lower-case letter with two indices either written u_m^n or $u_{m,n}$ refer to the *grid points* of the *finite difference approximation*, the former being for one spatial variable and a time variable, and the latter being used for two spatial variables. An uppercase letter followed by two indices without a comma, for example, U_{xx}, is the second-ordered partial derivative of U with respect to the continuous variable x. Here we are using the convention that we write the discrete-time point index as a superscript; we are not taking a power. For clarity, if we had a two-dimensional heat conducting sheet, say, and we partitioned the sheet across x, y, and t then the conventional notation for a grid point on the temperature field U would be

$$u(x_l, y_m, t_n) = u_{l,m}^n, \tag{9.50}$$

where l and m are the indices for the spatial coordinate grid points, and n is the index for the temporal grid points.

9.4.1.1 *Explicit Method*

Using a forward difference at time t_n and a second-order central difference for the space derivative at position x_m we get the recurrence equation:

$$\frac{u_m^{n+1} - u_m^n}{k} = \frac{u_{m+1}^n - 2u_m^n + u_{m-1}^n}{h^2}. \tag{9.51}$$

This is an explicit method for solving the one-dimensional heat equation in that we can compute the advanced time step u_m^{n+1} from the previous values such that

$$u_m^{n+1} = (1-2r)u_m^n + ru_{m-1}^n + ru_{m+1}^n, \tag{9.52}$$

where $r = k/h^2$.

To input the boundary conditions, we must fix $u_0^n = c$ and $u_M^n = c$, over all N time steps.

To visualize Equation (9.52), we can consider what is called the computational or numerical stencil (or molecule) of the explicit method. Figure 9.2 shows this stencil placed at some arbitrary interior location, that is, not near the boundaries of the domain. Here we use the notion that known values, either computed or belonging to the initial or boundary conditions, are represented by filled squares, and the unknown values, those yet to be computed, are represented by open circles. The stencil shows us that we are using three known values, specifically u_{m-1}^n, u_m^n, and u_{m+1}^n to compute the unknown value u_m^{n+1}. As we run the computation we can think of the stencil as moving down one row at a time, calculating the unknown value at the advance time step, until it reaches the edge of the grid (the domain boundary), where it moves to the top of the next column and repeats the process, finishing at the bottom of the last time step.

The explicit method is known to be numerically stable and convergent when $r \le 1/2$. As we have taken a forward difference for the time differential, and a central difference for the space differential the error in the approximation can be written as $\Delta u = \mathcal{O}(k) + \mathcal{O}(h^2)$.

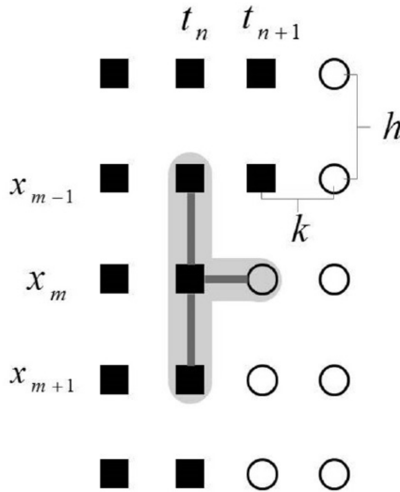

FIGURE 9.2: Explicit method stencil. Filled squares are computed points; open circles are to be computed.

9.4.1.2 *Implicit method*

If we use the backward difference at time t_{n+1} and a second-order central difference for the space derivative at position x_m we get the recurrence equation

$$\frac{u_m^{n+1} - u_m^n}{k} = \frac{u_{m+1}^{n+1} - 2u_m^{n+1} + u_{m-1}^{n+1}}{h^2}. \tag{9.53}$$

This is an implicit method for solving the one-dimensional heat equation in that to obtain the advanced time step t_{n+1} we compute u_m^{n+1} by solving a system of linear equations. We obtain that system of linear equations by rearranging Equation (9.53) and using the substitution $r = k / h^2$ to give

$$(1+2r)u_m^{n+1} - ru_{m-1}^{n+1} - ru_{m+1}^{n+1} = u_m^n. \tag{9.54}$$

The numerical stencil for the implicit method is shown in Figure 9.3. As we can see each equation contains one known value u_m^n and three unknown values u_{m-1}^{n+1}, u_m^{n+1}, and u_{m+1}^{n+1}. If there is a thought scratching at the back of your head, then remember that we move this stencil down one row at a time. Thus, for the grid points $m = 3,\ldots,M-2$ each unknown appears in three separate equations, hence we have sufficient information to compute the unknowns at

the advanced time step for those grid points. For instance, in Figure 9.3, we now have sufficient information to calculate grid point u_{m-1}^{n+1}. But what of grid points $m = 2$, and $m = M - 1$? They only appear in two separate equations, it looks as if we are missing two known values, one for each grid point. Of course, the answer lies in the boundary values. If we place our implicit method stencil at $m = 2$, say, then one of the "unknowns" is a boundary value, which is a known value. The same can be said at $m = M - 1$. Thus, we have sufficient information to calculate u_2^{n+1} and u_{M-1}^{n+1}. We will discuss how to deal with these boundary values in detail shortly.

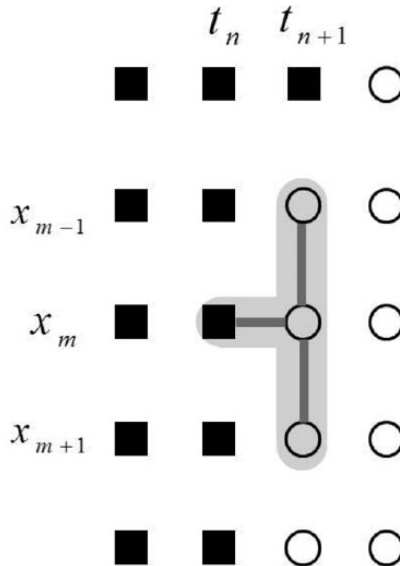

FIGURE 9.3: Implicit method stencil.

The implicit method is always numerically stable and convergent but is typically more computationally intensive than the explicit method as it requires solving a system of numerical equations at each time step. As we have used the backward time difference and the central space difference the (local) error in the approximation is the same as the explicit method. Note that the global error may be different due to the different methods of computation required to get to the advanced time step.

9.4.1.3 Crank–Nicolson Method

Finally, if we use a central difference approximation at time $t_{n+1/2}$ and a second-order central difference for the space derivative at position x_m we get the recurrence equation:

$$\frac{u_m^{n+1} - u_m^n}{k} = \frac{1}{2}\left(\frac{u_{m+1}^{n+1} - 2u_m^{n+1} + u_{m-1}^{n+1}}{h^2} + \frac{u_{m-1}^{n} - 2u_m^{n} + u_{m-1}^{n}}{h^2} \right). \tag{9.55}$$

Remember to get the central difference formula we sum the forward difference with the backward difference; the factor of one-half considers that we are assessing the time halfway between grid points.

Equation (9.55) is known as the Crank–Nicolson method. The method was developed by two Britons namely John Crank, a mathematical physicist, and Phyllis Nicolson a mathematician, in the mid-20th century. Note that this method is implicit as the approximation to the advanced time step is dependent on itself and we can obtain u_m^{n+1} from solving the system of linear equations given by

$$2(1+r)u_m^{n+1} - r\left(u_{m-1}^{n+1} + u_{m+1}^{n+1}\right) = 2(1-r)u_m^n + r\left(u_{m-1}^n + u_{m+1}^n\right). \tag{9.56}$$

The stencil for the Crank–Nicolson method is shown in Figure 9.4. Here we use three known values and three unknown values to generate an equation.

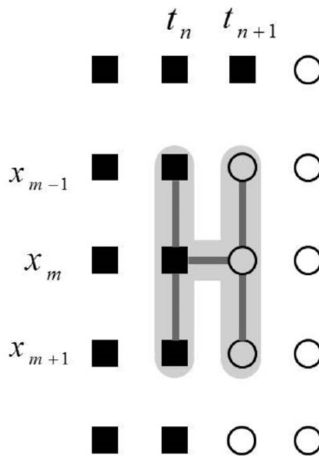

FIGURE 9.4: The Crank-Nicolson stencil.

Note that this stencil is still only moved down one row at a time. In its current position in Figure 9.4, we have enough information to compute grid point u_{m-1}^{n+1}.

The Crank–Nicolson method is always numerically stable and convergent. However, be aware that if $r > 1/2$ this method may produce unwanted decaying oscillations in the solution. The errors for the Crank–Nicolson method are quadratic over both the time step k, and the space step h.

9.4.1.4 General Finite Difference Method

We can condense these three methods into one elegant formula with a so-called weighted variable θ. Using this variable, we can rewrite the finite difference approximations above as

$$\frac{u_m^{n+1} - u_m^n}{k} = \theta \frac{u_{m+1}^{n+1} - 2u_m^{n+1} + u_{m-1}^{n+1}}{h^2} + (1-\theta)\frac{u_{m+1}^n - 2u_m^n + u_{m-1}^n}{h^2}, \quad (9.57)$$

where $\theta = 0$ gives the explicit method, $\theta = 1/2$ gives the Crank–Nicolson method, and $\theta = 1$ is the implicit method, though θ could have any value in the range $[0,1]$.

Rearranging Equation (9.57) into an expression for the advanced time step we obtain

$$\begin{aligned} &-(r\theta)u_{m+1}^{n+1} + (1+2r\theta)u_m^{n+1} - (r\theta)u_{m-1}^{n+1} \\ &= r(1-\theta)u_{m+1}^n + \left(1-2r(1-\theta)\right)u_m^n + r(1-\theta)u_{m-1}^n \end{aligned} . \quad (9.58)$$

Note that Equation (9.58) is for the *interior* nodes of the domain grid, that is, $m = 2,\ldots,M-1$. The values of the grid nodes u_1^n and u_M^n are given by the boundary conditions.

Although not obvious at all Equation (9.58) represents a tridiagonal system of linear equations, that is, it can be written in the form of

$$A\underline{u}^{n+1} = B\hat{\underline{u}}^n + \Omega^{n+1}, \quad (9.59)$$

where A is an $(M-2)$-by-$(M-2)$ tridiagonal matrix given by

$$A = \begin{bmatrix} 1+2r\theta & -r\theta & 0 & \cdots & 0 \\ -r\theta & 1+2r\theta & -r\theta & \ddots & \vdots \\ 0 & -r\theta & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -r\theta \\ 0 & \cdots & 0 & -r\theta & 1+2r\theta \end{bmatrix}; \quad (9.60)$$

$\underline{u}^{n+1}$ is an $M-2$ sized vector of the function values at the interior grid nodes at the advanced time step given by

$$\underline{u}^{n+1} = \begin{bmatrix} u_2 \\ u_3 \\ \vdots \\ u_{M-2} \\ u_{M-1} \end{bmatrix}_{n+1}, \quad (9.61)$$

$\hat{\underline{u}}^{n}$ is an M sized vector of the function values at the current time step, that is, the known values given by

$$\hat{\underline{u}}^{n} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{M-1} \\ u_M \end{bmatrix}_{n}, \quad (9.62)$$

B is an $(M-2)$-by-M tridiagonal-ish (technical term) matrix given by

$$B = \begin{bmatrix} r\varphi & 1-2r\varphi & r\varphi & 0 & \cdots & 0 & 0 \\ 0 & r\varphi & 1-2r\varphi & r\varphi & \ddots & \vdots & \vdots \\ \vdots & 0 & r\varphi & 1-2r\varphi & \ddots & 0 & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & r\varphi & 0 \\ 0 & 0 & \cdots & 0 & r\varphi & 1-2r\varphi & r\varphi \end{bmatrix}, \quad (9.63)$$

where for convenience $\varphi = (1-\theta)$; and Ω is a vector of size $M-2$ to deal with the boundary conditions given by

$$\Omega^{n+1} = r\theta \begin{bmatrix} u_1 \\ 0 \\ \vdots \\ 0 \\ u_M \end{bmatrix} \tag{9.64}$$

That is, there are $M-4$ zeros between the first and last elements, which are the values of the corresponding boundary nodes u_1 and u_M.

To explain Ω, consider Equation (9.58) at grid point $m = 2$:

$$\begin{aligned} &-(r\theta)u_3^{n+1} + (1+2r\theta)u_2^{n+1} - (r\theta)u_1^{n+1} \\ &= r(1-\theta)u_3^n + \big(1-2r(1-\theta)\big)u_2^n + r(1-\theta)u_1^n . \end{aligned} \tag{9.65}$$

Remember that we should write these equations with the unknown values on the left-hand side and the known values on the right-hand side. Since the value u_1^{n+1} is a boundary node it is a known value and should be moved to the right-hand side of the equation. A similar argument can be made for the grid point $m = M-1$ with the boundary value u_M^{n+1} also being moved to the right-hand side. Note that this manipulation is particular to Dirichlet boundary conditions. For other types of boundary conditions, different methods of dealing with these boundary values must be applied. We will discuss how to deal with a Neumann boundary condition in the next section.

If the source of these manipulations is unclear from the equations, then consider Figure 9.5(a). This shows the space-time domain which we have discretized into a grid of points; the space axis is vertical, and the time axis is horizontal. Thus, each column of points represents the solution across the spatial dimension at each time step we are taking. The open circles represent the unknown, interior grid points that we are attempting to calculate, that is, the left-hand terms of our equations, whereas the filled squares represent

our known values, that is, the right-hand terms of our equations. The first and last *rows* of the grid are given by the *boundary conditions*, whereas the first *column* is given by the *initial condition* of the function. Figure 9.5(b) shows the state of the grid at the *n th* time step. Here n columns have changed from unknown values to known values and we are ready to calculate the $(n+1)$ th time step. If we apply our Crank–Nicolson stencil to the grid point u_2^{n+1}, Figure 9.5(c), then we see that it contains a boundary value at the advanced time step. Necessarily, this value must be brought to the right-hand side of our equations; hence the value appears at the start of Ω. Similarly, the stencil applied to grid point u_{M-1}^{n+1} also includes a boundary value, hence the value at the end of Ω.

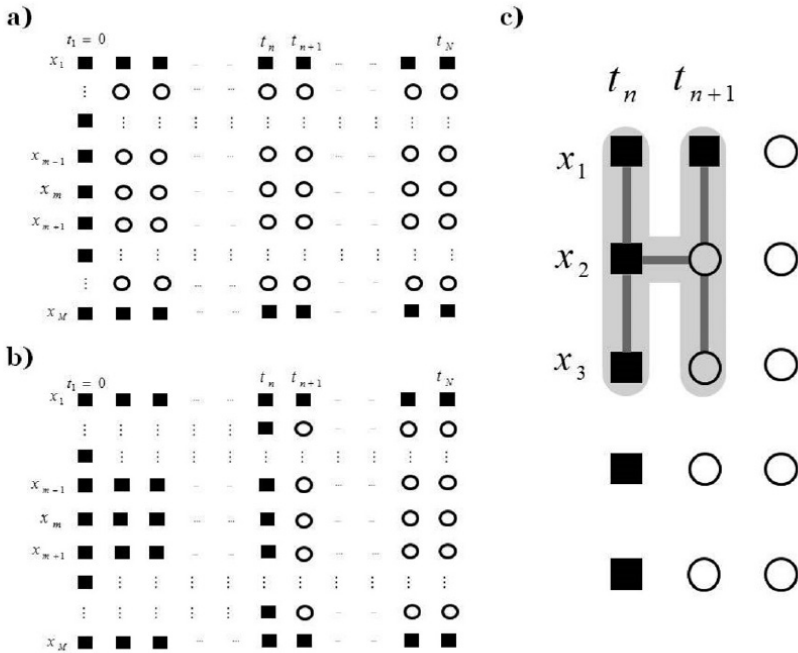

FIGURE 9.5: (a) The initial state of the system; (b) The state of the system at time t_n; (c) the general stencil applied at a boundary.

It was mentioned earlier that we would only deal with Dirichlet conditions that are constant values, and not functions of time (or other spatial coordinates). The reason for this becomes apparent when we consider the addition of Ω in Equation (9.59). As $u_1^n = u_1^{n+1}$

and $u_M^n = u_M^{n+1}$ for all n, we can incorporate Ω into matrix B such that the first and last entries of B both simplify to r.

To solve Equation (9.46) formally would require us to compute

$$\underline{u}_{n+1} = A^{-1}\left(B\underline{u}_n + \Omega_{n+1}\right). \tag{9.66}$$

As calculating the inverse of a matrix becomes infeasible for large values of M we must rely on other methods of solution. Fortunately, LAPACK offers several subroutines for doing such a task. Specifically, we have a symmetrical, tridiagonal matrix and thus require the subroutine "DPTSV" to solve our system. Be aware that this subroutine performs an entire solution of the system by first factorizing the matrix A, and using those factors to find the solution. In doing so it overwrites A with the factorized elements.

The program file *heatEqn1.f90* contains the code to perform the numerical solution of the one-dimensional, time-dependent heat equation for a specific set of boundary and initial conditions. As we have used the weighting parameter θ we can use this to set the finite difference method the code uses. Figure 9.6 shows some selected results of running the code with the ends of a 1 m length metallic rod both held at 100°C, with an initial temperature of 0°C, and with $\theta = 0.5$, which is the Crank–Nicolson method. Here we set h = 0.2 and k = 0.01 making $r = 1/4$. We can see that the metallic rod behaves (at least qualitatively) as we might expect; heat flows into the rod from the hot ends in a smooth and symmetrical fashion, until the entire rod reaches 100°C after-which nothing happens that is, it has reached a steady-state. Our numerical results suggest this happens in around a second or less but, of course, we initially set the diffusion constant (related to the thermal conductivity of the metal) to one. It should not be too much of a stretch to reintroduce it back into our program code. Figure 9.6 is somewhat of a cheat as it uses Excels' built-in smoothing function to guess at the values in-between the grid points, but you surely know how it does this by now, right? (If not, re-examine the chapter on interpolation).

The heat equation does have an analytical solution with Dirichlet boundary conditions that can be obtained through a technique called separations of variables. As we are treating the more general case of non-zero, Dirichlet boundary conditions can be a little tricky.

To recap we are solving the following PDE for the distribution of heat through a metallic rod of length L

$$U_t = kU_{xx}, \tag{9.67}$$

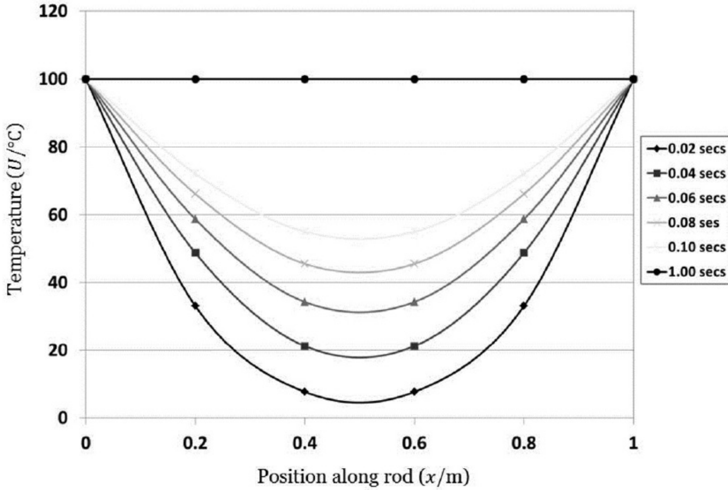

FIGURE 9.6: Numerical approximation to the transient flow of heat in a one dimensional metal rod with Dirichlet boundary conditions.

with boundary conditions

$$U(0,t) = c_1; \tag{9.68}$$

$$U(L,t) = c_2 \tag{9.69}$$

and with initial condition defined as

$$U(x,0) \equiv U_0(x). \tag{9.70}$$

The particular solution to this differential equation is given by

$$U(x,t) = U_E(x) + \sum_{n=1}^{\infty} D_n \sin\left(\frac{n\pi x}{L}\right) e^{-k\left(\frac{n\pi}{L}\right)^2 t}, \tag{9.71}$$

where the coefficients are

$$D_n = \frac{2}{L}\int_0^L \left(U_0(x) - U_E(x)\right) sin\left(\frac{n\pi x}{L}\right) dx \tag{9.72}$$

and we have the steady-state or equilibrium solution

$$U_E(x) = c_1 + \frac{c_2 - c_1}{L}x. \tag{9.73}$$

This solution is recognized as being a Fourier series and is the solution (give or take some notation) Fourier found to the heat equation in the original work. We can see that this solution makes physical sense because if $t \to \infty$ then the summation term in Equation (9.71) goes to zero and $U(x,t) \to U_E(x)$. Also, if the initial conditions $U_0(x) = U_E(x)$, then the coefficients are zero for all n and we have the situation where there is no time dependence. This is of course what our physical intuition expects; if the temperature starts out at the steady-state solution, then it will remain at that solution for all time; that is why it is called a steady-state solution.

We can use this analytical solution, Equation (9.71), to check the accuracy of our numerical approximation. This is presented as an exercise for you to modify the *heatEqn1.f90* to include this analytical solution to provide a measure of the error in the approximation. If we had no analytical solution, how might we estimate the error in our approximation, or at least ensure a nominal significant figure accuracy?

9.4.2 The Heat Equation with Neumann Boundaries

Consider again the metallic rod but with one end insulated, rather than held at a constant temperature. In this case, we are solving the same system

$$U_t = kU_{xx}, \tag{9.74}$$

but with the mixed boundary conditions of

$$U(0,t) = c\,; \tag{9.75}$$

$$U_x\,|_{x=L} = 0 \tag{9.76}$$

Again, the initial condition can be defined as

$$U(x,0) = U_0(x). \tag{9.77}$$

Equation (9.76) is telling us that there is no transfer of the physical quantity (temperature in this case) with respect to space

across the specific boundary $x = L$. In other words, the temperature gradient is zero across the boundary.

As the Neumann boundary condition, Equation (9.76), is a differential equation we need to replace it with a finite difference approximation. Recall that to maintain an error behavior of the entire system our treatment at the boundaries of the domain should match that which we use on the interior of the domain. Fortunately, each of the methods discussed (implicit, explicit, and Crank–Nicolson) have an $\mathcal{O}(h^2)$ dependence on the space variable. Hence, any difference formula we use with an $\mathcal{O}(h^2)$ error behavior to deal with the Neumann boundary condition will also apply to our general, weighted formula (Equation (9.57)).

One of the simplest ways of approximating Equation (9.76) with a finite difference equation is to use an $\mathcal{O}(h^2)$ forward or backward (depending on the location of the boundary) difference formula. The reason why we do not use a central difference formula is that it introduces a grid point that lies outside our computational domain, and while there are techniques to deal with this complication, they are beyond the scope of our discussion here.

For our Neumann boundary, we use the approximation

$$U_x \,|_{x=L} \approx \frac{1}{2h}\left(u_{M-2}^n - 4u_{M-1}^n + 3u_M^n\right) = 0, \tag{9.78}$$

for all n. This has an $\mathcal{O}(h^2)$ error behavior. Rewriting Equation (9.78), we obtain the following expression for the boundary value at $x = L$

$$u_M^n = \frac{4}{3}u_{M-1}^n - \frac{1}{3}u_{M-2}^n. \tag{9.79}$$

We now have a means of calculating the value of U at the insulated boundary from the neighboring interior grid points for a particular time step. But how does this affect our tridiagonal system of equations?

To answer that question, we substitute Equation (9.79) into Equation (9.58) taken at the boundary $x = L$ such that

$$-(r\theta)\left(\frac{4}{3}u_{M-1}^{n+1}-\frac{1}{3}u_{M-2}^{n+1}\right)+(1+2r\theta)u_{M-1}^{n+1}-(r\theta)u_{M-2}^{n+1}=\ldots \quad (9.80)$$

where … represents the right-hand side of Equation (9.58). We do not apply the substitution on the right-hand side as we either know the value of U at the boundary due to the initial conditions, or we calculate it from Equation (9.79). After some rearrangement, we find the following

$$\left(1+\frac{2}{3}r\theta\right)u_{M-1}^{n+1}-\left(\frac{2}{3}r\theta\right)u_{M-2}^{n+1}=\ldots. \quad (9.81)$$

Notice that we have not taken any terms over to the right-hand side of these equations to deal with the Neumann boundary. Thus, we make the following changes to our system of matrices and vectors. Matrix A remains unchanged except from its final two elements that now have the values $a_{M-2,M-3}=-2r\theta/3$, and $a_{M-2,M-2}=1+2r\theta/3$. The additional boundary vector, Ω, is modified such that its final entry is now zero. Incorporating this change into the matrix B we see that its final element returns to the expression $b_{M-2,M}=r(1-\theta)$; remember that in the previous section we used the constant Dirichlet boundary conditions to simplify the first and last entries of B both to r. The vectors $\underline{u}$ and $\underline{\tilde{u}}$ need no modification.

The first thing to notice is that A is *no longer symmetrical*. This means we require a different LAPACK subroutine to solve this system of equations. The subroutine "DGTSV" solves a *general* tridiagonal system of linear equations using Gaussian elimination. As with the previous LAPACK subroutine, the arrays representing A that is passed to "DGTSV" are overwritten by their factors, hence they require resetting before looping to the next time step. The second thing to notice is that after we have solved the system at a particular time step, we need to calculate the temperature at the Neumann boundary using Equation (9.79).

Figure 9.7 shows some of the results of running the program after making the necessary changes to the code found in *heatEqn1.f90*. Here we maintain the values of h and k from the previous section and the constant Dirichlet boundary is set to 100°C.

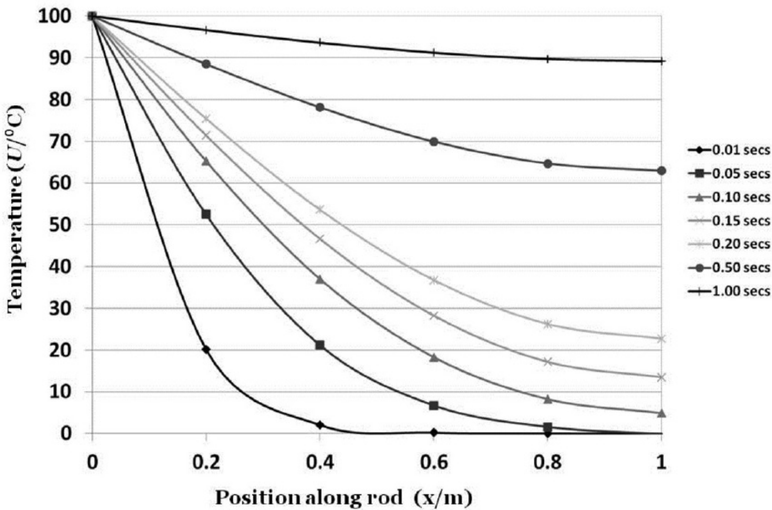

FIGURE 9.7: Numerical approximation to the transient flow of heat in a one-dimensional metal rod with mixed boundary conditions.

9.4.3 The Steady-State Heat Equation

We have already established that the solution to the heat equation consists of a transient phase that evolves in time, which eventually settles on to a steady-state solution. This steady-state solution is also referred to as the equilibrium of the system. We can write a differential equation that expresses this steady-state as

$$\nabla^2 U(x,y,z) \equiv \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + \frac{\partial^2 U}{\partial z^2} = U_{xx} + U_{yy} + U_{zz} = 0, \quad (9.82)$$

where we have explicitly used our notation for partial differentiation. Here U is some physical field (temperature in this case) defined over three spatial dimensions; notice there is no time dependence. Equation (9.82) is called the *Laplace equation* and crops up often in physics wherever a system reaches an equilibrium state.

Consider a two-dimensional metallic plate that is held at known but different temperatures along its boundaries (Dirichlet boundary conditions). We want to find the *steady-state temperature* of the plate. This is like the one-dimensional metallic rod problem we discussed earlier in that we know the temperature at the boundaries of the system. However, in this steady-state case, we are not interested

in the transient behavior of the system. As such we are not required to know the initial state of the system to find a unique solution. In fact, we wish to solve this problem using an iterative or relaxation method.

To simplify the mathematics let us consider the metallic plate to be a rectangle of dimensions $a \times b$, with its lower-left corner situated at the origin of our coordinate system. Thus, we can write

$$U_{xx} + U_{yy} = 0, \tag{9.83}$$

with the following boundary conditions proceeding clockwise from the origin:

$$U(0,y) = c_1;$$
$$U(x,b) = c_2;$$
$$U(a,y) = c_3;$$

and

$$U(x,0) = c_4, \tag{9.84}$$

for x on the interval $[0,a]$ and y on the interval $[0,b]$. For arguments sake the c_i are taken to be constants but in the general case, they would be functions of x and y.

To solve this problem numerically we derive the finite difference approximation to the second-ordered PDE of Equation (9.83) by defining a two-dimensional grid on our rectangular plate. Hence, we write

$$x_i = ih, \; i = 0,1,\ldots,N,$$

$$y_j = jk, \; j = 0,1,\ldots,M, \tag{9.85}$$

where $h = a/N$ and $k = b/M$. The finite difference approximation can be written as

$$\frac{1}{h^2}\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right) + \frac{1}{k^2}\left(u_{i,j+1} - 2u_{i,j} + u_{i,j-1}\right) = 0. \tag{9.86}$$

Remember that we are going to solve this problem *indirectly* using a *relaxation method,* so we need to solve Equation (9.86) for $u_{i,j}$. In doing so we find that

$$u_{i,j} = \frac{h^2k^2}{2h^2 + 2k^2}\left(\frac{u_{i+1,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} + u_{i,j-1}}{k^2}\right). \tag{9.87}$$

We can simplify Equation (9.87) further by imposing that $h = k$ and in this case

$$u_{i,j} = \frac{1}{4}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right). \tag{9.88}$$

Note that this has a rather simple geometric interpretation. It states that the solution at a particular grid node is the arithmetic mean of its (four) nearest neighbors.

It is left as an exercise for the reader to implement code to solve the steady-state heat equation in two dimensions. Use the file *gauss-seidel.f90* as a guide; remember that you will need to express the array u in two dimensions, looping over both i and j, as well as initializing the values on the boundary and the initial guess for the interior grid nodes. As a comment on strategy, start out simple that is, use a small grid and have boundary conditions for which you can guess the solution, then add complexity once you have some working code. How might you visualize the data?

One such complication we might add is the inclusion of a Neumann boundary condition along with one of the edges of our rectangular plate. For arguments let us replace the Dirichlet condition at the left-hand edge with the Neumann condition that

$$U_x\left(x, y_j\right)\big|_{x=0} = \alpha_j \approx \frac{u_{1,j} - u_{-1,j}}{2h}, \tag{9.89}$$

where we have introduced α_j as the partial derivative along the boundary $(0, y)$ for y on the interval $[0, b]$. Remember that the finite difference formula here is the first-ordered central difference approximation and thus has $\mathcal{O}\left(h^2\right)$ accuracy that matches the accuracy of the method used for the interior grid nodes. Using Equation (9.88), we can write the finite difference approximation for values along the left-hand edge $(0 < j < M - 1)$ as

$$u_{0,j} = \frac{1}{4}\left(u_{1,j} + u_{-1,j} + u_{0,j+1} + u_{0,j-1}\right). \tag{9.90}$$

As we can see this equation involves a term that lies outside our physical grid. However, we can use Equation (9.89) to substitute in for $u_{-1,j}$, which gives

$$u_{0,j} = \frac{1}{4}\left(2u_{1,j} - 2ha_{j} + u_{0,j+1} + u_{0,j-1}\right). \qquad (9.91)$$

In general, a_{j} will be a function of x and y but is typically modeled by a constant value. The physics of Equation (9.89) is that heat will flow if there is a temperature gradient across a boundary, and that it will flow from hot to cold. For instance, if a was negative then heat flows into the plate from the environment, if a is positive that situation is reversed. Note that this direction of heat flow depends on the location of the boundary. For instance, if we consider the right-hand edge of the plate then a negative temperature gradient (with respect to the x direction) implies heat flows *out of* the plate into the environment, whereas a positive temperature gradient implies the opposite.

9.4.4 The Wave Equation

The general wave equation in one spatial dimension is given by

$$U_{tt}(x,t) = c^2 U_{xx}(x,t), \qquad (9.92)$$

where c is the speed of the wave, and U represents some physical field, for example, the displacement of the wave in mechanical oscillations. As with the heat equations to solve this equation numerically, we define a discrete grid of points over both space and time, and then derive the finite difference approximation. Thus

$$\frac{1}{k^2}\left(u_m^{n+1} - 2u_m^n + u_m^{n-1}\right) = \frac{c^2}{h^2}\left(u_{m+1}^n - 2u_m^n + u_{m-1}^n\right), \qquad (9.93)$$

where we have imposed the discrete, uniform grid of points

$$x_m = mh,\ m = 0,1,\ldots,M\ ;$$

$$t_n = nk,\ n = 0,1,\ldots,N\ , \qquad (9.94)$$

with $h = (x_M - x_0)/M$ and $k = (t_N - t_0)/N$; we have yet to define the space and time boundaries of the system. For convenience we usually take $x_0 = 0$ and $t_0 = 0$.

As we are solving for a time-dependent problem then we need to find an equation for the advanced time step in terms of values from previous time steps. Rearranging Equation (9.93) for the advanced time step we obtain

$$u_m^{n+1} = \rho\left(u_{m+1}^n + u_{m-1}^n\right) + 2(1-\rho)u_m^n - u_m^{n-1}, \tag{9.95}$$

where we have introduced

$$\rho = \left(\frac{kc}{h}\right)^2. \tag{9.96}$$

Interpreting Equation (9.95) we see that we can compute u for all x_m so long as we know u for all x_m at the two previous time steps. Note that this is an initial value problem in that to compute a solution we need to know the initial value of u for all x. But Equation (9.95) is telling us we need u at two previous time steps to advance the solution; we appear to be missing information. This dilemma is resolved when we realize that we have both the initial function u and can determine the first-ordered time derivative of that initial function using a finite difference approximation. Explicitly

$$U_t\left(x_m, t\right)\big|_{t=0} = \tau_m \approx \frac{u_m^1 - u_m^{-1}}{2k}, \tag{9.97}$$

where we have introduced τ_m for the partial derivative with respect to time for all x_m at the initial time $t = 0$. In order to calculate the first time step from the initial condition we use Equation (9.97) in Equation (9.95) to obtain the following expression

$$u_m^1 = \frac{\rho}{2}\left(u_{m+1}^0 + u_{m-1}^0\right) + (1-\rho)u_m^0 + k\tau_m. \tag{9.98}$$

Note that we used a finite difference approximation of τ to derive the formula for the initial time step. However, τ might be some known function of x, for instance, if the initial function $u(x,0)$ is easily differentiable, then we need not compute the finite difference approximation and merely compute $\tau_m = \tau\left(x_m\right)$ for each spatial grid point.

The astute reader will have already noticed that these conditions are essentially Cauchy boundary conditions; we can think of the initial time as being a boundary on the time dimension. You also may

have noticed that we have not yet imposed boundaries on the spatial coordinate. In fact, we are not required to impose such conditions, however, this leads to rather uninteresting and unphysical systems. Imagine, if you will, an infinitely long, stretched spring. We send a pulse wave down that spring by displacing it in some way; the tension in the spring provides the force to drive the pulse. Assuming no attenuation that wave continues to travel along the spring in the same direction for eternity. For normal, earthly, finite springs we must fix at least one end (if not both) to stretch it and provide the tension required to carry a wave. This imposes the boundary conditions on our spatial coordinate.

It may or may not be of some surprise to you that it is in the modeling of those spatial boundary conditions rather than the wave equation itself that we find the most interesting physics. What are the spatial boundary conditions for that system? What if we were to fix both ends or leave both ends free to oscillate? Would the outcome be as satisfying? For more serious musicians, those who play instruments that require a little more skill than mere twanging, it is more the interaction of the vibrating string with its support structures that is important for producing the instrument's sound than is the vibrating string itself. Have you ever wondered why you don't see any square drums, or why brass and other wind instruments have a flared end?

Please write a program to implement the time-dependent wave equation in one dimension, initially with fixed spatial boundaries. Here we do not need any relaxation method or tridiagonal matrix solver as Equation (9.95) is explicit; we simply use that expression to advance our solution. The biggest issue here is in how one is going to visualize the data. You could save all the data to one large, two-dimensional array, where the columns represent the time steps and the rows represent the spatial grid points, and then write that array to a text file, say, for inspection with a graphics program, such as *gnuplot* or *Excel* for example. Another way would be to have three, one-dimensional arrays, one to hold all the u_m at time t_{n-1}, a second to hold all the u_m at time t_n, and the third to hold all u_m at the advance time step, t_{n+1}. The arrays are shuffled accordingly. You could then save a snapshot of the system every "x" number of

seconds for later inspection with a graphics package. For the more ambitious out there one may want to find a way of animating results instantaneously on screen.

9.5 POINTERS TO THE FINITE ELEMENT METHOD

It is at this point in the discussion of PDE that most textbooks will mention the finite element method. To describe succinctly the method in easy-to-understand, plain English is a difficult thing to do especially at an introductory level to computational physics. Indeed, the previous sections on finite difference methods were difficult enough. Take note that the finite element method is extremely useful and can produce some highly accurate, and even beautiful solutions to some particularly nasty, and complex problems.

First, let us expound on the difference between the finite element method (FEM) and the finite difference method (FDM). In FDM, we take our computational domain and partition it into discrete points. The derivatives at those points are given by difference formulas, which we then use to approximate the governing differential equation. Along with the boundary and/or initial conditions, the resulting system of linear equations is solved. In this way, we obtain an exact solution to an approximate problem. The FEM takes an alternative approach in that it uses a trial function, defined by some parameter, to estimate the solution, and the resulting equations are solved in some best sense. In other words, it finds an approximate solution to the exact problem.

Next comes the precise formalism of the FEM. It involves talking about piecewise linear trial functions, basis functions, weighted residuals, and the Galerkin method.

Computational Differential Equations (1996) by K. Eriksson, D. Estep, P. Hansbo, and C. Johnson contains several chapters on the practical use of the FEM, as is accessible to the undergraduate student with some background knowledge (i.e., after having read this book) on numerical techniques.

An Introduction to the Finite Element Method (3rd ed.) by J. N. Reddy is more geared toward engineering undergraduates but does provide an excellent reference to the topic.

The Finite Element Method: A Practical Course (2003) by S. S. Quek and G. R. Liu provides an in-depth look at FEM and takes the reader through the basics of the method, providing examples and comprehensive discussions of applications and implementations.

The Finite Element Method: Volume 1: The Basis (5th ed.) by O. C. Zienkiewicz and R. C. Taylor provides a comprehensive and up-to-date overview of the topic and it accessible to undergraduate students. Volumes 2 and 3 are more complex but provide excellent grounding for anybody studying higher-level FEM.

EXERCISES

9.1. How might you go about setting up an actual experiment in the lab to test the accuracy of our numerical method for the heat distribution in a metallic rod?

9.2. The temperature of one end of a metallic rod is held at $0°C$ the other is held at $100°C$. Using a finite difference method determine how the temperature of the rod evolves in time if the initial temperature of the rod was $20°C$ throughout. Ensure 4 significant figures of accuracy. (Hint: use the analytical, Fourier series result, and/or Richardson extrapolation.)

9.3. Consider the differential equation

$$y'' + 3y' - 5y = 7x$$

subject to the boundary conditions

$$y(0) = -20,\ y(1) = 100$$

Find a numerical solution to this equation using a direct finite difference method. Ensure at least 4 significant figures of accuracy.

9.4. Solve the same differential equation in the previous question but using a relaxation method. Ensure the same level of accuracy. Comment on any differences between the two methods.

9.5. The temperature of a unit square metal plate is subject to the following conditions

$$U(0,y) = e^{-10(y-0.5)^2},\ 0 < y < 1,\ U(x,0) = U(x,1) = 100x$$

and the right-hand side boundary is insulated. Find the steady-state temperature of the plate.

9.6. A string on a guitar is plucked such that the initial function of the string can be described as

$$u(x,0) = \begin{cases} 0, & x = 0 \\ e^{-80(x-ct-0.5)^2}, & 0 < x < 1 \\ 0, & x = 1 \end{cases}$$

In other words, the string is of unit length and held by rigid supports at its ends. The tension in the string is 12.8N and has a mass of 2g. Here the speed of the wave is $c^2 = T/\mu$ where T is the tension and μ is the mass per unit length. Evaluate what happens to the wave as time progresses. Did you observe phase reversal at the rigid supports?

9.7. Repeat the previous exercise but with one of the supports free. That is either

$U_x\,|_{x=0} = 0$ or $U_x\,|_{x=1} = 0$.

(Hint: Use a finite difference approximation to find an expression for u_0^n or u_M^n.)

9.8. A more realistic model is to assume the supports have inertial mass, M. If the (vertical) force on the supports is given by

$$F = TU_x\,|_{x=0,1}$$

find an expression for the boundary conditions and modify your program appropriately to study their behavior in terms of their inertial mass.

9.9. Assuming the supports behave like damped, simple harmonic oscillators try to establish a more realistic model of the guitar string.

CHAPTER 10

ADVANCED NUMERICAL QUADRATURE

The advanced nature that we boldly state in the chapter title is more to with the derivation of the methods rather than the application of the methods themselves. As with many computational algorithms we do not need an in-depth understanding of why they work, just the knowledge that they do work and how to apply them. However, if we are going to use them, we should make a little concerted effort to try to understand how they work to use them effectively.

This chapter covers the derivation and use of the Gauss–Legendre and Gauss–Laguerre quadrature. These two schemes will generate the most accurate numerical solutions for the least amount of computational effort and should be used wherever possible in the numerical solution of a physical problem that involves integrals.

10.1 GENERAL QUADRATURE

In general, any integration can be approximated by a numerical quadrature written in the form of

$$\int_a^b f(x)\,dx \approx \sum_{m=1}^{N} w_m f(x_m), \tag{10.1}$$

where x_m are the evaluation points, w_m are weights given to the mth point, and there are N evaluation points in total. For convenience, the points are set with uniform spacing, typically denoted by h, and we can derive the numerical quadrature methods as discussed in Chapter 5 (trapezoidal rule, Simpson's rule, etc.). To do this, we begin by deriving the simplest formula from Equation (10.1) which is to only consider the limits of the integration a and b thus

$$I \approx w_1 f(x_1) + w_2 f(x_2) \tag{10.2}$$

where $x_1 = a$ and $x_2 = b$. In the limit of the integration interval $(b-a)$ going to zero, we require that Equation (10.2) be exact for any function. This sounds like a difficult task, however, if we consider any function that has a Taylor series expansion, we note that the first two terms of that expansion are multiples of 1 and x (note that sometimes the multiple is zero c.f. sine). Hence, we now attempt to find the weights, w_1 and w_2, that will give

$$\int_{x_1}^{x_2} 1dx = x_2 - x_1 = w_1 + w_2 \tag{10.3}$$

and

$$\int_{x_1}^{x_2} xdx = \frac{x_2^2 - x_1^2}{2} = w_1 x_1 + w_2 x_2. \tag{10.4}$$

Note the equivalencies; these equations are *exact*. We have two equations and two unknowns, namely the weights. Solving for the weights we find

$$w_1 = w_2 = \frac{x_2 - x_1}{2}. \tag{10.5}$$

Typically, we would write $h = x_2 - x_1$ and our approximation for the integral of any function becomes

$$\int_a^b f(x)dx \approx \frac{h}{2}(f_1 + f_2), \tag{10.6}$$

which is the (primitive) trapezoidal rule. Note that it is from this derivation that we can explicitly state the error behavior of a numerical quadrature using what is called the Lagrange remainder (this

is derived using the mean value theorem applied to the remainder term in the Taylor series expansion)

$$\int_a^b f(x)dx = \frac{h}{2}(f_1 + f_2) - \frac{h^3}{12} f(c), \tag{10.7}$$

where c lies somewhere in the integration interval.

Of course, we can add more evaluation points. If we now take three points $x_1 = a$, $x_2 = (b+a)/2$, and $x_3 = b$ such that $h = (b-a)/2$ we can write the following equivalencies,

$$x_3 - x_1 = w_1 + w_2 + w_3\,, \tag{10.8}$$

$$\frac{x_3^2 - x_1^2}{2} = w_1 x_1 + w_2 x_2 + w_3 x_3 \tag{10.9}$$

and

$$\frac{x_3^3 - x_1^3}{3} = w_1 x_1^2 + w_2 x_2^2 + w_3 x_3^2, \tag{10.10}$$

where we have used the first three terms of the Taylor series expansion. Here we have three equations and three unknowns, and solving for the weights gives us Simpson's rule

$$\int_{x_1}^{x_3} f(x)dx = \frac{h}{3}(f_1 + 4f_2 + f_3) - \frac{h^5}{90} f^{[4]}(c). \tag{10.11}$$

Adding another evaluation point leads to Simpson's three-eighths rule, five points lead to Boole's rule, and so on. As a reminder all these formulas require that the $f(x)$ be expressible as a polynomial, that is, they have a Taylor series expansion. Indeed, usually, these integration rules are derived by considering a polynomial approximation to the function and integrating that approximation exactly. For instance, the trapezoidal rule is a linear approximation, Simpson's rule is a quadratic approximation, and so forth.

In the treatment above we constrained the evaluation points to be equally spaced. However, this is not a requirement, and we can take the evaluation points to be anywhere within the integration region. In fact, by removing this constraint we can derive more accurate numerical quadrature methods, but they are not so easy to derive as we increase the number of evaluation points.

To demonstrate, if we consider the simplest integration formula in which a single evaluation point can be located anywhere within the integration interval then we now have two unknowns, namely the weight *and* the evaluation point. To solve for these two unknowns, we need two equations, and as before we obtain these equations by requiring that the quadrature be exact for the first two lowest ordered polynomials $f(x)=1$, and $f(x)=x$. This gives the necessary equations thus

$$(b-a)=w_1 \tag{10.12}$$

and

$$\frac{\left(b^2-a^2\right)}{2}=w_1x_1. \tag{10.13}$$

Solving for both the evaluation point and the weight we find that $x_1=(b+a)/2$ and $w_1=b-a$. This is the mid-ordinate rule and is exact for linear functions. Note that the trapezoidal rule is also exact for linear functions, but we must take an extra function evaluation. In fact, the mid-ordinate rule can be considered as the first-ordered Gauss–Legendre quadrature, though normally taken on the normalized integration interval $[-1,1]$; more on this shortly.

If we now add a second evaluation point, we now must find four unknowns which requires four equations. Again, we require that the quadrature be exact for the first four lowest polynomials such that

$$(b-a)=w_1+w_2, \tag{10.14}$$

$$\frac{1}{2}\left(b^2-a^2\right)=w_1x_1+w_2x_2, \tag{10.15}$$

$$\frac{1}{3}\left(b^3-a^3\right)=w_1x_1^2+w_2x_2^2 \tag{10.16}$$

and

$$\frac{1}{4}\left(b^4-a^4\right)=w_1x_1^3+w_2x_2^3. \tag{10.17}$$

To solve these equations for the weights and abscissas we must normalize the integration region such that the interval $[a,b]$ is mapped to the interval $[-1,1]$. After the equations have been rewritten in terms of this mapping it is a (relatively) straightforward task

of finding the weights and abscissas. After performing the necessary steps, we find that the weights are equivalent and equal to one and, the evaluation points, or abscissas to use their technical name, are $x_1 = \sqrt{(1/3)}$ and $x_2 = -\sqrt{(1/3)}$. Note that the quadrature

$$\int_{-1}^{1} f(x)dx \approx f\left(\frac{1}{\sqrt{3}}\right) + f\left(-\frac{1}{\sqrt{3}}\right) \tag{10.18}$$

is now exact if $f(x)$ is a polynomial of order three or less. Remember the trapezium rule is only exact for linear functions (order one or zero). This is the second-order Gauss–Legendre quadrature.

Increasing the number of abscissas to three we find that we have six unknowns and thus require six equations. If one has spotted the pattern then we require the quadrature to be exact for the first six lowest ordered polynomials, that is, up to $f(x) = x^5$. Extending this to N points, we have $2N$ unknowns and therefore require $2N$ equations. This means an N point Gauss–Legendre quadrature is exact for polynomials of order $2N - 1$ and less.

The job then is to find the weights and corresponding abscissas for each of those N points. Rather than trying to continue to solve sets of simultaneous equations, which would get somewhat difficult (and tedious), let us turn to another method, orthogonal polynomials. It is from these that Legendre gets his name appended to the method.

10.2 ORTHOGONAL POLYNOMIALS

We may have heard the term orthogonal banded about the place when discussing vectors or considering Cartesian coordinates. For instance, to determine whether two vectors are orthogonal we take what is known as their scalar product. This product is also known as the dot product or the inner product. If the resulting outcome of the inner product is zero, we know that the two vectors are orthogonal. Essentially, orthogonal is another word for perpendicular or at right-angles to but has a deeper meaning when applied to functions; it means they are fundamentally different.

Orthogonal polynomials are a set of polynomials φ_m defined over a finite range $[a,b]$ such that they obey an orthogonality relation given by

$$\int_a^b w(x)\phi_m(x)\phi_n(x)\,dx = \delta_{mn}c_n \qquad (10.19)$$

where $w(x)$ is a weighting *function*, δ_{mn} is called the Kronecker delta that is equal to 1 when $m = n$ and zero otherwise, and c_n is some constant coefficient.

It would be a hopeless task to try to identify a set of orthogonal polynomials via substitution into Equation (10.19), however, that is not the purpose of the relation. We would do the opposite and use the relation to *construct* a set of orthogonal polynomials. To illustrate this process, let us make life easier for ourselves and reduce the complexity of the relationship somewhat. Let us assume that we are on the normalized integration interval such that $a = -1$ and $b = 1$, and let us also assume our weighting function is constant and equal to one. By choosing the integration limits as such we are not losing any generality as the interval can always be mapped back onto any finite region through a change of variables. With these simplifications in place, we can construct the first polynomial, φ_0, by using

$$\int_{-1}^{1} \varphi_0(x)\varphi_0(x)\,dx = c_0. \qquad (10.20)$$

It is often convenient to normalize the polynomials such that all $c_n = 1$, in which case the polynomials are referred to as orthonormal (a contraction of orthogonal and normalized). There are many polynomials that satisfy Equation (10.20) so let us choose the simplest (non-trivial) case that is $\varphi_0 = k_0$, where k_0 is constant. Performing the integration, we find that $2k_0^2 = 1$, hence

$$\varphi_0 = 1/\sqrt{2}. \qquad (10.21)$$

We find the next polynomial φ_1 in the set by requiring that

$$\int_{-1}^{1} \varphi_0(x)\varphi_1(x)\,dx = 0. \qquad (10.22)$$

It is tempting here to just set φ_1 equal to x, which would satisfy Equation (10.21). However, we should be more general in our

approach. Our strategy is to assume that the Nth-ordered polynomial of the orthogonal set is given by the linear combination

$$\varphi_N(x) = k_N\left[u_N(x) + \alpha_{NN-1}\varphi_{N-1}(x) + \ldots + \alpha_{N0}\varphi_0(x)\right], \quad (10.23)$$

where $u_m = x^m$, and we can use the k_m to normalize φ_m, and the a_{mn} are chosen to force orthogonality. We have already found the first polynomial of the set, $\varphi_0 = 1/\sqrt{2}$, hence we write

$$\varphi_1(x) = k_1\left(x + \frac{\alpha_{10}}{\sqrt{2}}\right) \quad (10.24)$$

and we can force the integral of Equation (10.21) to be zero by choosing an appropriate α_{10}. Substituting Equation (10.24) into (10.22), we find that α_{10} is zero. The normalization constant, k_1, is found by performing

$$\int_{-1}^{1}\varphi_1(x)\varphi_1(x)dx = k_1^2\frac{2}{3} = 1, \quad (10.25)$$

such that the next orthonormal polynomial in the set is

$$\varphi_1(x) = \sqrt{\frac{3}{2}}x. \quad (10.26)$$

The next polynomial in the set is then found by the linear combination

$$\varphi_2(x) = k_2\left(x^2 + \alpha_{21}\varphi_1(x) + \alpha_{20}\varphi_0(x)\right). \quad (10.27)$$

Now, we require that φ_2 is orthogonal to both φ_1 and φ_0 such that

$$\int_{-1}^{1}\varphi_0(x)\varphi_2(x)dx = 0 \quad (10.28)$$

and

$$\int_{-1}^{1}\varphi_1(x)\varphi_2(x)dx = 0. \quad (10.29)$$

After performing the necessary calculations, we find that $\alpha_{21} = 0$ and $\alpha_{20} = -\sqrt{2}/3$. Again, we normalize the polynomial by finding k_2 using

$$\int_{-1}^{1} \varphi_2(x)\varphi_2(x)dx = 1 \cdot \tag{10.30}$$

After some manipulation we find that

$$\varphi_2 = \sqrt{\frac{5}{2}}\frac{3x^2 - 1}{2}. \tag{10.31}$$

We can continue in this fashion to find any order of polynomial that fits in this orthonormal set. It is of consequence that Equations (10.21), (10.26), and (10.31) are the first three (normalized) Legendre polynomials.

The Legendre polynomials are the orthogonal set specific to the weighting function equal to one and for the integration range $[-1,1]$. For other weighting functions and integration limits, we would necessarily construct a different set of orthogonal polynomials. For instance, with $w(x) = e^{-x}$ on the integration range $[0,\infty]$ we would arrive at the Laguerre polynomials.

The general process of finding a set of orthogonal (orthonormal) polynomials in this way is due to Jorgen Pedersen Gram, a Danish Mathematician, and Erhard Schmidt, a German Mathematician who jointly developed the eponymous Gram–Schmidt process in the late 19th century.

10.3 GAUSS–LEGENDRE QUADRATURE

To apply our newfound knowledge of orthogonal polynomials to the Gauss quadrature we reformulate our integration such that

$$\int_{a}^{b} f(x)w(x) = \sum_{m=1}^{N} w_m f(x_m) \tag{10.32}$$

where the weighting function $w(x)$ is what is known as positive definite, that, it is never negative, and is the same function as used with the orthogonal polynomials. We have $2N$ unknowns in this equation due to the weights and abscissas. Thus, we require that integration of the polynomials of order up to and including $2N-1$ are to

be given exactly by the quadrature, and we use this requirement to define the weights and abscissas for the quadrature.

To do this, we let $f(x)$ be some arbitrary polynomial of order $2N-1$, and we define φ_N as an orthogonal polynomial of order N that is particular to the weighting function $w(x)$ and the region of the integration $[a,b]$ expressed in Equation (10.32). If we now divide $f(x)$ by φ_N we obtain a quotient term, q, and a remainder term, r, both of which will be polynomials of order $N-1$. Remember this is polynomial long division, something you should have covered in an A-level (or equivalent) mathematics course.

The integral of Equation (10.32) can now be expressed as

$$\int_a^b f(x)w(x) = \int_a^b q_{N-1}(x)\varphi_N(x)w(x)dx + \int_a^b r_{N-1}(x)w(x)dx. \quad (10.33)$$

For the remainder of this argument, we can ignore the remainder term as it is no longer required in our derivation of the weights and abscissas for the quadrature. The quotient polynomial can be expanded as a linear combination of a set of polynomials ranging in order from zero to $N-1$. Fortunately, we already have a (complete) set of polynomials that we can use for this task; $\{\varphi_m\}$, the orthogonal set of polynomials. The curly braces indicate a set and the φ_m are members of the set. Explicitly, we write

$$q_{N-1}(x) = \sum_{m=0}^{N-1} d_m \varphi_m(x), \quad (10.34)$$

where the d_m are constants. With this expansion, we can now express the integral of the quotient term as

$$\begin{aligned}\int_a^b q_{N-1}(x)\phi_N(x)w(x)dx &= \sum_{m=0}^{N-1} d_m \int_a^b \phi_m(x)\phi_N(x)w(x)dx \\ &= \sum_{m=0}^{N-1} d_m \delta_{mN} c_N = 0. \quad (10.35)\end{aligned}$$

The zero emerges as the summation only goes up to $N-1$ and for the Kronecker delta to be non-zero, that is, one, m must equal N.

At the start of our argument, we required that polynomials of order $2N-1$ are given exactly by the quadrature. The product $q_{N-1}(x)\varphi_N(x)$ is a polynomial of order $2N-1$ therefore we can write

$$\int_a^b q_{N-1}(x)\varphi_N(x)w(x)dx = \sum_{m=1}^{N} w_m q_{N-1}(x_m)\varphi_N(x_m) = 0. \quad (10.36)$$

As we have kept the argument general q_{N-1} is an arbitrary polynomial and as such is not necessarily zero at the abscissas. The only way to ensure the sum is zero is to require that all the $\varphi_N(x_m)$ are zero, ignoring the trivial case where all the weights are zero. In other words, we find the roots of the polynomial $\varphi_N(x)$ and select them as our abscissas. As an N order polynomial will have N roots (ignoring the case where the roots are complex) we have found all the abscissas for our N point Gauss quadrature. Now for the weights.

As the quadrature is exact for polynomials of order $2N-1$ it must also be exact for polynomials of lesser order. Here we have the freedom to choose any polynomial that has an order less than $2N-1$ however, we should choose one that simplifies the mathematics. Fortunately, others that have come before us have identified the polynomial we need. Lagrange's interpolating polynomial, or more specifically, the multiplication factor (Equation (3.14)) has the properties we desire. Rewriting it here in terms of our current parameters

$$\lambda_{i,N}(x) = \frac{\prod_{l=1\neq i}^{N}(x-x_l)}{\prod_{l=1\neq i}^{N}(x_i-x_l)} \quad (10.37)$$

where the x_i and x_l are the abscissas. This polynomial is of order $N-1$ and has the property that

$$\lambda_{i,N}(x_m) = \begin{cases} 0, m \neq i \\ 1, m = i. \end{cases} \quad (10.38)$$

We are therefore able to write that

$$\int_a^b \lambda_{i,N}(x)w(x)dx = \sum_{m=1}^{N} w_m \lambda_{i,N}(x_m) = w_i. \quad (10.39)$$

In other words, we can find the weights of the corresponding abscissas by performing the analytical integration on the left-hand side of Equation (10.39).

Currently, this treatment has been general in that we have not defined our integration limits or the weighting function. Let us do this now. We know from our discussion on orthogonal polynomials that by setting our integration region to $[-1,1]$ and choosing $w(x)=1$, we obtain the Legendre polynomials as our orthogonal set. Hence the abscissas for the Gauss–Legendre quadrature are the roots of the Legendre polynomials. Once we have found those roots, we use Equation (10.39) with the appropriate parameters to compute the weights.

To illustrate, consider the Gauss–Legendre quadrature with two points. We use the (normalized) Legendre polynomial

$$\varphi_2 = \sqrt{\frac{5}{2}}\frac{3x^2-1}{2}, \tag{10.40}$$

which has roots

$$x_m = \pm\frac{1}{\sqrt{3}}. \tag{10.41}$$

Notice that we would obtain the same roots using the non-normalized polynomial.

Performing the integration of Equation (10.39) with the appropriate parameters explicitly gives

$$w_1 = \int_{-1}^{1}\lambda_{1,2}dx = \int_{-1}^{1}\frac{x-x_2}{x_1-x_2}dx = \frac{1}{x_1-x_2}\left[\frac{x^2}{2}-x_2x\right]_{-1}^{1} = \frac{-2x_2}{x_1-x_2} = 1$$

and

$$w_2 = \int_{-1}^{1}\lambda_{2,2}dx = \int_{-1}^{1}\frac{x-x_1}{x_2-x_1}dx = \frac{1}{x_2-x_1}\left[\frac{x^2}{2}-x_1x\right]_{-1}^{1} = \frac{-2x_1}{x_2-x_1} = 1.$$

This is the same result we got before by solving the set of (non-linear) simultaneous equations, Equations (10.14)–(10.17). Although getting here was tough and weights for $N=2$ using orthogonal polynomials was easier than solving a set of non-linear simultaneous equations. Even if you do not we now have a general method to obtain the weights and abscissas for any number points N, which was worth it.

In fact, you don't have to do this work as the quadrature is so frequently used that the weights and corresponding abscissas have been published and tabulated, several times over, and to varying degrees of precision. Just type "Gauss–Legendre weights and abscissas" into your favorite search engine and you'll find them.

10.4 PROGRAMMING GAUSS–LEGENDRE

There are two ways forward to programming the Gauss–Legendre quadrature. We can either enter all the weights and abscissas into a "look-up table" expression in a header file and include that header whenever we write a program that involves the quadrature. Or we can write a function that evaluates the weights and abscissas for us every time we wish to perform the quadrature. Both methods have their advantages. Storing the values is a good idea for speed as the values need only be read from memory to be used, though they would have to be entered with care; a typo in a list of numbers can be very tedious to track down. In addition to this, you are limited to the number of weights and abscissas you can be bothered to enter as well as their precision. Whereas the function method gives you the flexibility to decide to use, say, a 20-point Gauss–Legendre quadrature if you so wished, and to whatever precision can be achieved. However, this means these values must be computed every time you perform an integration which may add valuable time on to your overall computation.

The code library that accompanies this book has a header file called *GuassKnotsWeights.h* that contains look-up tables for the knots and weights for various n point Gauss–Legendre and Gauss–Laguerre quadrature. However, we also provide a means to calculate these values so that an N of any value can be used. The source file *Quadrature.cpp* contains implementations of the classes `Legendre` and `Laguerre` both of which can compute the weights and knots (synonym for abscissas) in their constructors if they are not provided for in the look-up tables. In this way, if you are concerned with performance, after some manipulation you could store the weights and

abscissas computed for a particular value of n to a plain text file, say, and then read those values from the file when required.

Legendre polynomials are defined by the following recursive rule

$$\varphi_0 = 1, \tag{10.42}$$

$$\varphi_1 = x, \tag{10.43}$$

$$\varphi_n(x) = \frac{1}{n}\left[(2n-1)x\varphi_{n-1}(x) - (n-1)\varphi_{n-2}(x)\right]. \tag{10.44}$$

The roots of φ_n are not generally analytically soluble so we have to apply a root-finding algorithm. Our Newton–Raphson algorithm will perform the job nicely. We can use Newton–Raphson rather than the secant method as we can determine the analytical first-ordered derivative of the φ_n from Equation (10.44). Explicitly the recursion relation for the derivatives are

$$\varphi_n'(x) = \frac{n}{x^2-1}\left(x\varphi_n(x) - \varphi_{n-1}(x)\right). \tag{10.45}$$

To speed up our root searches, we use the fact that the first guess x_0 for the i th root of a n-order polynomial φ_n can be given by

$$x_0 = \cos\left(\pi\frac{i-1/4}{n+1/2}\right). \tag{10.46}$$

As Equation (10.46) gives us a relatively decent estimate of the root we do not need the robustness of a bisection method in our root search. After we get the abscissas x_m via the root search to some precision, we compute the appropriate weights by

$$w_m = \frac{2}{\left(1-x_m^2\right)\left[\varphi_n'(x_m)\right]^2}. \tag{10.47}$$

Once the weights and abscissas are computed for a N point quadrature, we can approximate an integral over any interval $[a,b]$ by

$$\int_a^b f(x)dx \approx \frac{b-a}{2}\sum_{m=1}^{N} w_m f\left(\frac{b-a}{2}x_m + \frac{a+b}{2}\right). \tag{10.48}$$

As mentioned, the class `Legendre`, implemented in *Quadrature.cpp*, is designed to perform the Gauss–Legendre quadrature method for any given function that accepts a **`double`** type argument and returns the result as a **`double`** type. It either loads the weights and corresponding abscissas from the look-up tables defined in *GaussKnotsWeights.h* or computes them from the Legendre polynomials; see the implementation of the `initialise` and `compute_x_w` member functions. The latter of these two functions finds the coefficients of the *N*th Legendre polynomial using the recurrence relation defined by Equation (10.44), and once we have those coefficients we can then find its roots. As the cosine function gives a relatively good estimate of the root, the Newton–Raphson method will always converge to the required root of the polynomial. It is not obvious that the code here calculates the polynomial function and its derivative for the given x_m. It uses the array of coefficients, $P1$, to build up the function and its derivative for use with the Newton–Raphson root search. One may find it instructive to perform the calculations manually for small N or have the program print out the polynomial values as a check. Once the roots are found to the desired precision or we have performed a specified number of iterations of the root search, we store them in a `vector`. The corresponding weights are calculated and stored to a second `vector`. Those vectors are combined into the return value, which in turn are stored to the relevant data members. As an aside, could this code be improved?

The knots and weights are then used by the member function `integrate` to integrate the function across the domain specified using the Gauss–Legendre method.

The application code found in *gauss_legendre.cpp* integrates the exponential function over the range [1,4] for an increasing number of knots. The integration has the following analytical solution:

$$\int_1^4 e^x dx = e^4 - e^1 = 51.8798682\ldots \tag{10.49}$$

The program prints the number of points used in the quadrature, the value obtained, and the error in the solution. After you compile and run this program you should find that the error reduces

rather rapidly as we increase the number of points used. If you were to compare the accuracy achieved to the number of function calls required for Gauss–Legendre quadrature to the other quadrature methods developed in Chapter 5. Notice that because we apply the Gauss–Legendre quadrature to a finite integrand it is possible to build a composite rule for the quadrature.

10.5 GAUSS–LAGUERRE QUADRATURE

One limitation to the Gauss–Legendre quadrature is that it only applies to integrals with finite limits. In physics, it often happens that a physically significant quantity can be given by the semi-infinite integral

$$I = \int_0^\infty g(x)\,dx. \tag{10.50}$$

For the integral to be finite $g(x)$ must vanish more rapidly than the inverse of x (c.f. convergence of an infinite sum). One way to ensure this condition is to recast where possible the integrand function as

$$I = \int_0^\infty g(x)\,dx = \int_0^\infty e^{-x} f(x)\,dx, \tag{10.51}$$

as the exponential weight vanishes more quickly than $1/x$. We now have an integral in the form of the left-hand side of Equation (10.32). As before, we need to find a set of orthogonal polynomials that will satisfy these limits and the weighting function.

This work has already been done before and the set of polynomials we need are the Laguerre polynomials. We can define the Laguerre polynomials recursively, defining the first two polynomials as

$$\varphi_0(x) = 1 \tag{10.52}$$

and

$$\varphi_1(x) = 1 - x, \tag{10.53}$$

then using the following recurrence relation for any $n \geq 1$:

$$\phi_{n+1}(x) = \frac{1}{n+1}\left[(2n+1-x)\phi_n(x) - n\phi_{n-1}(x)\right]. \qquad (10.54)$$

Following the same strategy as before we find the roots of the Laguerre polynomial. To do this we note that

$$\varphi_n'(x) = \frac{n}{x}\left(\varphi_n(x) - \varphi_{n-1}(x)\right). \qquad (10.55)$$

Unfortunately, there is no other formula for the estimation of the roots so you may find it useful to plot the Laguerre polynomials and find estimations for the roots manually. A numerical recipes handbook may offer more guidance here; in our library code, we use the formula obtained from Stroud & Secrest, *Gaussian Quadrature Formulas*.

Once the roots are found they are used to obtain the weights using the relation

$$w_m = \frac{x_m}{(n+1)^2\left[\phi_{n+1}(x_m)\right]^2}. \qquad (10.56)$$

Note that the denominator contains a factor of the polynomial squared evaluated at the abscissa rather than the derivative at the abscissa as with the Legendre weights.

Of course, the weights and abscissas have been tabulated and published elsewhere and could be implemented, for example, as look-up tables or readable from an external file.

As the upper limit of the integral is infinity, we cannot derive a direct composite formula for the quadrature. If higher accuracy is needed than the Gauss–Laguerre quadrature can achieve, you can always separate the semi-infinite region into two, using a Gauss–Legendre quadrature (perhaps a composite version) up to some finite limit, and then a Gauss–Laguerre quadrature from that limit up to infinity. Though, one must remember to adjust the variables for the change in the lower limit of the integration.

There exists several other Gaussian quadrature methods for different integration limits and weighting functions. Of note are the

Gauss–Hermite, Gauss–Chebyshev, and Gauss–Jacobi quadrature methods, which you should lookup. Though different they all share the same common algorithm—define the set of polynomials to use, find the roots of those polynomials, use those roots to compute the corresponding weights and abscissas, and finally compute the quadrature for a given number of points.

EXERCISES

10.1. Either using the code provided or with your own program compute

$$\int_{-1}^{1} x^m \, dx$$

for $m = 0,1,...,15$ using Gauss–Legendre quadrature to double precision. If the code is correct how accurate should the quadrature be for the appropriate number of points used?

10.2. Compare the effort required to compute

$$\int_{\mu-\sigma}^{\mu+\sigma} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx$$

to 10 significant figures of accuracy for the trapezoidal rule, Simpson's rule, and Gauss–Legendre quadrature. You may find it illustrative to plot the relative error against the number of points used for each method.

10.3. Write a program to use the Gauss–Laguerre quadrature. To test that the `Laguerre` class is correct, evaluate the integral

$$\int_{0}^{\infty} e^{-x} dx$$

It should equal the sum of the weights for the number of points used.

10.4. In Planck's treatment of black body radiation, the following integral appears:

$$\int_0^\infty \frac{x^3}{e^x - 1} dx$$

Evaluate the integral ensuring 10 significant figures of accuracy.

10.5. The integral

$$f(\theta) \approx -\frac{2m}{\hbar^2} \int_0^\infty rV(r) \sin(2kr \sin(\theta/2)) dr$$

appears in the theory for the cross-section of a quantum scattering event. It describes the force felt by a quantum particle as it interacts with the scattering potential V as a function of the incident angle θ. If the particle is an electron scattering from an atomic nucleus then the scattering potential is given by

$$V(r) = \frac{1}{4\pi\varepsilon_0} \frac{Zq^2}{r} e^{-r/r_0}$$

where Z is the proton number of the nucleus, q is the proton charge and r_0 is the so-called screening length. Plot f as a function of θ for an atom of your choice; use the Bohr radius as the screening length.

CHAPTER 11

ADVANCED ODE SOLVER AND APPLICATIONS

In this chapter, we explore a more advanced ODE solver and how we can apply it to solve some "difficult" physics problems from finding chaos in a driven pendulum to sending a spaceship to the Moon (and beyond) to the wavefunctions of electrons in an arbitrary electrical potential. We will also show how to use the solver in combination with the Fast Fourier Transform subroutine in Chapter 7 to analyze the frequency spectrum of the Van der Pol oscillator, which will provide the guide for you to analyze the spectrum of the chaotic pendulum.

11.1 RUNGE–KUTTA–FEHLBERG

In Chapter 6, we explored the use of the finite difference method to solve ODEs. During this discussion, we developed a technique to make the step sizes adapt to the local nature of the solution by halving the step size when it was too large and generating too much local error then doubling the step size when it was too small and wasting computational effort. Although this method is effective there is a

better way of making the step size adaptive, rather than just halving or doubling its length.

Erwin Fehlberg published several adaptive steps Runge–Kutta methods in two NASA technical reports in 1968 and 1969. In Fehlberg's algorithm, two Runge–Kutta methods of different order are run simultaneously. At each step, the lower-ordered method is computed first, producing an estimate of the solution, y_{n+1}. Next, the second, higher-ordered method is computed with more function evaluations producing the estimate $\hat{y}_{n+1}$ for the same step size. The difference between these two methods gives an estimate of the local error for the step size used. If this estimation of the error is within the prescribed tolerance the step is accepted, and the solution is advanced. If not, the step is rejected, and the process is repeated with a reduced step size. Whenever a step is accepted the next step size is estimated using the values obtained from y and $\hat{y}$, and we use the more accurate value of $\hat{y}$ as our initial value for the next step; more on this shortly.

At first glance, this may not seem like we are saving much on computational resources; although the steps will be adaptive, we must make several extra function evaluations at each step in order to compute the two Runge–Kutta methods. However, the beauty of Fehlberg's algorithms is that he found coefficients such that the two methods share function evaluations and only a few extra evaluations are required for the higher-ordered method. One such Fehlberg algorithm is based on the classic fourth-ordered Runge–Kutta, with a fifth-ordered Runge–Kutta used as the higher-ordered estimate. The informed reader may now be guessing that this is why one of the differential equation solvers in MATLAB/OCTAVE is named "ode45."

In the Runge–Kutta–Fehlberg fourth-fifth (RKF45) algorithm each accepted step requires a total of six intermediary function evaluations: four for the fourth-order Runge–Kutta, and two more for the fifth-order Runge–Kutta. Fehlberg's equations for the RKF45 method are as follows: the intermediary function evaluations are given by

$$k_0 = f\left(t_0, y_0\right), \tag{11.1}$$

$$k_1 = f\left(t_0 + \frac{h}{4}, y_0 + \frac{h}{4}k_0\right), \tag{11.2}$$

$$k_2 = f\left(t_0 + \frac{3h}{8}, y_0 + \frac{3h}{32}k_0 + \frac{9h}{32}k_1\right), \tag{11.3}$$

$$k_3 = f\left(t_0 + \frac{12h}{13}, y_0 + \frac{1932h}{2197}k_0 - \frac{7200h}{2197}k_1 + \frac{7296}{2197}k_2\right), \tag{11.4}$$

$$k_4 = f\left(t_0 + h, y_0 + \frac{439h}{216}k_0 - 8hk_1 + \frac{3680h}{513}k_2 - \frac{845h}{4104}k_3\right), \tag{11.5}$$

$$k_5 = f\left(t_0 + \frac{h}{2}, y_0 - \frac{8h}{27}k_0 + 2hk_1 - \frac{3544h}{2565}k_2 + \frac{1859h}{4104}k_3 - \frac{11h}{40}k_4\right), \tag{11.6}$$

where f is the function defining the differential equation, h is the step size, t is the independent variable, and y is the dependent function. The index zero refers to the values at the beginning of a given step. Using these definitions for the intermediary function evaluations the fourth-ordered Runge–Kutta is written as

$$y_{n+1} = \hat{y}_n + h\left(\frac{25}{216}k_0 + \frac{1408}{2565}k_1 + \frac{2197}{4104}k_3 - \frac{1}{5}k_4\right) \tag{11.7}$$

and the fifth-ordered Runge–Kutta is written as

$$\hat{y}_{n+1} = \hat{y}_n + h\left(\frac{16}{135}k_0 + \frac{6656}{12825}k_1 + \frac{28561}{56430}k_3 - \frac{9}{50}k_4 + \frac{2}{55}k_5\right), \tag{11.8}$$

where $\hat{y}_n$ is the value of the previous, successful step. As stated, we can estimate the local error in the step by finding the difference between Equations (11.8) and (11.7). Performing this operation and after some rearrangement, we arrive at the following expression for the error in the local step

$$\sigma \equiv \hat{y}_{n+1} - y_{n+1} = h\left(\frac{1}{360}k_0 - \frac{128}{4275}k_2 - \frac{2197}{75240}k_3 + \frac{1}{50}k_4 + \frac{2}{55}k_5\right). \tag{11.9}$$

Note that with this expression for the estimate of the local error we do not have to calculate the expression for y_{n+1}, Equation (11.7), to compute σ. Of course, $\hat{y}_{n+1}$ still needs computing.

Seeing as we know how the error behaves with step size (*fourth*-order Runge–Kutta) we should be able to use this information to estimate the next step size from the step we have just computed. To do this, we note that we can write

$$\frac{|h|\varepsilon}{|\sigma|} = \left(\frac{h'}{h}\right)^4 \tag{11.10}$$

where ε is the global error tolerance we want in our solution thus $|h|\varepsilon$ is our desired local error tolerance, h' is an estimate of the "ideal" step that will produce the desired local error tolerance, and h is the step size for which we have just calculated σ. Notice that the absolute value of the step size is used to take into account a reverse integration.

Rearranging Equation (11.10) for this "ideal" step size we obtain

$$h' = \alpha h\left(\sqrt[4]{\frac{|h|\varepsilon}{|\sigma|}}\right), \tag{11.11}$$

where we have included a parameter α that takes values in the range $[0,1]$ as a factor, we can adjust to provide a more conservative estimate of the next step. Remember that Equation (11.11) is an *estimate* of the "ideal" step found from considering the error behavior, *not* the absolute value h^4 itself. Erring on the side of caution we assume that it produces a step size that is (slightly) larger than the "ideal" thus warranting the inclusion of the adjustment parameter. Setting $\alpha \approx 0.9$ typically gives reasonable estimates but can be adjusted if we find the error tolerance in the results to be unsatisfactory as it does depend on the properties of the ODE we are trying to solve.

We should also be conservative in our approach to adapting the step size and as before we introduce maximum and minimum step sizes to take account of any singularities, discontinuities, or asymptotes, that is, where the differential equation may change in nature. In addition to these limits, we should also be conservative by how much the step size changes from one step to the next. If we find that the algorithm wants to increase the step size by more than a factor of

ten, say, then we should be cautious and limit the increase to a factor of ten or less. Similarly, if the step size is suddenly decreased by a large factor, we should also be cautious here and set an appropriate limit. These four limits, maximum and minimum step sizes, and the factors of maximum increase and decrease, should be experimented with for different, differential equations as, if you will excuse the pun, one-size does not fit all, and it is likely we are unable to predict the nature of the (numerical) solution to the differential equation before-hand. One general strategy that might be applied is to make the maximum and minimum step sizes some fraction of the integration interval.

The class `RKF45`, implemented in *ODESolvers.cpp*, is designed to perform Fehlberg's adaptive method as we have just discussed. As this class is derived from the `ODESolver` base class it is the interface is similar to the other ODE solvers we have used before and can autonomously handle first and second-order ODEs. Read through the code and make sure you understand the variables and the task they perform. Here we have defined each of the coefficients in Equations (11.1) through (11.9) as parameters to ensure that they are not changed by the program, and to make the code more readable. Also, we have defined them as fractions to ensure the best possible precision. Additionally, all the conservative precautions we have taken are given as adjustable class parameters so they can be easily changed in the application code if necessary.

To check that the algorithms behave as expected and produce tolerable errors in their numerical solutions write a program to compute the dynamics of simple harmonic motion. Does the algorithm adapt the step size as expected? Does the algorithm remain numerically stable, and if so for how long? Indefinitely perhaps?

11.2 PHASE SPACE

Normally, and quite intuitively, we plot the state of a dynamical system, for example, the displacement of a pendulum as a function of time. This will quite naturally tell us things like the amplitude of the oscillations and their frequency. However, we can create a plot

that has no explicit dependence on time, one in which we plot the position of the pendulum, say, against its velocity (or more generally its momentum). This plot is known as the phase space of the dynamical system. As time advances, a point in phase space representing the current *phase state* of the dynamical system will shift, tracing out a *phase trajectory*. When we plot several phase trajectories for different initial conditions, say, or for different parameters, we call that a *phase portrait*.

To illustrate, consider the motion of an undamped, mass-on-a-spring undergoing simple harmonic motion. We all know that the displacement of the mass is described by a sinusoidal function and that this function lags that describing the velocity of the mass by one-quarter of a cycle. In other words, if $x = \sin(t)$ then $\dot{x} = \cos(t)$. Plotting the velocity $\dot{x}$ against the displacement x we would obtain a circle in phase space. This circle tells us that the motion must be oscillatory and that the energy of the system is conserved. What would happen to the phase portrait if $x = \sin(\omega t)$ for $\omega \neq 1$? What effect does a change in the amplitude of the oscillations have on the phase portrait? How might the phase space portrait look if we were considering damped oscillations, for example, energy lost through air resistance? You should by now be thinking of how you can show your answer to those questions to be correct using the (computational) tools at your disposal.

For reasons that will have become apparent, the origin of the phase portrait of the *damped* simple harmonic oscillator is called an *attractor*. The phase trajectory *spirals into* the origin as the oscillator *loses* energy. If we were to follow that trajectory in reversed time, then we would see that the phase state of the system spirals out from the origin. In other words, the oscillator is gaining energy and thus, in this case, we must have a driving force.

We have already seen in Chapter 6 that when both driving and damping forces are present an oscillatory system goes through a transient phase before settling into a steady state. The nature of the transient phase is dependent upon the initial conditions of the system, whereas the steady-state is not. How does this look on a phase space portrait, and what does *limit cycle* mean? How would resonance be detected using phase space?

11.3 VAN DER POL OSCILLATOR

Balthasar Van der Pol was a Dutch Physicist and electrical engineer who experimented with some novel electronic circuits containing triodes (vacuum tubes) in the 1920s. One of those circuits is now known as the Van der Pol oscillator that is described by the non-linear differential equation

$$\ddot{y} = -y - \mu\left(y^2 - 1\right)\dot{y}, \tag{11.12}$$

where y is some position coordinate as a function of time, and μ is a parameter indicating the strength of the non-linear damping term. When $\mu = 0$, we have simple harmonic motion. Equation (11.12) describes self-sustaining oscillations in which energy is fed into small oscillations and removed from large oscillations; to see this from the equation consider the damping term when $y > 1$ and when $y < 1$.

11.3.1 Van der Pol in Phase Space

The Van der Pol oscillator equation is difficult to solve analytically due to the non-linear damping term but can be tackled using perturbation theory. However, this only works when μ is small, that is, $\mu \ll 1$, which is hardly interesting at all. This is where our numerical integrator steps in to provide a solution.

Figure 11.1 shows the phase space portrait of the Van der Pol oscillator for various values of the parameter μ. To obtain these plots we set the initial values of the displacement and the velocity to 2 and 0, respectively, and the tolerance was set to $\varepsilon = 10^{-5}$. Here we show that for $\mu = 0$, we recover simple harmonic motion; the phase plot is a circle. Note that the initial values were chosen so that they lie on the limit cycle for the oscillations.

Here we can see the effect of the non-linear damping term on the phase trajectory of the oscillator. You may find it illustrative to also plot the regular displacement–time graphs to match up corresponding points from the phase portrait.

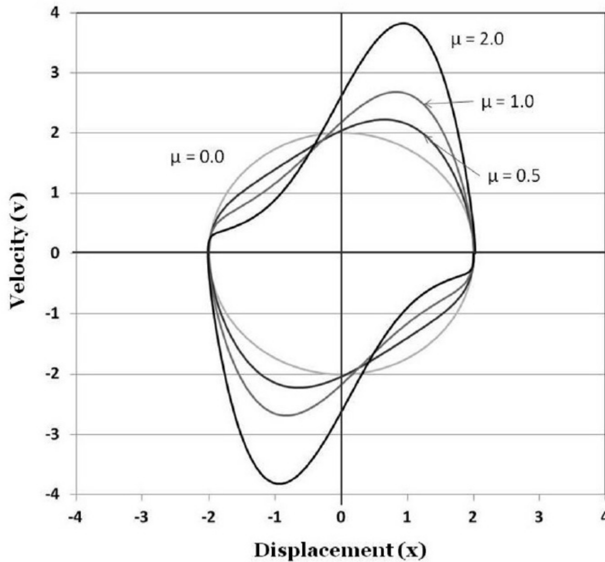

FIGURE 11.1: Phase portrait for the Van der Pol oscillator for various values of μ.

11.3.2 Van der Pol FFT

Normally, we use Fourier transforms to gain (extra) insight into experimental data that is some function of time, say, by converting it into a function expressed in terms of frequency. We can also apply this analysis to numerical data, such as that which has been synthesized through computation.

We have just seen that the phase space diagram of the Van der Pol Oscillator is an alternative and useful way to study the behavior of the oscillator. The Fourier transform of the data will augment our understanding of that behavior.

If you recall, the Fast Fourier Transform (FFT) subroutine requires (time) data that is set at constant increments, but our Runge–Kutta–Fehlberg solver is an adaptive step algorithm. Rather than try to interpolate the data from the solver we shall instead modify the routine to store values at constant time intervals. We can then pass this data directly to the FFT subroutine without the need to change it in any way.

The file `RKF45` class contains a data member to encode when we want the algorithm to store data at regular intervals but still employ

an adaptive routine. In essence, we check to see if we have reached our target time increment, and if we have, we increment our data counter, write the data to an array, and move the goal to the next desired data point. If not, we continue the integration without storing any data. However, before moving on we include a check to see if the next step will take us past our current target and adjust the step such that it will hit the target. In this way, we should maintain an error that is less than the desired tolerance. As these oscillations tend to have an initial transient phase before settling into a steady-state we should, in general, set a non-zero initial goal to ensure we start taking data from the steady region. We run the integration until we have filled our data array, which we can then pass to an FFT function for analysis (conveniently we already have one written).

When sampling the data in this way we must remember that the FFT subroutine works best when the time increments are commensurate with the period of oscillations. Put another way, we should ensure that we are not aliasing our data, and we fit an (exact) integer multiple of oscillations in our data array. We recall that our time increment (or sample rate) is given by

$$\Delta t = \frac{kT}{N}, \tag{11.13}$$

where k is some integer, T is the time period of the oscillations, and N is the number of data points we will use in the FFT function.

By far the easiest way to obtain T is to use the displacement–time graph to estimate a value; take the period for several oscillations and divide through by this number. For $\mu = 2$, we found the period of oscillations to be around 7.631 s. As we are taking 512 data points (this number is *not* arbitrary; a power of two is required for the FFT subroutine) we should sample with a time increment of, say, $\Delta t = 0.23846875$ seconds to include 16 full oscillations in our data array. This choice in the number of oscillations is again not arbitrary. By choosing a power of two we get an exact integer number of points per oscillation (give or take some small error in the calculation of the time period), in this case, $2^9 / 2^4 = 2^5 = 32$.

You will know if your calculation of the period is accurate by the quality of the Fourier spectrum. Figure 11.2 shows the Fourier

spectrum for the Van der Pol Oscillator with $\mu = 2$, simulated with our RKF45 class with a tolerance of 10^{-5}. Note that the intensity axis is logarithmic such that the decrease in intensity from one peak to the next is not linear but exponential. Here we have computed a decent time increment as the peaks are sharp, almost delta functions, and the "background" spectrum is small ($<10^{-4}$). The broadening of the peaks is most likely due to bin leakage as the period we have calculated is close to actual but not quite exact. It may also be caused in part by the numerical errors in simulating the oscillations. Of course, this can easily be investigated by making changes to the tolerance in the RKF45 algorithm, and small changes to the period.

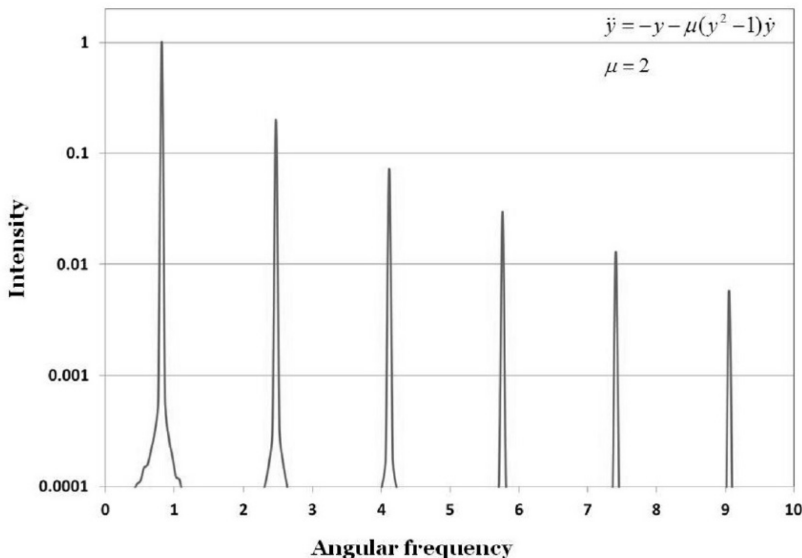

FIGURE 11.2: Fourier spectrum of the Van der Pol Oscillator, with damping parameter $\mu = 2$.

11.4 THE "SIMPLE" PENDULUM

You all probably remember the simple pendulum experiment from your physics classes at school. It provides a practical introduction to how to deal with experimental errors (e.g., timing several

oscillations to obtain a more accurate measure of the period) and how to use approximation to simplify the mathematics. We can derive the differential equation for the pendulum from the geometry of the system such that

$$\ddot{\theta} = -\frac{g}{l}\sin\theta, \tag{11.14}$$

where θ is the angular position of the pendulum, g is the strength of gravity at the Earth's surface, and l is the length of the pendulum. Here we assume that the oscillations are free, in that there is no driving force (other than gravity), and there is no damping due to frictional or resistive forces. For the rest of this section let us also assume the pendulum is rigid. As Equation (11.14) is rather difficult to solve analytically (if not impossible?) the usual trick is to assume the small-angle approximation that is $\sin\theta \approx \theta$ for $\theta \ll 1$ (in radians), and we obtain the simple harmonic oscillator equation. However, our numerical solver has no issues tackling this equation head-on.

11.4.1 Finite Amplitude

With our new description of phase space, we should be able to clearly visualize the behavior of the pendulum, specifically seeing at what angles the small-angle approximation holds. For small angles, we should see a circular phase trajectory that will morph into something different as we increase the amplitude of the oscillations.

We have two approaches to consider in how to vary the amplitudes. We can mimic what we would do given a physical pendulum. That is, we monitor the trajectory by directly varying the initial angle of release and setting our initial velocity to zero. Or we consider the total energy of the pendulum, that is its potential energy plus its kinetic energy, and work out the angular velocity of the pendulum as a function of the total energy when the angle is zero, that is, the angular velocity at the bottom of the swing, and use that as our initial conditions. We then vary the total energy (which will vary the amplitude) to see what affect this has on the phase trajectory. This second method is more practical in terms of phase space as the area encompassed by the phase trajectory is proportional to the energy in the system.

We recognize for any (mechanical) system the total energy is given by

$$E = T + V\,, \tag{11.15}$$

where

$$T = \frac{1}{2} m l^2 \dot{\theta}^2 \tag{11.16}$$

is the kinetic energy and

$$V = mgl\left(1 - \cos\theta\right) \tag{11.17}$$

is the (gravitational) potential energy of a pendulum. When the pendulum is at the bottom of its swing, we have $\theta_0 = 0$ and

$$E = T = \frac{1}{2} m l^2 \dot{\theta}_0^2, \tag{11.18}$$

as this is where we have defined our zero-potential energy. Rearranging Equation (11.18) for angular velocity yields

$$\dot{\theta} = \sqrt{\frac{2E}{ml^2}}. \tag{11.19}$$

Using units such that $g = l = 1$, and $m = 0.5$ we obtain the phase portrait of the simple pendulum as shown in Figure 11.3. It is of note that using these units we set our unit of time as $\sqrt{l/g}$. Here we have computed the phase trajectories for total energies of 0.25 to 1.5 in steps of 0.25. Here the units of energy are dictated by those we chose for the other parameters. As expected with lower energy (smaller amplitude) the pendulum behaves approximately like a simple harmonic oscillator. As we increase the energy (larger amplitude) that approximation no longer holds with the phase trajectory, elongating along the θ axis. The phase trajectory when $E = 1$ is called a *separatrix* that, as we can see from Figure 11.3, defines a fundamental change in behavior of the pendulum. It is where the pendulum has sufficient energy such that the oscillation becomes circular motion; the pendulum goes over the top.

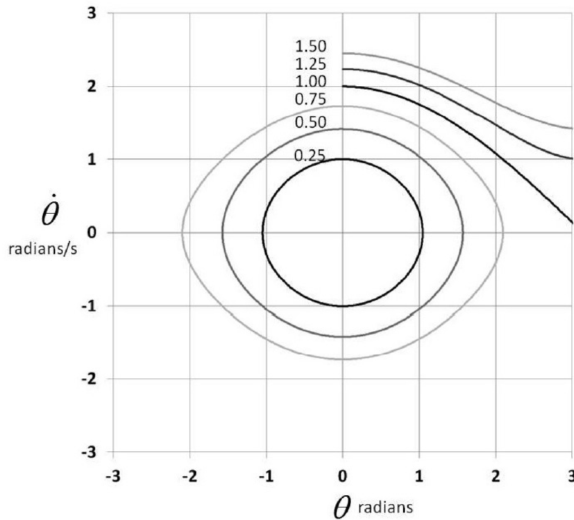

FIGURE 11.3: Phase portrait of the simple pendulum for various energies (amplitudes).

11.4.2 Utter Chaos?

Now that we have the phase description of the simple pendulum under our belts let us consider a more realistic system. As with the mass-on-a-spring system, we introduce both a driving force, F_D, and a resistive, drag force, F_R, into our equations. The differential equation governing the motion of the pendulum is then given by

$$\ddot{\theta} = -\frac{g}{l}\sin\theta + \frac{F_D}{ml} + \frac{F_R}{ml} \tag{11.20}$$

where we have introduced the mass of the pendulum m into our equation. Here we consider that the mass of the pendulum is located at the very end of its length. To keep things simple (relatively speaking) let us assume the driving force is described by a periodic function such that

$$F_D = f_0\cos(\omega_0 t)\,, \tag{11.21}$$

where f_0 is the strength of the force and ω_0 is its frequency, and the resistive force can be described by

$$F_R = -\rho v = -\rho l\dot{\theta}, \tag{11.22}$$

where ρ is the coefficient of the drag force, and v is the tangential velocity of the pendulum's mass.

If we rewrite Equation (11.20) in a dimensionless form, where we again choose $g = l$ (not necessarily equal to one) such that the unit of time is $\sqrt{l/g}$ we obtain

$$\ddot{\theta} + q\dot{\theta} + \sin\theta = b\cos(\omega_0 t), \tag{11.23}$$

where $q = \rho / m$ and $b = f_0 / ml$ are adjustable parameters, along with the driving frequency ω_0.

Depending on the relative values of p, b, and ω_0 the motion of the pendulum can either be periodic or chaotic as shown in Figure 11.4. The discontinuities arise because we map the angle back into the physically valid range. When writing a program to drive this simulation we must remember that our angle θ can only exist in the range $[-\pi, \pi]$. To provide this functionality for all of the ODE solvers we have developed, the `ODESolver` base class has member functions that can be used by any derived class to wrap the independent variable to $[-\pi, \pi]$. Note that the member function `fullSolveWrapped` uses the single-step member function `solve` within a loop to integrate over the entire domain.

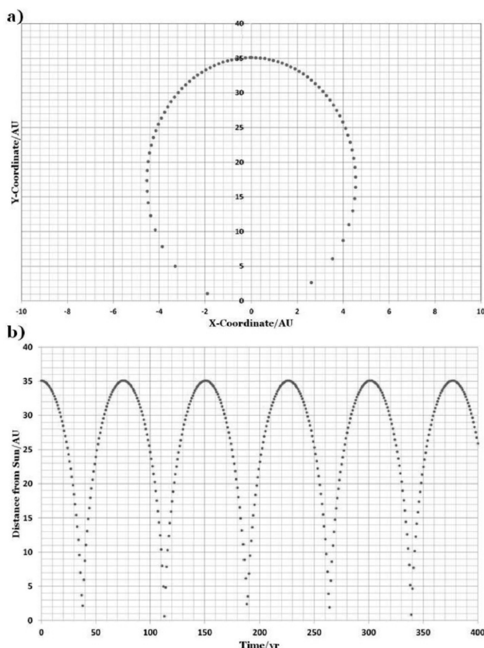

FIGURE 11.4: The angle as a function of time and the phase trajectories of the driven pendulum showing periodic motion (top) and chaotic motion (bottom) for the parameter values shown.

An exercise is provided for the reader to investigate the relative values of the parameters required to bring about chaos. Note that whole books are dedicated to the study of chaos and chaotic motion, and it is still under much academic research. One thing to remember is that you can always analyze its Fourier spectrum.

11.5 HALLEY'S COMET

Halley's Comet is probably the best known short-period comet and is visible from Earth with the naked eye every 75–76 years. Halley's returns to the inner Solar System have been observed and recorded by astronomers since at least 240 BC. Clear records of the comet's appearances were made by Chinese, Babylonian, and medieval European chroniclers but were not recognized as reappearances of the same object until much later. In 1705, English astronomer Edmond Halley was the first to calculate the comet's periodicity and was rewarded with having it named after him. Halley's Comet last appeared in the inner Solar System in early 1986.

Halley's Comet has a highly elliptical, planar orbit with large differences in its velocities at the aphelion (furthest distance) and the perihelion (closest distance) of its journey around the Sun. The equation governing the comet's trajectory, that is, the force, F, acting on the comet, is Newton's Law of gravitation,

$$F = -\frac{GMm\hat{e}_r}{r^2} = -\frac{GMm\underline{r}}{r^3}, \tag{11.24}$$

where G is the universal gravitational constant, M is the solar mass, m is the mass of the comet, $\hat{e}_r = \underline{r} / r$ is a unit vector that points from the center of the Sun to the center of the comet, and r is the distance between the center of the Sun and the center of the comet. Here we assume that the influence of other bodies in the solar system is insignificant compared to the gravitational pull of the Sun.

We usually give the units of Equation (11.24) in SI form. That is, distance is measured in meters, time in seconds, and mass in kilograms. This makes the universal gravitational constant $G = 6.67384 \times 10^{-11}\text{m}^3\ \text{kg}^{-1}\ \text{s}^{-2}$, the solar mass $M = 1.9891 \times 10^{30}$ kg, and the aphelion distance is 5.28×10^{12} m. Using the vector form

of Equation (11.24), we must calculate the distance cubed. This is going to lead to precision problems if we use SI units and a change of units is required. The first thing to note is that both G and M are constants thus we can write $G_S = GM$ as the universal gravitational constant *per solar mass*. We now need to choose the units for length and time so that both the solar-comet distance r, and our gravitational constant per solar mass G_S have exponents that ideally reduce to zero, and certainly no more than one. Instead of arbitrarily choosing some units let us use some that are more natural. The astronomical unit, AU, defines the mean distance between the Earth and the Sun, which has a value of $1.49597871 \times 10^{11}$ m. This makes the aphelion distance equal to 35.1 AU so this appears to be a good choice; remember this is the greatest distance from the Sun.

The sidereal (pronounce si-dea-re-al) year is defined as the orbital period of the Earth around the Sun relative to the background of "fixed" stars. The name sidereal comes from the Latin "sidus" meaning "star." It is used often in astronomy and contains 365.256363 days equal to 3.1558150×10^{7} s. If we define our computer-friendly units of length, mass, and time as the astronomical unit (AU), solar mass (M), and sidereal year (yr) respectively then $G_S = 39.489 \text{AU}^3 \text{ M}^{-1} \text{ yr}^{-2}$. (To obtain this value you multiply G by M in SI units, then multiply by the square of the number of seconds per year, finally diving by the cube of the number of meters per AU.)

To solve Equation (11.24), which is a second-ordered ODE, and thus determine the trajectory of Halley's Comet we need to know two pieces of (initial) information. That is, we need to know the comet's position and (instantaneous) velocity at a particular time, which we take to be our origin in time. As we already know the aphelion distance, we can use that position to start the integration given that the aphelion velocity is 912 ms^{-1}. After making the appropriate changes to the units and writing a program that uses the RKF algorithm for a seconded-ordered ODE with *two* dependent functions (x and y coordinate of the comet) we plot the comet's trajectory in Figure 11.5. Figure 11.5(a) shows the trajectory for a single orbit where each data point is computed at an interval of one year; note the difference in the scales of the x and y axes. Figure 11.5(b) shows the *distance* from the Sun plotted as a function of time for several orbits; again, the data points are 1 year apart.

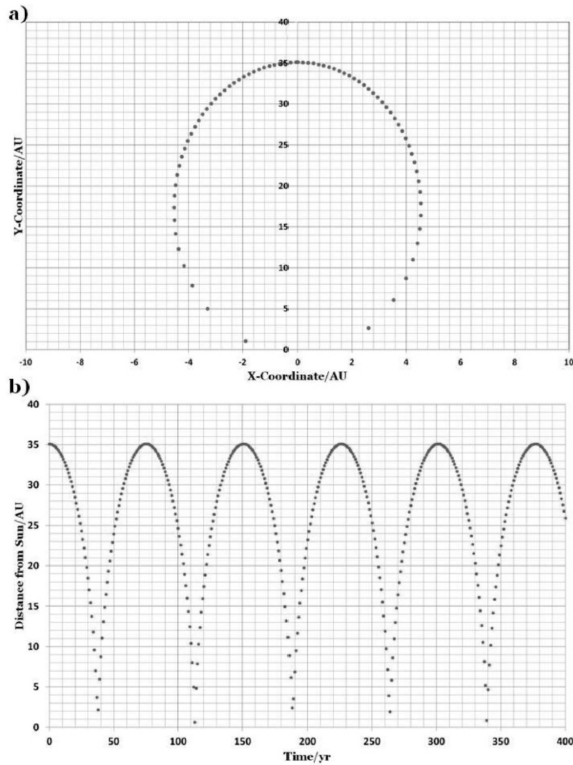

FIGURE 11.5: Trajectory of Halley's comet (a) looking down on the orbital plane, where the Sun is located at the origin; (b) comet's distance from the Sun as a function of time. In both the points shown are spaced one year apart.

Note that if you tried to solve this differential equation using a constant step length algorithm you will find that unless the step length is small, that is, much less than a year, the solutions are very unstable orbits with the comet shooting off somewhere as it approaches and goes past the Sun—this clearly does not happen.

11.6 TO INFINITY AND BEYOND

Humans have long wondered what is out there among the stars. Exploration, it seems, is in our nature. Our first goal is to get to our nearest neighbor in the solar system; the Moon. The first obstacle

to overcome is how we get off the planet in the first place. This task is left for the reader (see Exercise 5). Assuming we have achieved a stable orbit about our planet, our next obstacle is to navigate to the moon. Unlike the movies, we do not have the luxury of an endless supply of fuel and are reliant on short burst thrusters only, meaning that the motion of our spaceship is (mostly) dictated by Newton's Law of gravitation. The speeds we consider are nowhere near relativistic, neither are the gravitational forces, such that Newton's Laws are an adequate description of the physics. We pick our frame of reference as the Earth–Moon system; this frame of reference is in orbit about the Sun and as such we can consider the relative motion of the Earth, Moon, and spaceship independently from their motions about the Sun (and the Sun's motion about the galaxy, the galaxy's motion about the local cluster, and so on). With these descriptions in place let us go to the Moon.

To start let us just consider the Earth–Moon system. Normally, we state this as the Moon orbiting the Earth but in fact they orbit each other about some, common center-of-mass (COM). Note that this is true of any two-body system orbiting one another. It makes sense, therefore, to fix our origin at this COM. In general, orbital trajectories are elliptical with one of the foci located at the origin of the system. However, the eccentricity of the Earth–Moon orbit is sufficiently small that their trajectories can be assumed to be circular. In addition to this, their orbits are planar, that is they can be sufficiently described by two spatial coordinates. If d is the center-to-center distance between the Earth and the Moon, then the distance of the center of the Earth to the COM is

$$r_E = \frac{m_M}{m_M + m_E} d \tag{11.25}$$

and the distance of the center of the Moon to the COM is

$$r_M = \frac{m_E}{m_M + m_E} d, \tag{11.26}$$

where m_E is the mass of the Earth and m_M is the mass of the Moon. If you are wondering how we arrive at these equations, then the trick is to consider turning moments.

The Earth–Moon system rotates about its common COM with a sidereal orbital period T. If the Moon lies on the positive x-axis at $t = 0$, then

$$\varphi_M = \omega t \tag{11.27}$$

and

$$\varphi_E = \omega t + \pi \tag{11.28}$$

where ϕ_M is the angular location of the Moon, ϕ_E is the angular location of the Earth at time t, and ω is the angular frequency of the orbit; the angle is measured from the x-axis. To remain in a circular motion a body must be constantly accelerated toward the center of motion, with an acceleration of magnitude $\omega^2 r$, where r is the length of the radius of the motion. For the Earth that acceleration is provided for by the gravitational force between the Earth and the Moon such that

$$\frac{Gm_M}{d^2} = \omega^2 r_E. \tag{11.29}$$

After substitution of Equation (11.25) and some manipulation, we arrive at Kepler's third law for planetary motion

$$\omega^2 = \frac{4\pi^2}{T^2} = \frac{G(m_M + m_E)}{d^3}. \tag{11.30}$$

We would arrive at the same relationship if we had first considered the acceleration of the Moon. Rigorously speaking this is not really proof of Kepler's third law as we have assumed circular orbits and more generally, we should consider elliptical orbits. However, Equation (11.30) does hold for elliptical orbits but in this case, d would be the semi-major axis of the ellipse rather than the center-to-center distance.

Equations (11.25) through (11.30) now form a practical description of the relative motion of the Earth and Moon about each other. As stated, our spaceship is currently in a stable orbit about Earth. Let's assume that this orbit is 500 km *above the surface* of the Earth. When the spaceship reaches some angular location θ in its orbit, it fires its thrusters and accelerates to some speed v in a direction tangent to the orbit at θ. Here we will assume that this thrust

acceleration is instantaneous in comparison to the total journey time; you will see that this is a reasonable assumption once we perform the computations. Our spaceship is now in motion toward the Moon. But as the spaceship travels, so do the Earth and the Moon move about their COM, and our spacecraft is influenced by their gravitational fields such that

$$\underline{F} = m\underline{a} = -Gm\left[\frac{m_E}{(r-r_E)^3}(\underline{r}-\underline{r}_E) + \frac{m_M}{(r-r_M)^3}(\underline{r}-\underline{r}_M)\right], \quad (11.31)$$

where m is the mass of the spaceship, which neatly cancels from our equations, and $\underline{r}$ is the position vector of the spacecraft. Writing these in component form for the x and y directions we have

$$\ddot{x} = -G\left[m_E \frac{x-x_E}{d_E^3} + m_M \frac{x-x_M}{d_m^3}\right] \quad (11.32)$$

and

$$\ddot{y} = -G\left[m_E \frac{y-y_E}{d_E^3} + m_M \frac{y-y_M}{d_m^3}\right], \quad (11.33)$$

where the distance of the spaceship from the center of the Earth is given by

$$d_E^2 = (x-x_E)^2 + (y-y_E)^2, \quad (11.34)$$

and the distance of the spaceship from the center of the Moon is given by

$$d_M^2 = (x-x_M)^2 + (y-y_M)^2. \quad (11.35)$$

The x and y components of the Earth and the Moon distances from the COM are given by

$$x_E = r_E \cos(\varphi_E),\ \ y_E = r_E \sin(\varphi_E) \quad (11.36)$$

and

$$x_M = r_M \cos(\varphi_M),\ \ y_M = r_M \sin(\varphi_M). \quad (11.37)$$

As a task to the reader: find out the required physical constants you will need to compute the spacecraft's trajectory for different θ

and v. This list consists of the mean center-to-center distance of the Earth to the Moon (d); the mass of the Earth (m_E); the mass of the Moon (m_M); and the *sidereal* Earth–Moon orbital period (T). We already know G from previous sections in this chapter. Is there anything else we should know? The position vector of the spaceship is computed as the distance from the *center* of the Earth and the Moon, and these bodies are certainly not point masses.

Once we have discovered the necessary physical constants, we are ready to compute. Or are we? Remember that we need computer-friendly units such that we are not dealing with numbers that have large variations in their exponents. The strategy to employ here is as before with Halley's comet; to use the physical constants you have found as the units of measure. For instance, we would use the Earth–Moon distance as the unit of length, the sidereal orbital period as the unit of time, and the mass of the Earth as the unit of mass.

Once you have a program written to find the trajectory of the spaceship you should check there are no bugs in your code, such as incorrect entry of a physical constant, or a mistake in the change of units, and so on. To do this set the mass of the moon to zero and check that you get a stable, circular orbit of the spaceship about the Earth when you set the necessary velocity; you will have to use a variant of Equation (11.29) to work out the velocity required. If you get a circular orbit, then we are ready to attempt to make that trip to the moon.

To monitor the progress of the spacecraft we should store the distance to the center of the moon for evaluation, and perhaps terminating the program once we are at or within the radius of the Moon. Obviously, this means we have likely collided with the Moon but landing on the moon is another problem to solve. An exercise for you to do to find values for θ and v that will get the spaceship to the moon.

For more animated applications of ODE solvers to BIG physics, you should have a lookout for the computer games "Universe Sandbox," or for a more comical bent "Kerbal Space Program."

11.7 TO THE INFINITESIMAL AND BELOW

As discussed in Chapter 4, how to obtain the solutions of the Schrodinger Equation as applied to the infinite square well and the finite square well. In the infinite square well case, we found the solutions analytically, whereas for the finite square well we had to rely on root finding to provide the energy eigenvalues. With our Runge–Kutta–Fehlberg ODE solver, we should be able to tackle any arbitrarily defined electrical potential function with ease.

As a starting point, we can try to emulate the results we obtained for the finite square well using root finding with our adaptive ODE solver. Rather than starting entirely from scratch let us use the energies found from the root search applied to the functions

$$f(E) = \beta \cos(\alpha a) - \alpha \sin(\alpha a) = 0, \tag{11.38}$$

for the even parity states and

$$f(E) = \alpha \cos(\alpha a) + \beta \sin(\alpha a) = 0, \tag{11.39}$$

for the odd parity states and plug those into our differential equation. Here we are assuming we do not know the form of the solution of the wavefunction, instead, we are relying on our integrator to provide us with the answer. As such we need to provide our integrator with a starting point. We could use the middle of the well where we know from experience that even parity states have $\psi(x) \neq 0$ and $\psi'(x) = 0$, and odd parity states have those relations reversed. We would then integrate from the middle of the well to the left, and then integrate from the middle of the well to the right to provide us with the full solution. However, this relies on the knowledge of the behavior of the wavefunctions in the well, which in general we do not know, and in fact, is why we are using the solver in the first place. We need a more general starting location.

We know that for any (arbitrary) potential the wavefunction vanishes to zero as we go deeper into a classically forbidden zone. Let us choose a starting location, x_0 that is deep into the barrier, left of the well, and integrate to the symmetrical position on the right of the well. At the starting location, we can set

$$\psi(x_0) = 0. \tag{11.40}$$

It is also true that the derivatives of the wavefunction vanish the deeper we penetrate the barrier. However, if we set $\psi'(x_0) = 0$, we would obtain a solution that implied the wavefunction was zero everywhere; we would have no particle in our system. Therefore, we set $\psi'(x_0)$ to some small, positive value; positive because we know that the probability of finding the particle in the barrier increases as we approach the boundary with the well. So how do we choose the magnitude of the starting differential? The answer is that the size really does not matter from a qualitative point of view. All the size of the differential at the starting point does is *scale* the numerical solution of the wavefunction. If we chose $\psi'(x_0) = \delta$ and performed the integration, then changed the value of $\psi'(x_0)$ to 5δ, say, then our wavefunction from this integration would simply be five times that of the previous integration. The physically significant scale factor is the one that normalizes the probability function such that

$$\int_{-\infty}^{\infty} \psi^*(x)\psi(x)\,dx = 1 \tag{11.41}$$

but as we are only interested in the qualitative results for this discussion, that is another problem to solve elsewhere.

On a practical note, try not to start the integration too deep into the barrier. You will find that if you do you then, even with values of $\psi'(x_0)$ on the order of the machine precision, our adaptive step integrator *will not* be able to cope with the change in nature of the differential equation as we cross the boundary between the barrier and the well. Not unless we relax the error tolerance significantly, and this would then give us doubts about the validity of our numerical results. For a 10Å well, centered at the origin we found that a starting point of $x_0 = -8$ Å was about as deep as we could go without any difficulty (using `double` variables); here we set the derivative equal to the error tolerance we used for the root search and integrator, specifically 10^{-8}.

Figure 11.6(a) shows the results of the integration as discussed above for the ground state function. We see that the regions I and II seem to have been computed correctly showing the same qualitative

result as the root-finding function. But what is going on in region III? We see that the wavefunction initially behaves as expected as we enter the barrier but as we go deeper it blows up exponentially. This anomaly can also be seen in the higher energy states. As we are using a proven adaptive step solver with a low degree of tolerance (10^{-8}) then we can rule out numerical error as the cause. Also, as the wavefunction behaved as expected in the other two regions we can rule out programming error with some confidence (though a check might be prudent in general). To gain further insight into the cause of this unphysical behavior of the wavefunction let us perform the same integration in the reverse direction. That is, starting at $x_0 = 8\,\text{Å}$ and integrating backward to $x = -8\,\text{Å}$. Here $\psi'(x_0)$ will now be some small, *negative* value. The results of this reverse integration are shown in Figure 11.6(b). Here we see the same problem but now in region I, not region III. The fact that the backward integration looks like a reflection in the vertical axis of the forward integration lends credibility to our supposition that the programming is correct. Clearly the direction of the integration affects the numerical solution.

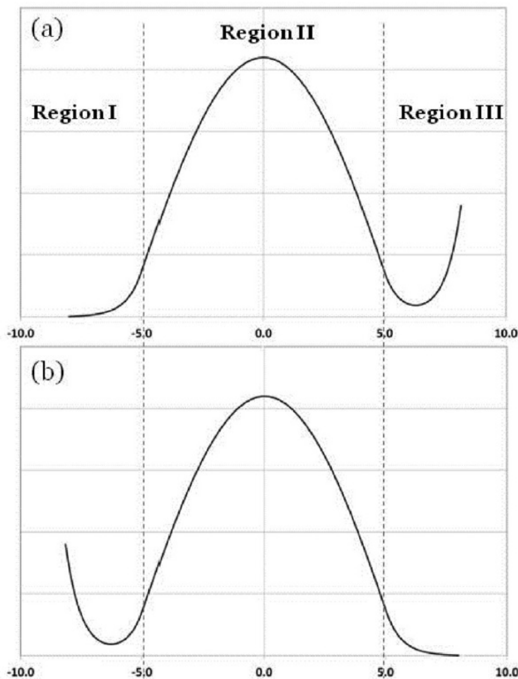

FIGURE 11.6: Results of the integrator: (a) integrating left to right; (b) integrating right to left.

To answer that question, we look toward the *general* solution for Schrodinger's equation in the barrier regions, specifically in region I we have

$$\psi_I(x) = Ce^{\beta x} + De^{-\beta x}. \tag{11.42}$$

Thus, mathematically speaking, within the barrier the wavefunction consists of two exponential terms; one grows while the other decays. Recall that we set D to zero using a physical argument based on the results of observation and experiment. Thus, we assumed the wave function had the form $Ce^{\beta x}$ only. However, mathematical equations tend to be oblivious to our physical reasoning. What does this mean for our integration? Let us consider the first integration. Here we progress the solution forward from a *negative value* of x such that in region I the *magnitude* of x decreases. This means that our desired wavefunction term $Ce^{\beta x}$ is the *growth* term, whereas the *unwanted* term $De^{-\beta x}$ is the *decay* term; take your time to verify this. Hence, we are integrating into the direction where the unwanted term decays.

In region III of the forward integration, we start at a *positive* value of x, namely the well border, and progress from there such that the *magnitude* of x *increases*. From symmetry arguments, the general solution to Schrodinger's equation for region III is the same as for region I (with a coefficient sign reversal for odd parity wavefunctions). In this case, the desired solution is the $De^{-\beta x}$ term and the unwanted term is the $Ce^{\beta x}$. In other words, our desired solution decays while the unwanted term grows. Hence, we have identified the cause of our problem. We can apply similar arguments to the backward integration and find that while the unwanted term decays in region III, it grows in region I. The reason why the unwanted terms exist in the first place is because of the slight imprecision in the calculation of the energy eigenvalue. Even though we have calculated it using a root searching algorithm to a precision of at least the order of 10^{-8} it is *not* an exact value, and the coefficient of the unwanted term is not exactly zero, but some minute yet finite number. However, the exponential term grows rapidly with x; exponentially in fact! Eventually, there will come a point where this small coefficient multiplied by the exponential growth factor will become the dominant term and cause our solution to blow up where we would expect it to decay.

The remedy then is to always integrate from a classically forbidden region toward a classically allowed region. In this way, any unwanted solution will decay, while the desired solution grows. For any symmetrical potential, this is particularly easy as we can simply integrate from the left of the well to the middle of the well for various energies. We find the eigenvalue by finding the energy E that satisfies

$$\psi'(x = 0, E) = 0, \tag{11.43}$$

for even parity wavefunctions, and

$$\psi(x = 0, E) = 0, \tag{11.44}$$

for odd parity wavefunctions. The rest of the wavefunction will just be the mirror image of that calculated reflected in the vertical axis at the middle of the well; see Figure 11.6.

A slightly better way of searching for the eigenvalue, in that it removes the ambiguity in the choice of $\psi'(x_0)$, is to search for the energy that satisfies the logarithmic derivative being zero, that is

$$\left.\frac{\psi'(x,E)}{\psi(x,E)}\right|_{x=0} = 0, \tag{11.45}$$

for even parity wavefunctions, and the inverse of this for odd parity wave functions.

Be aware that although useful for instruction, symmetric potentials rarely arise in real quantum mechanical systems, and as such will not contain pure even and odd parity wavefunctions. However, the general strategy of solution still applies; choose a matching point in the classically allowed region, that is, the "well"; integrate up to this point from opposite sides in the classically forbidden regions, and compare the logarithmic derivative at the matching point. Mathematically, we find the energy eigenvalues that satisfy

$$\left.\frac{\psi'_L(x,E)}{\psi_L(x,E)}\right|_{x=x_m} = \left.\frac{\psi'_R(x,E)}{\psi_R(x,E)}\right|_{x=x_m}, \tag{11.46}$$

where ψ_L is the numerical solution for the wavefunction integrated from the left to the matching point x_m, and ψ_R is the numerical solu-

tion for the wavefunction integrated from the right to x_m. As the notion of pure even and odd states does not apply Equation (11.46) holds for *any* energy eigenvalue. Note that it may happen that at the matching point we choose the wavefunction tends to zero and we end up with a singularity in the calculation of the logarithmic derivative. Is there any location within the domain of interest where we know a wavefunction must have some finite value, regardless of the shape of the well?

The source file *numerov_ocv.cpp* contains code that attempts to implement the general strategy we have just discussed. In this case, we are using the `Numerov` class that implements the eponymous algorithm to solve Schrodinger's equation for any arbitrarily defined potential. The derivation of the Numerov algorithm is left as an exercise for the reader; there are many references in the literature and online. The matching point in this program is determined from the potential and is defined as the point where the energy of the particle crossed the potential barrier. Know that you can swap the `Numerov` class for the `RKF45` class but setting the RKF45 class to produce results at a specified target (so we hit the matching point).

So, there you have it, we have robust ODE solvers that can tackle problems on the scale of the universe to the scale of the quantum to a user-defined precision.

EXERCISES

11.1. Investigate the Van der Pol oscillator further through variation of the damping parameter μ and the initial conditions. Can the Van der Pol oscillator ever become chaotic?

11.2. Establish a relationship between the driving frequency and the period of oscillations for a periodic, that is, not chaotic, driven pendulum for set values of q and b. Is there a more general relationship as we vary q and b, but still within the non-chaotic region?

11.3. Duffing's oscillator is described by the differential equation

$$\ddot{x} + \alpha\dot{x} + \beta x^3 + \gamma x = \delta\cos(\omega t)$$

where α through δ are constants, and ω is the frequency of the driving force. Investigate the motion of the oscillator for different relative values of these constants.

11.4. Find the initial values for θ and v for our spaceship to loop the Moon and return to Earth.

11.5. Model the motion of a rocket that is launched from the surface of the Earth and establishes a stable orbit at 500 km above the Earth's surface. To produce the thrust the rocket burns fuel and propels the gases out of its rear end, such that the mass of the rocket changes with time. Additionally, the density of the atmosphere is a function of height above the Earth's surface and this should be considered.

11.6. Write a program to simulate a journey to our next planet outward in the solar system, Mars. How precise should we make our calculations?

11.7. In realistic, solid-state quantum well devices the potential walls of the well are better modeled by a graduated slope rather than an abrupt "cliff edge." Investigate the effect of the steepness of the sloped walls on the bound energy states of the well.

11.8. Investigate the bound states of the potential centered on the origin, defined by

$$V(x) = \begin{cases} V_1, & |x| > b \\ V_2, & a < |x| \le b \\ 0, & |x| \le a \end{cases}$$

where $V_2 > V_1 > 0$ and $|b| > |a|$. Comment on the states with energy eigenvalues greater than V_1 but less than V_2. (Tip: It would be extremely useful to sketch this potential before trying to solve it computationally).

11.9. Asymmetrical anharmonic potential in one dimension can be written as fourth-ordered polynomial such that

$$V(x) = \alpha x^4 + \beta x^2.$$

Find the first four energy eigenvalues for $\alpha = 0.5$ and $\beta = 1.0$ to at least 6 significant figures of accuracy. Study the effect of different values of α and β on these energy eigenvalues. You should plot $V(x)$ with the wavefunctions computed, offset by the corresponding energy eigenvalue. Investigate the effect on the wavefunctions as we add odd powers of x to the potential. Note that when $\alpha = 0$ we have a *harmonic* oscillator.

CHAPTER 12

HIGH-PERFORMANCE COMPUTING

In the other chapters of this book, we have only looked at getting algorithms to work as computer code. In this chapter, we look at getting algorithms to work quickly or efficiently—these are not necessarily the same thing. To that end, this chapter explores two methods to achieve high-performance computing. Firstly, loop unrolling and blocking that attempts to make efficient use of the computer's memory architecture and, secondly, parallelism that attempts to utilize the total potential computing power of multiple-core processors.

This chapter will discuss some of the fundamental ideas of memory structure and memory access but is by no means exhaustive or comprehensive. As with all things in computing, there are levels of abstraction, the more levels you peel away the more technical (and usually complex) the ideas get. At the core of high-performance computing is understanding the "mechanics" of the underlying hardware. This chapter will show examples of both the OpenMP extension to C++ and the modern C++ API for threaded applications. You will find the code for this chapter in its own subdirectory of the "progs" directory names "high_perf_progs."

It is worth pointing out here that any third-party library you use, that has been professionally developed, will highly likely have taken the ideas discussed here on board and have developed highly optimized methods for a wide variety of problems. In trying to develop your own "high performance" methods you would very much be

reinventing the wheel, only yours would likely be cuboid in shape and made of concrete. Still, these ideas are good to know where perhaps third-party libraries are not available, or you are working with an unusual environment that requires hand-crafted optimizations.

12.1 INDEXING AND BLOCKING

In this section, we will discuss in more detail the underlying structure of your computer and how we as programmers can make the best use of that structure. Note that the majority of what we will discuss can be handled automatically by most modern compilers. However, it is always prudent to be aware of how a computer is put together and how it operates to ensure the best possible performance the hardware can manage, or at least know how not to make fundamental mistakes.

12.1.1 Heap and Stack

The memory that a program uses is typically divided into a few different areas, called segments, that exist in RAM when the program is executed. The code segment is where the compiled program, or binary, sits in memory. The data segment is where explicitly initialized global and static variables are stored. The heap is the memory segment where dynamically allocated variables are stored. And the call stack, where function parameters, local variables, and other function-related information are stored. Here, we focus on the heap and the call stack where most of the interesting mechanics occur during the operation of a program.

The heap segment (or the "free store") keeps track of memory used for dynamic memory allocation. For example, in C++, when you use the `new` operator to allocate memory, this memory is allocated in the process's heap segment. Similarly, Fortran has the variable attribute `allocatable` that will store the variable on the heap. Regardless of the programming language, in general, you do not have to worry about how this memory is allocated to the process. However, it is worth knowing that sequential memory requests in source code may not result in consecutive addresses being allocated in computer memory. In both C++ and Fortran arrays allocated on

the heap are stored in an unbroken (contiguous) memory. When a dynamically allocated variable is deleted, the memory is returned to the heap ready to be reallocated on a future request. Whenever a process terminates, either normally or abnormally, any memory it had allocated on the heap is cleaned up by the operating system.

The heap has advantages and disadvantages:

- Allocating memory on the heap is slow compared with the stack.
- Allocated memory persists until it is specifically deallocated or the application ends (a "memory leak" is heap-allocated memory that has not been properly deallocated during program operation).
- Accessing a variable that has been dynamically allocated on the heap is generally slower than accessing a variable directly on the stack.
- Arbitrarily large data structures can be allocated on the heap (up to the system limits).
- The heap can fragment, that is pockets of free memory can occur between allocated memory blocks as a program runs. A program may run out of memory even though the combined size of the free "fragments" could accommodate the memory allocation request.

The call stack, or more simply the stack, is a special region of your computer's memory that stores temporary variables created by each function (subroutine) call. The stack is a "LIFO" (last in, first out) data structure, that is managed by the operating system.

Every time a function is called during the program operation, it is "pushed" onto the stack, we refer to this as a stack frame. In C/C++ a stack frame consists of:

- the memory address of the instruction following the function call referred to as the return address;
- the function arguments;
- the memory required to store any local variables (this is determined at compile time).

Stack frames will contain similar data for other programming languages. Once a function exits the stack frame is "popped" off the stack, freeing the memory used for arguments and local variables, and the return address is used to resume execution after the function call. Thus, as a program executes the stack grows and shrinks as functions are called then return. Typically, we picture the stack as growing away from memory address "zero" in a downward fashion, but in some systems, the stack is more accurately seen as growing toward memory address "zero" in an upward fashion.

When function calls are nested, a function call contains a function call that contains another function call and so on, each new call will allocate the required memory for a corresponding stack frame and we say the frames are "stacked". The execution of those functions remains suspended until the very last function returns its value. At that point, the frames will "unstack" or unwind in the correct order. This makes it simple to keep track of the stack, as freeing a block from the stack is nothing more than adjusting a value contained in a CPU register, sometimes referred to as the stack pointer.

As the stack is a limited block of memory, you can cause a stack overflow by calling too many nested functions, for example, many recursive function calls, or allocating too much space for local variables. Often the memory area used for the stack is set up in such a way that writing beyond the given extent of the stack will trigger a trap or exception in the CPU. This exceptional condition can then be caught by the operating system which terminates the process and displays an error message to the user. The size of the stack is operating system dependent.

In a multi-threaded environment, each thread will have its own independent stack, but they will typically share the heap.

To summarize the stack:

- the stack grows and shrinks as functions are called then return
- local variables are allocated and freed automatically
- the stack has size limits
- stack variables only exist while the function that created them, is running

Because the stack is relatively small, it is generally not a good idea to do anything that eats up lots of stack space. This includes passing by value or creating local variables of large arrays or other memory-intensive structures.

12.1.2 Computer Memory

As discussed in the introductory chapter, each level of computer memory can be thought of as a huge filing cabinet, each draw representing a memory address in which we can store one word (typically, a word is 4 bytes, or 32 bits long). The memory can only be accessed one draw or address at a time and the current address is referenced by the system's address pointer, generally referred to as the program counter. To change from one address to another the program counter can either step from one address to the next or can be instructed to jump. Think of it like changing the channel on your TV using the channel + and – buttons shifting to adjacent channels or inputting the number directly and jumping to that channel. Computer memory is commonly referred to as being contiguous; the address locations share a common border.

We also discussed in Chapter 1 that computer memory is split into a hierarchical system whereby the smallest memory, the CPU cache, is the fastest, and the largest memory, the storage device, is the slowest. RAM exists between CPU cache levels and storage. When operating, the CPU will ask for the variables that require work. If the variables are not already in cache a signal is sent to fetch them from RAM. If the variables are not in RAM, then a signal is sent to fetch them from the storage device. The variables are then read and copied from storage into RAM, then read and copied in the CPU cache levels. Once in cache the CPU performs the required operation and writes the result and/or changes to the variables back to RAM, which in turn writes those changes back to the storage device. Each one of these stages requires at least one clock cycle to complete, and accessing storage may require several hundred. Note that those variables will now persist in RAM (the stack and heap) and the CPU cache levels until they are flushed by the system. This persistence allows the CPU quicker access to those variables should they be required again soon.

Typically, programs need to work on arrays and will consecutively work on those arrays. For example, let us say we have two vectors, a, and b of length N that require addition. Code is written as a loop that iteratively steps through the arrays one element at a time, performing the addition. It would be inefficient if the fetch instruction only brought up the two variables from storage that required immediate addition; the fetch instruction would have to be issued N times, that is, the storage device must be accessed N times. Far more efficient would be to bring up a block of variables at a time, the length of which we referred to as the cache line, temporally storing them to RAM then the cache, and if N is sufficiently large, filling the cache levels. Subsequent array variables can now be accessed quickly with fewer calls made to storage. The number of fetch instructions sent to access the storage device is now approximately the ratio of the array length to the cache line. You may therefore think that the best cache line would be the length of the array, however, it is limited by the amount of data that can be passed via the memory buses, typically gold alloy wires, that connect the different memory components. Generally, the cache line is some fraction of the size of the level 1 cache and will be some integer multiple of eight. For clarification, the cache line is measured in bytes rather than the actual number of variables contained in the line as of course, the variables could be of different types. For example, if the cache line were thirty-two bytes long this would be enough to store eight, four-byte words (single precision) or four, eight-byte words (double precision).

This idea of blocks of memory affecting the performance of your computer is one you may have come across before if you have ever defragmented your hard drive to make it run quicker. Programs store variables and data on the HDD in blocks of memory that are accessed when that program is run. During the lifetime of your computer, those blocks become broken and jumbled up, in technical parlance fragmented. This has a detrimental effect on your computer because the CPU must issue more fetch commands to receive the correct pieces of memory. By defragmenting the HDD those blocks reform into unbroken pieces of memory, which helps improve the performance of your computer. It also has the secondary effect of freeing up some storage space. If you were born after the year 2000 then you likely have no idea about defragmenting a

hard disc drive. SSDs work differently from disc drives and do not require defragmenting.

Of course, while the CPU is busy performing the required operations on the variables now stored in level 1 cache the other components do not have to be idle. Other blocks of memory can be fetched up to fill level 2 cache, and once full, begin to fill RAM. As the blocks of memory are finished with, they are written back down the memory hierarchy, flushed from the cache, and fresh ones moved into level 1 cache to be worked on. This process is known as pipelining and is continuously working away in the background during the operation of your computer, making it incredibly efficient at number crunching. Normally this efficiency is implicit; the computer just does its thing. However, a computer is only as clever as the program telling it what to do. Sometimes the requests we make of the computer are, to put it technically.

12.1.3 Loopy Indexing

Imagine we are adding two exceptionally large matrices and storing the result in a third matrix of the same size. First, how do the elements of a matrix get stored in computer memory? A matrix is a two-dimensional array, computer memory is, in essence, a one-dimensional array. We must resize the matrix into one dimension. There are two solutions, one is to store the matrix in a column-major format that is, the order they appear in the columns, the second is to store the matrix in a row-major format that is, the order they appear in the rows. In either case, an n-by-m matrix essentially becomes an $(n \times m)$-by-one vector in computer memory.

For arguments, let us suppose we store a two-dimensional matrix in column-major format, and the elements are stored in an unbroken block of memory (this is how Fortran stores matrices). How then do we write code to efficiently access and sum the elements as they are stored in computer memory? In this case, it is most efficient to access and sum elements as they are laid out in memory, that is keeping the column index constant as we increment the row index. In this way we take advantage of the values fetched in the cache line from memory and the CPU can spend most of its time doing the useful work of summing elements and storing them to the relevant location in the resultant matrix. Now imagine we reverse the order

of loops; row index constant while the column index increments. This change does two things. Firstly, the program counter now must jump over the total number of rows of the matrix to find the next elements to add. Secondly, and most detrimentally, the remaining variables that are fetched in the current cache line are immediately redundant; they are flushed from cache memory before becoming useful as the matrix is larger than the cache size. In essence, we are forcing the CPU to cache only one element at a time for each matrix involved in the sum.

The source file *loop_index_order.cpp* highlights the importance of knowing how the underlying data structure is arranged in computer memory. Our representation of a two-dimensional matrix in code is a one-dimensional array of length equal to the number of elements in the matrix, and we have assumed a row-major format. We create two 1000-by-1000 matrices, that is, one million elements each, then sum them together and store the result in a third matrix of the same size. The sum is performed using nested loops, row index, and column index, and we show that the order of these loops is important by timing their execution. Notice that this is somewhat contrived as we could have simply looped over a single index (the index computation is the same for each matrix involved) but that would have missed the point of this section.

On a practical note, it is always a good idea to repeat timing measurements: firstly, the system clock may not have a very high resolution such that very fast operations may not be timed at all accurately; secondly, the machine could have been doing something else while you were running your test thus skewing the results; third you can attempt to measure each individual run (bearing in mind the first point) thus providing you with the data to compute statistical analytics (mean, standard deviation, etc.). With an eye on the second point, it is usually a good idea to minimize the number of other programs running on your system such that they do not interfere with the timing results. Also, beware of compiler optimizations. Usually, compiler optimizations are desired but when trying to time programs they can be a nuisance. This is especially true when performing repeat loops that, in essence, do nothing useful from the compiler's point-of-view, and may get removed in the executable

code (for optimization levels greater than zero). The usual strategy is to make the code do some sort of small modification to the data within the repeat loop, and whose execution time is insignificant when compared to the operation you are trying to time.

Another thing to bear in mind when producing timing results for a test program is that not all computers are equal. Comparing algorithms by their execution time on the same machine is okay but to do so against a different machine is unfair. Instead, we should quote performance as the number of floating-point operations per second. For a single thread of execution, this can be approximated by the execution time multiplied by the CPU frequency. This does not account for differences in memory architecture: cache levels; RAM; storage device(s), so we should still be cautious.

If you are building programs with high performance in mind, then these kinds of memory structure considerations should be paramount. You should also consider how to keep total memory usage to a minimum. For instance, in the matrix addition example above do we need to keep the source matrices in memory when all we want is the resultant addition?

12.1.4 Blocking

Matrix multiplication is a little a more involved than matrix addition. Matrix multiplication is defined elementwise by:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \tag{12.1}$$

for each i and j, where m represents the inner dimension of the matrix product. For example, matrix A with dimensions n-by-m, matrix B with dimensions m-by-p, resulting in matrix C with dimensions n-by-p. Note that there are three indices namely i, j and k. Clearly, for large matrices (where the three matrices combined are larger than half the cache size) memory access will become a bottleneck for matrix multiplications. It can be shown that for matrices of the dimensions shown in Equation 12.1 the number of floating-point operations required to multiply them is $2nmp - np$. Thus, for large matrices, the number of floating-point operations increases as

$\mathcal{O}\left(N^3\right)$, where N is the largest dimension. This is for dense matrices, that is where most of the elements are non-zero. For sparse matrices where most of the elements are zero, the matrix multiplication algorithm can be modified to remove unnecessary multiplications by zero. Sparse matrices are beyond the scope of the discussion in this section, but you should be aware that they occur often in numerical analysis and require special treatment. We will continue the rest of this discussion assuming we are dealing with dense matrices.

The first step in writing a more memory-efficient algorithm for matrix multiplication is to realize we can separate the matrices into sub-matrices of block rows or block columns. We then treat the whole matrix multiplication as performing multiplications using these strips. The strip width is the number of rows or columns contained within the strip and can be set so that the total amount of memory consumed when using the strips is equivalent to the cache size. However, there is a flaw in our strategy here. As the size of the matrix increases the width of our strips necessarily reduces to maintain a cache block size. Eventually, a matrix of sufficient size will make the strip width equal to one and we have lost any performance improvement our strategy might have afforded us. Therefore, we need to generate a sub-matrix whose size is independent of the size of the total matrix, and equivalent to or less than the cache size. Taking the lead from the strip idea, whereby we divided the matrix along one of its dimensions, we now divide along the second dimension thus forming blocks. The easiest way of thinking about performing matrix multiplications with blocks is to treat the blocks as if they were elements. The row by column process still applies. Figure 12.1 illustrates this block multiplication process. Here the multiplication operations in the brackets can be done in any order, which is useful to know for parallel programming which we will discuss shortly. Using this process, we can bring up two blocks into cache, one each from matrix A and B, perform a normal matrix multiplication, and store the result in the corresponding block of matrix C. The notation we will use for the block, sub-matrices will be A_{nm} where n is the block row index and m is the block column index.

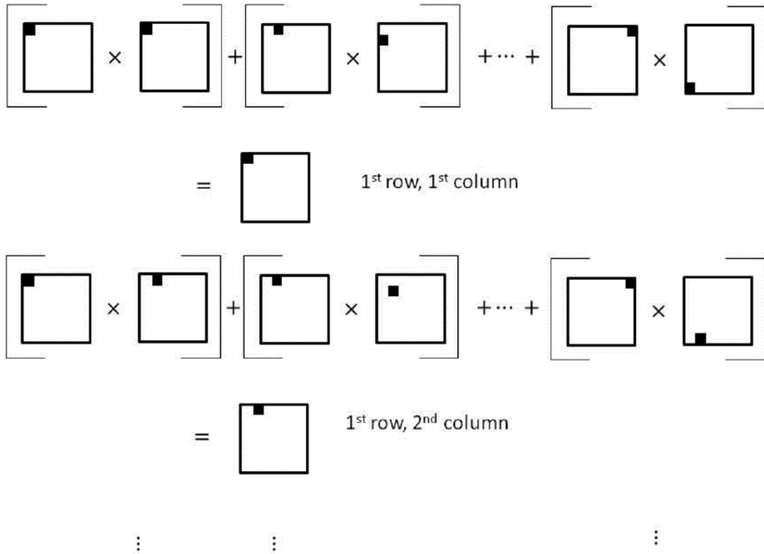

FIGURE 12.1: Block matrix multiplication. The bracketed terms are matrix multiplications in of themselves.

The idea with memory-efficient programs is to ensure that the relevant variables, that is the variables we wish to work on, persist in the higher levels of memory (cache and RAM) until they have no further use and can be flushed. Looking more closely at the multiplication process we note that the first block of matrix A is only ever involved with the first block row of matrix B. In fact, only the first block *column* of A is involved with the first block *row* of matrix B. It is easy to extend this to the k th block column of A and the k th block row of B. This is a consequence of the inner product nature of matrix multiplication. Elementwise we must have $c_{ij} = a_{ik}b_{kj}$ thus similarly for our sub-matrix blocks, we must have $C_{ij} = A_{ik}B_{kj}$. Here we are using the Einstein notation that repeated indices are summed over. Thus, we might proceed by keeping A_{11} in cache, while we iterate through the first block row of B. Then move to A_{21} and repeat the iteration through the first block row of B. We continue in this fashion until we have completed the first block column of A. In this way, so long as we have chosen the correct block size, the first block row of B will be kept in level 2 cache while the calculations are performed. Then we move to the second block column of A and the second block row of B. We continue with this pattern until the whole matrix

has been covered. So long as we keep the bookkeeping correct the order of the sub-matrix multiplications is unimportant.

There are other ways to improve the speed of matrix multiplications. Strassen's algorithm is one of them. Strassen's algorithm partitions the matrices into sub-matrices then, with a clever bit of manipulation, reduce the number of sub-matrix multiplications required to get the correct result by one. The sub-matrix multiplications are swapped for matrix additions and subtractions. Applying this idea recursively to the sub-matrices we can significantly reduce the required number of operations for matrix multiplication (for large matrices). This recursion can continue until the sub-matrices degenerate into numbers, however, in practice, it continues until the sub-matrices are of such size that the naïve matrix multiplication algorithm becomes more efficient than continuing the recursion. We already know that the runtime associated with the naïve approach to matrix multiplication is proportional to $\mathcal{O}\left(N^3\right)$, where N is the size of the matrix (assumed square). Strassen's algorithm provides a runtime that is proportional to $\mathcal{O}\left(N^{2.8074}\right)$. It is worth noting that Strassen's algorithm is less numerically robust than the naïve approach as it involves the subtraction of sub-matrices to compute. If the corresponding elements of those sub-matrices are sufficiently close in value, then this will lead to unit round-off errors in the result.

12.1.5 Loop Unrolling

Another way of squeezing performance out of a computer program is to unroll incremental loops. That is instead of incrementing the loop index by one on each iteration we increment it by a larger integer value and adjust the contents of the loop appropriately. The increment of the loop is referred to as the stride. For instance, if we were summing the elements of a vector, then normally the stride is one and we would write the loop as:

```
for (int i = 0; i < v.size(); ++i) {
        sum += v[i];
}
```

However, if we change the stride to two then we write:

```
for (int i = 0; i < v.size(); i += 2) {
       sum += v[i];
       sum += v[i + 1];
}
```

Here we increment the array index `i` by two. This has the effect of reducing the number of instructions spent controlling the loop, such as pointer arithmetic and testing for the end of the loop. In this way, more time is spent on the calculations we require. It has the additional benefit of hiding inherent latencies, especially the delay in reading variables from memory. Notice that should the size of the vector not be multiple on the stride length then some additional code is required to deal with the remaining elements.

Manual loop unrolling is only advised if we want to squeeze every bit of performance out of a particular program. As you can imagine the process becomes rapidly tedious as we try to extend the unrolling and has diminishing returns in terms of runtime. Modern compilers are designed to optimize the binary code produced from your source code and can apply loop unrolling automatically.

12.2 PARALLEL PROGRAMMING

Multiple-core machines are now ubiquitous. They offer a means of performing computations in parallel rather than in sequence. For some problems making the computations performed in parallel is rather straightforward, performing a direct numerical quadrature, say, or summing the elements of an array. For other problems-making algorithms, the parallel is not quite such a simple task, for instance, matrix factorizations or some iterative methods such as Successive Over Relaxation. The difficulties tend to arise from inter-dependencies between the different subroutines used to solve the problem or the elements of the array themselves.

No program can run faster than the longest chain of dependent calculations, known in network theory as the critical path. As an

analogy consider making a cup of tea. If you want to do this efficiently that is, in the quickest time possible, then you would fill and turn on the kettle first. Then as the water boils you would find a mug, put a tea bag in it, fetch the milk, and probably still have time before the water finishes boiling. Once the water has boiled you pour it over the tea bag in the mug, allow it to brew, extract the bag, and add the milk. Note that the critical path here is filling the kettle, waiting for it to boil, adding the hot water, allowing it to brew, extracting the tea bag, and adding the milk. Each of these tasks is dependent on the last and therefore cannot be done in parallel. Strictly speaking, this is task parallelism rather than a true analogy of multiple-core parallelism; which would be several people making a single cup of tea at the same time.

This section does not give an in-depth study of parallel computing but should provide the reader with practical instruction, and hopefully draw your interest for further study in the topic.

12.2.1 Many (Hello) Worlds

The OpenMP (OMP) directives do not constitute a new language rather an extension to the C++ we already know. However, just like learning a new language, we should still start with a relatively basic program to get used to the syntax. The file *omp_helloworld.cpp* contains the program code that will output to screen the text hello world and from which processor (thread) the message is coming from. Assuming you have a process capable of producing N logical threads then you should receive N hello-world messages. Notice that the program uses the C function "printf" to output the messages. Instead, try using the C++ output stream "cout" to achieve the same result. What happens and why?

If you read the Makefile for the high-performance programs you will notice that for your program to use the OMP pre-processor directives you must supply the option "-fopenmp." This instructs the compiler to parse all the "#pragma" directives as OMP instructions. If you wish to use any actual OMP functions then you must include the relevant header file, *libomp.h*, and link in the corresponding library, *omp*. Where these are located is OS-specific but are generally in the usual places.

Once you have successfully compiled and run the program you should have received the appropriate number of hello worlds to your screen, plus the processor identification number the message came from. Note that the processor numbering starts from zero. If you are unsure of the number of processors on your system the omp hello-world program will report the maximum number of logical cores on your system. The number of logical cores may be different from the number of physical cores due to what is called hyper-threading technology on Intel processors. Essentially, hyper-threading allows one physical core to perform two tasks in parallel, up to a point. Thus, an Intel, quad core processor will have a maximum of 8 logical cores and consequently a maximum of eight threads. However, if the operating system is unaware of the hyper-threading technology (such as using Cygwin through Windows) then it will see the threads as individual, physical cores. For instance, if you have an Intel i7 processor that is *quad core*, you may see an eight for both the maximum number of threads available *and* the number of cores on your system. For AMD processors the maximum number of available threads will be equivalent to the number of cores your processor contains.

Returning to our parallel hello-world program let us have a look at the new syntax we have introduced. It is rather straightforward in that we declare a parallel section in our using the `"#pragma omp parallel"` pre-processor directive, that can be scoped using the curly brace delimiters. Note that we need an extra include to get access to the OMP functions but in general this is not necessary.

As an aside, if you think of the computational work being done as a line, or thread, on a piece of paper, as the code enters a parallel region the thread separates, or forks, into several threads equal to the number of processor cores on your machine (here we ignore the hyper-threading of Intel's processors), performing the computations simultaneously. Each separate thread is being worked on by a separate core and cores do not swap threads (unless specifically programmed to do so). After the parallel region, the threads are joined and the work continues on the master thread (or core), which is identified as thread zero.

Now that we have seen how to set up a parallel section using OMP let us apply that to something more mathematical.

12.2.2 Vector Summation

Let us start with the relatively simple task of summing the elements of a vector; this has used within statistics for finding the mean value of a data set, say. On a single-core serial processor, we would loop through the entire array incrementing the index by one and updating the sum. Usually, we call this operation accumulation, indeed the standard template library for C++ has function, std::accumulate, that performs this task. For a parallel architecture, we can split the entire vector into equally sized chunks and perform the summation on each chunk simultaneously. To obtain the sum for the entire vector we would then add up the contributions from each chunk on a single thread. If we define T_s as the time taken to perform the summation on a single core, in a serial manner, then we might expect the time taken for the code to run in parallel to be T_s/P, where P is the number of threads used. This is known as the ideal case; the speedup in program runtime is directly proportional to the number of threads used. In general, the speedup is given by the ratio T_s/T_P, where T_P is the time taken to run the code in parallel on P threads.

For the sake of rigor, the serial time T_S is *not* the same as T_1. That is, T_S is the time taken to perform the code as normally written, whereas T_1 is the time taken to perform the code where we have spawned a parallel region using only one thread. Typically, $T_1 > T_S$ by a small amount as we have an overhead associated with setting up the parallel region. That said sometimes the "serial" code may be threaded by the operating system automatically meaning that our speedup measurements might be skewed if we assume the computations are being carried out on a single core. So long as we are clear about which measurement of the "serial" time we are taking we should not run into problems. For the rest of this chapter, we will take the serial time as T_1. Also, note that an even more rigorous treatment of speedup is to consider the number of floating-point operations per second (flops) rather than runtime as the flops measure is independent of the computer architecture used. For the purposes of our discussion, we will stick with the runtime as a suitable measure of performance.

The file *accumulate_ocv.cpp* contains the source code to perform the vector summation in parallel. In fact, the OMP parallel section resides in *phys_accumulate.h* as we have declared the "accumulate" function as a template function. This means we can apply the "accumulate" function to vectors containing any type that can be summed. Note that the function will return the type used to initialize the "sum" variable. You will also find in that header file an implementation of a parallel "accumulate" function using just the C++ language. It uses several advanced C++ constructs that make the code somewhat difficult to read and certainly less elegant than the OMP version. In essence, the OMP version hides the details of the parallelism in the directive and, consequently, makes it easier to read. However, the entirely C++ version requires no additional, external software to work.

Focusing on the OMP version, we start the directive as we did in our hello-world parallel program, but we specify that it is the following `for` loop that we want to be done in parallel. The *reduction* clause allows the variable sum to be updated as the addition of each thread's local value of `sum` on the master thread. To clarify, each thread *reads* the value of `sum` from the (thread) shared memory, creating a local copy on which to work. *Without* the reduction clause once the work has been completed each thread writes its value of `sum` back to the shared location. This means that the updated value of `sum` would be whichever thread finished last. *With* the reduction clause each thread, once finished, passes its local value of `sum` to the master thread which then combines them as specified by the operator argument in the clause; in this case, it adds them.

Compile and run *accumulate_ocv.cpp*. You may want to modify the code to display the speedup, T_1 / T_P, of the parallel sections rather than the execution time. Did you obtain the speedup you expected? Was it anywhere near the ideal case? On an AMD, quad core processor I found the speedup (T_S / T_4) to be around 3.7. On an Intel, quad core processor using hyper-threading technology running the same code, I found that the best-observed speedup (T_S / T_8) was around 5.1. (Remember that both these processors have four physical cores thus hyper-threading does offer a performance benefit, but not as much as actually having eight physical cores.)

In its current, default state the OMP directive will divide the iterations among the threads equally, and where the size of the loop is not exactly divisible by the number of threads the remainder gets evenly distributed. This is called static assignment. We can change this assignment through the use of a scheduling clause. The syntax for this clause is relatively simple and all we do is add *schedule (type, chunk)* after the OMP `for` the directive. Here type is either *static*, *dynamic*, *guided*, or *auto*, and *chunk* is an integer value that has slightly different meanings depending on the type used.

We have already seen that *static* assignment allocates all the iterations to each thread before they execute at runtime, with the assignment being divided equally among the threads by default. We can change this behavior by specifying a chunk size. For instance, if we keep the static schedule type and set the chunk size to one then each thread gets one iteration to complete before being assigned its next iteration; the stride pattern is the number of threads. In *dynamic* assignment only some of the iterations are assigned to threads before execution of the loop; the number being set by chunk. Once a particular thread finishes its allotted iteration(s), it requests another chunk of iterations from those that remain. If you were to print out which thread did which iterations, you would find that the assignment would be (somewhat) random from execution to execution. *Guided* allocates a large portion of iterations to each thread dynamically, as above, but then decreases the portion size after each successive allocation until it reaches a minimum size specified by chunk. *Auto* allows the compiler to decide the best type and chunk to use at runtime.

The "best" scheduling strategy is to ensure none of the threads are idle during the parallel sections and that we make the most effective use of our parallel machine. Essentially, we are ensuring a uniform distribution of work across all threads. Add a schedule clause to the OMP directive for the accumulated function and play around with the scheduling clause to see if it affects the performance of the program by any significant amount.

12.2.3 Overheads: Amdahl Versus Gustafson

Generally, we find that the improvement in performance due to spawning a parallel region of code is limited by overheads.

Overheads arise from several places not least the necessarily serial portions of the code, and the communication (or message passing) between threads. For instance, in our vector summation (accumulation) example the reduction at the end of the procedure to obtain the total for the entire vector must be done in serial on the master thread. For all parallel regions of code there is one unavoidable overhead; the time it takes to fork (create) and join (destroy) parallel threads.

The point of increasing the number of threads used in a parallel region of code is to improve the performance (runtime or flops) of that region. A typical program has regions that are both parallel and serial. As we increase the number of threads, the parallel regions compute faster but the serial portions are unaffected. Thus, the overall performance of the program is limited by the amount of time it takes to compute the serial regions of code. This is Amdahl's law. We can formulate this law into an equation that states that the speedup in a program is given by

$$S(P) \equiv \frac{T_S}{T_P} = \frac{T_S}{T_S\left[(1-\beta)+\frac{\beta}{P}\right]} = \frac{1}{(1-\beta)+\frac{\beta}{P}} \tag{12.2}$$

where β is the portion or fraction of the code that is or can be made parallel. We can see this formulation makes sense in that with no serial fraction of code ($\beta = 1$) the speedup is simply given by the number of threads used in the parallel region (ideal case). Obvious but worth pointing out that when the code is completely serial, $\beta = 0$, then there is no speedup, $S(P) = 1$. Figure 12.2 plots the consequence of Amdahl's law on the speedup of various programs compared to the number of processors (threads) used. Each line represents a program with different fractions of code that can be made parallel and shown for comparison is the ideal case ($\beta = 1$). These results seem quite pessimistic, where even a code with only a 10% serial portion diverges significantly from the ideal case for a relatively low number of processors. Eventually, for any code less than ideal adding more processors fails to further improve the performance. The situation is worse as we have not considered the overhead due to communication between processors and adding more processors may decrease the speedup after a certain point.

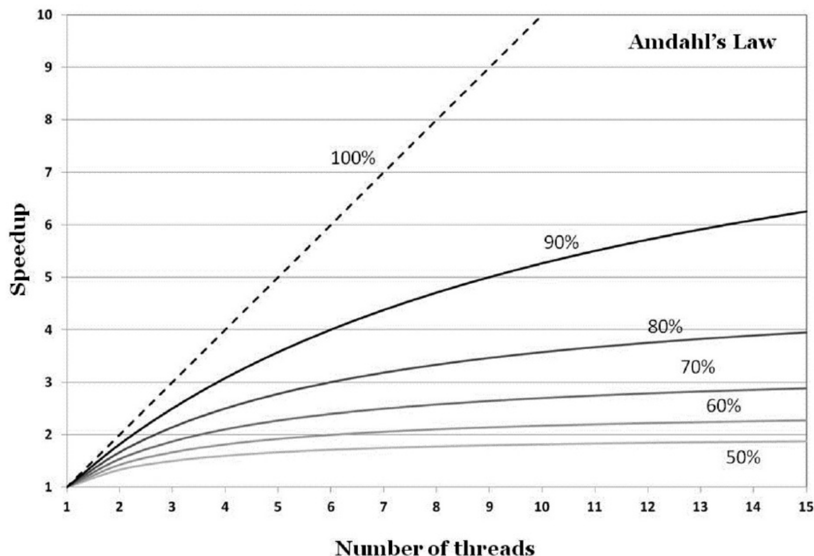

FIGURE 12.2: Amdahl's law for the speedup of programs as a function of the number of threads used.

We can test Amdahl's law by changing the number of threads we use in the parallel region of code. OMP comes with a suite of library functions that can affect hardware parameters. For instance, we can set the number of threads available to the program (up to the maximum number of threads we have on our system) by specifying that number as the argument to the function call `omp_set_num_threads`. In this way, we can see how much speedup we obtain using a different number of threads in the parallel region.

Before we throw our toys out of the pram and claim that parallel programming is fundamentally flawed it must be noted that Amdahl's law treats the problem as having a fixed amount of work to do and measuring the time taken to do that work. Gustafson argues that programmers tend to set the size of problems to use the available equipment to solve those problems within a practical fixed time. Hence, if faster, that is, more parallel, machines are available, larger problems can be solved in the same amount of time as smaller problems on slower, less parallel, machines. In essence, we can think of the individual processor workload as remaining fixed and as we

add more processors, we necessarily solve a bigger problem. The formula for Gustafson's Law is given by

$$S(P) = P - \alpha(P-1) = \beta P + \alpha \tag{12.3}$$

where α is the fraction of the code that cannot be made parallel that is, the serial portion of the program and note that $\alpha + \beta = 1$. It should be noted that in the formulations of Amdahl's law and Gustafson's law it is assumed that the parallel portions of code are uniformly distributed among all P threads, that is the threads are always doing useful work.

If we go back to our tea-making analogy of parallelism at the start of this section, it is mentioned that multi-core parallelism is like several people making one cup of tea at the same time, with only one kettle, one teabag, one cup, and so on. This is Amdahl's view. In Gustafson's view, each person makes one cup of tea where there is enough equipment, kettles, tea bags, cups, etc., for each person.

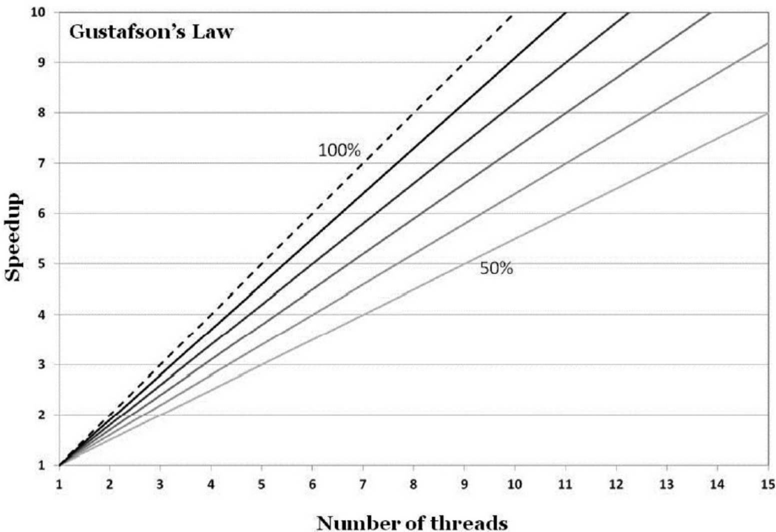

FIGURE 12.3: Gustafson's law for the (scaled) speedup of programs as a function of the number of threads used.

We can test Gustafson's law by increasing the size of our summation vector for a fixed number of threads; this should vary α such that

we should be able to quantify its value to some degree of accuracy for some given vector size.

In either case of Amdahl's law or Gustafson's law, it is clearly beneficial to make $\alpha = (1-\beta)$, the strictly serial portion of the code, as small as possible, thus making the parallel portion as large as possible.

EXERCISES

12.1. Write a program to determine the size of the level 1 and level 2 caches on your machine (and level 3 if your processor has it).

12.2. Write a program to determine the cache line size on your machine.

12.3. Check that the optimal index order for the naïve matrix multiplication algorithm is (i, j, k) for Fortran. Can you explain why that is/is not the case?

12.4. Apply manual loop unrolling to the outer loop of the matrix multiplication algorithm. Does this improve the performance of the multiplication and why/why not?

12.5. Write a program that performs matrix multiplication in parallel for different numbers of threads. Determine the value of α (or β) in your code for a given matrix size using Amdahl's law. What is causing this serial portion of code and can you quantify the amount of runtime it takes?

12.6. Using your value of α from the previous question does Gustafson's law (scalable speedup) hold true?

12.7. Choose any numerical quadrature we have discussed in the previous chapter and attempt to write it in parallel code. Why is writing parallel code for an initial value, ODE problem fundamentally flawed?

12.8. The Jacobi iteration scheme to approximately solve a general second-order ODE is given by

$$f_i^{(n)} = -\frac{1}{\theta}\left(\varphi\, f_{i+1}^{(n-1)} + \psi\, f_{i-1}^{(n-1)} - \delta\, x_i\right)$$

where θ, φ, and ψ are constants related to the coefficients of the differential equation; i is the index of the discrete grid approximation of the continuous space x; f is some quantity in that space; and n is the iteration count. Write a parallel program that exploits the "red-black" nature of this scheme. (Tip: Use the program(s) from Chapter 9 as a guide)

12.9. Attempt to repeat Exercise 8 but for the Gauss–Seidel iteration scheme (or SOR). Encounter any difficulties?

BIBLIOGRAPHY

The following is a list of reference literature that was useful in writing this book and developing the accompanying code. This also provides a guide to more general reading to the topics covered in the text.

General Physics

Longair, M. S., *Theoretical Concepts in Physics*, Cambridge University Press, 1984.

Orzel, C., *How to Teach Physics to Your Dog*, Scribner, 2009.

Susskind, L. & Hrabovsky, G., *The Theoretical Minimum: What You Need to Know to Start Doing Physics*, Allen Lane, 2013.

Computational Physics

DeVries, P. L., *A First Course in Computational Physics*, John Wiley & Sons, 1994.

Klein, A. & Godunov, A., *Introductory Computational Physics*, Cambridge University Press, 2006.

Koonin, S. E. & Meredith, D. C., *Computational Physics: Fortran Version*, Addison-Wesley, 1990.

Landau, R. H., Páez, M. J., & Bordeianu, C. C., *A Survey of Computational Physics: Introductory Computational Science*, Princeton University Press, 2008.

Pang, T., *An Introduction to Computational Physics,* 2nd ed., Cambridge University Press, 2006.

C++ Language

Meyers, S., *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*, O'Reilly, 2014.

Lippman, S. B., Lajoie, J., Moo, B. E., *C++ Primer*, Addison-Wesley, 2012.

Alexandrescu, A., *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001.

Linux and Unix

Kerrisk, M., *The Linux Programming Interface: A Linux and Unix System Programming Handbook*, No Starch Press, 2010.

Powers, S., Peek, J., O'Reilly, T., & Loukides, M., Unix Power Tools, O'Reilly, 2002.

Classical Mechanics

Goldstein, H., Poole, C. P., & Safko, J. L., *Classical Mechanics*, 3rd ed., Addison Wesley, 2001.

Kleppner, D. & Kolenkov, R. J., *An Introduction to Mechanics*, McGraw Hill, 1973.

McCall, M. W., *Classical Mechanics*, John Wiley and Sons, 2001.

Finite Element Method

Braess, D., *Finite Elements*, Cambridge University Press, 2001.

Johnson, C., *Numerical Solution of Partial Differential Equations by the Finite Element Method*, Cambridge, 1987.

Fourier Analysis

Briggs, W. L. & Henson, V. E, *The DFT: An Owner's Manual for the Discrete Fourier Transform*, SIAM 1995.

Dym, H. & McKean, H. P., *Fourier Series and Integrals*, Academic Press, 1972.

Folland, G. B., *Fourier Analysis and Its Applications*, Brooks/Cole Publishing Co., 1992.

Körner, T. W., *Fourier Analysis*, Cambridge University Press, 1988.

Tolstov, G. P., *Fourier Series*, Dover, 1972.

Walker, J. S., *Fourier Analysis*, Oxford University Press, 1988.

High Performance/Parallel Computing

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J. & Menon, R., *Parallel Programming in OpenMP*, Academic Press, 2000.

Dowd, K. & Severance C., *High Performance Computing*, 2nd ed., O'Reilly, 1998.

Hager, G. & Wellein, G., *Introduction to High Performance Computing for Scientists and Engineers*, CRC Press, 2010.

Koelbel, C. H., *The High-Performance Fortran Handbook*, MIT Press, 1994.

Williams, A., *C++ Concurrency in Action: Practical Multithreading*, Manning, 2012.

Interpolation and Approximation

Davies, P. J., *Interpolation and Approximation*, Blaisdell, 1963, (reprinted by Dover, 1975).

Nürnberger, G., *Approximation by Spline Functions*, Springer, 1989.

Powell, M.J.D., *Approximation Theory and Methods*, Cambridge, 1981.

Monte Carlo Method

Sobol, I. M., *A Primer for the Monte Carlo Method*, CRC Press, 1994.

Numerical Analysis

Demmel, J.W., *Applied Numerical Linear Algebra*, SIAM, 1997.

Golub, G. H. & Van Loan, C., *Matrix Computations*, 3rd ed., Johns Hopkins, 1996.

Higham, N. J., *Accuracy and Stability of Numerical Algorithms*, SIAM, 1996.

Overton, M. J., *Numerical Computing and the IEEE Floating Point Standard*, SIAM, 2001.

Ordinary Differential Equations

Hairer, E., Norsett, S. P. & Wanner, G., *Solving Ordinary Differential Equations I: Nonstiff Problems*, 2nd ed., Springer, 2000.

Hairer, E. & Wanner, G., *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, 2nd ed., Springer, 2004.

Lambert, J. D., *Numerical Methods for Ordinary Differential Equations: The Initial Value Problem*, 2nd ed., John Wiley and Sons, 1991.

Shampine, L. F., *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, 1994.

Partial Differential Equations

Folland, G., *Introduction to Partial Differential Equations*, 2nd ed., Princeton, 1995.

John, F., *Partial Differential Equations*, 4th ed., Springer, 1991.

Richtmeyer, R.D. & Morton, K. W., *Difference Methods for Initial-Value Problems*, 2nd ed., Krieger, 1994.

Taylor, M. E., *Partial Differential equations: Basic Theory*, Springer, 1996.

Tveito, A., Winther, R., *Introduction to Partial Differential Equations: a Computational Approach*, Springer, 2005 (2nd print)

Quantum Mechanics

Gasiorowicz, S., *Quantum Physics*, John Wiley and Sons, 1974.

Griffiths, D. J., *Introduction to Quantum Mechanics*, Prentice Hall, Englewood Cliffs, 1995.

Landau, L. D. & Lifshitz, E. M., *Quantum Mechanics: Non-Relativistic Theory*, 3rd ed., Pergamon Press, 1991.

Rae, A. I. M., *Quantum Mechanics*, 3rd ed., Institute of Physics, London, 1998.

APPENDIX

A CRASH COURSE IN C++ PROGRAMMING

Here we discuss the various things you need to know to get going writing and compiling C++ programs. The font `Consolas` in bold is used to highlight C++ keywords from the main text, for example, **`return`**. First, we start by discussing how to compile C++ source files into object files, executables, and libraries from the command line.

Command-Line Compilation

To build object files from C++ source files on the command line you invoke the following command:

```
g++ -c source.cpp -o object.o
```

Here we assume any included, non-standard headers are in the same directory as the source file. The `-c` flag tells the complier that it should omit the linking phase, that is, we are building object files, not executables. The `-o` flag means that we can specify the name of the object file immediately following said flag. If this is omitted the object file name is the same as the source file name but with a *.o* extension. We can also specify a directory located in the `-o` option as well as naming the output.

To build an executable from an object file we invoke the command:

```
g++ object.o -o executable
```

Here the `object` file is linked into the `executable`. If the `-o` flag is omitted the executable gives the default name *a.out*. Note that we can have multiple object files, complied from different source files, that all link to create the executable. For this simple illustration, we can put the compiling and linking phases into one:

```
g++ source.cpp -o executable
```

This command performs the compilation then linking phases of the build in one go and is the most sensible method for simple single-source files. If there are multiple source files that will create a single executable then we can just append these to the source file list, for example:

```
g++ source1.cpp source2.cpp source3.cpp -o executable
```

Typically, `source1.cpp` will contain the `main` function of your executable, whereas the other source files will contain function definitions or class implementations that are used by `main`. As a project grows it will accumulate an increasing number of source files. It becomes cumbersome and inefficient to keep having to write out awfully long compiler commands to build executables. It would be better to compile each source file (excluding the `main` source file) once into its own object file and in some way combine all object files into a single, linkable entity. This is what we call a library, and they come in two types, static libraries and shared (or dynamic) libraries.

Executables that are linked to a static library have all the object files contained in the library wrapped up with them. This means that the executable is a self-contained program, requiring no extra software to run, but the executable file itself can consume many bytes of storage, depending on the size of the executable's source code and the library (or libraries) to which it is linked. This means all

the functionality of the library is loaded into memory at runtime whether it is needed or not. Executables that are linked to shared or dynamic libraries load only the required functionality at runtime from the external library files (.so extensions on Linux/Unix, .dll on Windows). As a result, the executable file has a smaller storage footprint and generally requires less memory to run. Bearing in mind that large, complex projects may link with several different libraries thus a shared or dynamically linked executable should be preferred. However, consider that if you are building a dynamically linked executable for a different machine than your build machine, the executable is dependent on the target machine having the precise version of the shared (dynamic) libraries installed. A statically linked executable does not have this concern and so might be preferred if the project is relatively simple and only uses a small number of libraries.

Before describing how to generate these libraries we need to mention that with large projects it is conventional to split source and header files into their own directories. The complier can find standard headers because they are in "standard" directories, typically */usr/include* or */usr/local/include* or */opt/local/include* depending on your OS, the compiler is configured with these locations when it is installed. However, if project-specific header files are located separately from the source files, the compiler will not be able to find them unless we tell it where to look. To do this we add the location to the complier's include search path using the option `-I` in the `g++` invocation. To illustrate, this looks like:

```
g++ -I/path/to/project/include source.cpp -o
executable
```

To create a static library file, we can use the archiver program invoked with the following command:

```
ar rcs libname.a <object file list>
```

where the `object file list` is simply a list of all the required object files, for example, *object1.o object2.o* …. For static library files on Unix/Linux-like systems, the library "name" must be prepended with "lib" and end with the extension ".a". You can think of

the archiver as simply copying all the object files in the list into the library file. The options `rcs` mean replace object files of the same name (if the library already exists), create the archive (if it does not already exist), and `s` makes the archiver also write an object-index into the library file for faster access at runtime; this is equivalent to running `ranlib` on the archive (library) file.

To create a shared (object) library we use `g++` directly. First, we use `g++` to create our object files but so that they contain position-independent code, for example:

```
g++ -c -fpic source1.cpp -o object1.o
```

To understand what the option `-fpic` does would require an understanding of assembly language but essentially means the code in the object file can be placed anywhere in memory and still work, which is a requirement for library files. Note that the option `-fpic` may now be redundant as all code generated from modern `g++` versions should be position-independent.

With the object files built we invoke `g++` again to create the shared library file:

```
g++ -shared -o libname.so <object file list>
```

Here the shared library name requires the “lib” prefix, just as a static library does, but has the “.so” extension to identify it as a shared (object) library.

To build executables linked against these libraries we invoke the `g++` command with the `-L` and `-l` flags, for example:

```
g++ [-static] -L/path/to/library -o executable
main.cpp -lname
```

We omit the “lib” prefix and the dot extension from the library name when invoking the complier to build the executable. If we have both a static and a shared library with the same “name” at the specified path, `g++` will use the shared library as default. In this case, to use the static library, we must provide the `-static` option. You can link with multiple libraries if necessary, each of which might be in a different

location. For more on building C/C++ libraries see *cprogramming.com/tutorial/shared-libraries-linux-gcc.html*, note `g++` can in essence be used interchangeably with `gcc`.

We have not mentioned other useful options that you can pass to `g++` as they were not pertinent to the discussion. To use modern C++ language features, you will need to provide the `-std=c++11` option at least. Other standards exist including `c++14`, `c++17`, and `c++20` (they are named for the year they were released), though you will have to check if your compiler version can support these standards. The option, `-O`, which is a capital "o" not zero, allows you to specify a complier optimization level. The level can be zero, one, two, or three; the default is zero. Zero implies no optimizations—the code that you see is the code that you get, useful for debugging purposes, and is typically used with the `g` option which generates debugging information for use with gnu debugger, *gdb*, for instance. Level one switches on some optimizations and gives a significant improvement in performance. Level two switches on more optimizations and will give a noticeable performance improvement over level one. Level three is the most aggressive optimization option and is not generally recommended as it *may* break the semantics of your code and typically offers only minor improvements in performance over level two, and in some cases may make it worse. The `-W` option allows you to specify warnings the complier should look out for when parsing your code. It is recommended you always use `-Wall` which switches on all the standard warnings. It is also recommended that you treat any warnings generated during compilation as errors. If you use any macro-defined names, for example to conditionally guard code for platform specifics or "debug" code, you can define them using the `-D` option.

Doing all of this on the command line can become tedious. Once you have mastered the simple command-line invocations as outlined above you should instead learn how to write and edit *Makefiles* such that all the above can be reduced to the deceptively simple command:

```
make
```

or

```
make target
```

where `target` is the name of a "recipe" in the *Makefile*. The code found at the GitHub repository for this book has basic Makefiles that will build the source code into a library and executables that link with that library. These are by no means optimized and should only be used as a basis to learn the (GNU) Makefile framework; *gnu.org/software/make/* is a good starting point for further investigation.

Guidelines to Good Code

C++ is a free format language in that the spacing is of no importance to the complier. However, spacing is important to people, and the readability of your code matters. There are some "best practice" guidelines about how we should layout our code:

- have no more than one statement per line;
- use blank lines between functions;
- logically group sections of code;
- use consistent indentation;
- use space around binary operators; and
- do not use space between a unary operator and its operand.

Statements must be explicitly terminated by the semicolon. However, any statements that run over a single line do not require a continuation character to indicate overrun, with the limitation that identifiers, keywords, and literals cannot be broken over new lines.

Identifiers are the names we give to variables, functions, classes, and objects in C++. They must abide by the following rules to be a valid name:

- consist of letters and digits;
- must start with a letter (not a digit);
- underscore is considered a letter;
- are case sensitive; and
- are not keywords, for example, **`return`**, **`using`**, **`class`**, etc.

C++ is case sensitive. This means that the identifiers `different` and `Different` are different.

There are also a few identifier recommendations that do not need to be adhered to but are considered to improve the style of C++ code. First, use descriptive names that are not abbreviated, for example *message* not *msg*. However, be sensible. A 25-character descriptive name may make the variable easier to identify but the whole code more difficult to read; this is where comment lines are useful. If the identifier is made up of more than one word then you should make that clear by either using an underscore between the words, or capitalizing the subsequent words, for example *smoothing_factor* or *smoothingFactor*, say. Try to avoid using identifiers that are only subtly different as this may lead to confusion for example, `different` and `Different`. For data members of a class, it is convention to identify them as such by prepending their names with a single underscore, _, or the combination `m_`.

What is C++?

C++ is an abstract, high-level, *complied* programming language. It essentially acts as a translator between us as humans and the computers as machines. We write programs in source files (typically ending in the extension *.cpp* or *.cxx*) that can be understood by humans and these are compiled or translated into machine language files that can be understood by the computer. These translated files are known as object files that when linked together produce the binary executable (otherwise known as a program) that can then be run on the computer. This is different from scripted languages (MATLAB, Python, Ruby, etc.) which are *interpreted* at runtime; the machine language exists already and is unchangeable, rather than being created from source code.

C++ supports a combination of procedural, functional, object-orientated, generic, and (template) metaprogramming features.

C++ executes one statement at a time in the order that they were written (ignoring parallel or threaded execution), that is it follows the procedure of the source code. A statement is one or more expressions terminated by a semicolon. An expression can consist of a primitive-type declaration, variable initialization, variable assignment, mathematical operations, logical and comparative operations, and control flow and looping constructs. We describe these properties next, beginning with primitive types in C++.

C++ Primitive Types

C++ primitive type	Representation
int	32-bit integer
float	32-bit floating-point number
double	64-bit floating-point number
char	8-bit character
bool	Boolean logical (true or false; 8-bit)

The table above lists the most common primitive types in C++. The bit lengths quoted are those found on a typical off-the-shelf machine but could vary depending on the platform and/or the operating system.

Although primitive integer type bit lengths can be modified using the type modifiers *long* and *short* you should prefer using the C type definitions found in the GNU C library, accessed via the header *stdint.h* (or *cstdint.h* for C++) that can be used to declare integers of exact size for any machine. They are declared as `intN_t`, where N represents the bit length you require (8, 16, 32, or 64). To declare unsigned versions of these integers just pre-pend the letter *u* to the type, that is, `uint8_t` for an unsigned 8-bit integer.

In modern C++ when initializing a primitive variable, you should prefer to use the curly brace initializer (`{}`) over the assignment (=) symbol. For example:

```
int x{42}; //some meaningless value
double posixTime{0.0}; //1970-01-01 00:00:00
```

C++ Operators

Operators can be classified according to the number of operands they take. Unary operators take a single operand, for example the unary minus which returns the negative of the operand. Binary operators take two operands, for example the binary plus which returns the sum of its two operands.

C++ has the usual mathematical operators: +, −, *, and / (add, subtract, multiply, and divide). Note that the asterisk, *, has multiple meanings, depending on context; we will return to these in

due course. The modulo operator (**%**) returns the *remainder* of the division between its operands, for example 12 **%** 7 is 5. These operators have a precedence order, and this is an important part of the C++ language (as it is with any other language). For example, the multiplicative operators (*, **/**, and **%**) have a higher precedence than the additive operators. Full list of operator precedencies can be found on the internet.

Assignment (*operator*=) in C++ is an expression that returns a result as well as having a side effect on the left-hand operand; the left-hand operand is overwritten with the value from the right-hand operand, this is what we mean by assignment. The result (or return value) from the assignment is what is called an lvalue (left-value). An lvalue is a variable or object that has a memory address and may be used as the target of assignment. To illustrate this feature, we can write the following valid statements:

```
int x, y, z;
x = y = z = 0;
```

Here x, y, and z all receive the value zero. Operator associativity defines how the operands of binary operators will group, either to the left or to the right. The multiple assignment statement above has right associativity; that is z is assigned zero, the result of that assignment is assigned to y, and finally x is assigned the result of the assignment to y.

Relational operators relate one variable to another and are commonly used in conditional statements that require a Boolean result. We can test equality between variables using the *equality operator* (==), and similarly the *inequality operator* (!=) that is, not equal to. A common mistake is to use a single equals sign to try to test equality. This instead will assign the value on the right-hand side to that on the left. Additionally, as the assignment returns an lvalue it is likely the condition will evaluate to a valid Boolean (the value is implicitly type converted to a **`bool`**) and lead to some odd behavior during the execution of the program. Some compilers may not pick up on this programming mistake (it is syntactically valid code) so it is something to watch out for. Other relational operators include greater

than (>), less than (<), and the compound versions of these; greater than or equal to (>=) and less than or equal to (<=). Note that the compound symbols follow the order of the English phrase.

The logical *AND* and *OR* operators are written as *&&* and ||, respectively. These should not be confused with the operators *&* and | that perform bitwise comparisons of their operands. The logical operators are binary operators and evaluate their operands from left to right. They are also shortcut operators in that they will not evaluate further than they need to. To explain, if the left-hand operand of the *&&* operator evaluates to false it does not evaluate the right-hand operand; the overall outcome is already false. Similarly, if the left-hand operand of the || operator evaluates to true it does not evaluate the right-hand operand; the overall outcome is already true. Keep this in mind when using these operators.

There is one operator that takes three operands and thus is referred to as *the* ternary operator; the conditional expression that is invoked with the question mark symbol. The operands are the condition on the left, and the two possible result expressions on the right; result if true followed by result if false separated by a colon. A simple example of this is finding the maximum between two values:

```
int max = a > b ? a : b;
```

This reads as "if a is greater than b then max is initialized with a, else max is initialized with b."

The condition expression may also be used as an lvalue. For example, we may write

```
first > second ? first : second = 0;
```

This reads as "if first is greater than second then first is assigned zero, else second is assigned zero."

It is often found in programming that we must write expressions like

```
total = total + subtotal;
```

The compound operator provides a convenient way of writing this sort of expression

```
total += subtotal;
```

This becomes easier to read once you are familiar with the syntax. The expression `a += b` can be read as `a` incremented by the value of `b`. For complex data types the shorthand, compound expression can also be more efficient than the long hand version. All the mathematical operators (+, –, *, /, and %) have a compound operator.

The most common value by which we increment or decrement a value is by one. C++ further refines the compound operator when just incrementing by 1. For example

```
value += 1;
```

Can be written as

```
++value;
```

Note that this is the prefix version of the increment operator. It adds one to the value and returns that new value. We can also write the post-fix version

```
value++;
```

The difference between the two is subtle but can be particularly important. The post-fix version takes a reference of the value, stores the current value, increments the reference, and returns the stored value. Let us illustrate this so that it is clear. Writing

```
result = ++pre_increment;
```

is equivalent to,

```
pre_increment += 1;
result = pre_increment;
```

Whereas writing

```
result = post_increment++;
```

is equivalent to,

```
result = post_increment;
post_increment += 1;
```

Be aware of this difference as it can very easily cause bugs in your code that can be difficult to track down.

C++ Enumeration Types

There are some types whose purpose is to take only one of a few named values. For example, (UK) traffic lights have a fixed set of colors (red, amber, and green), or the size of clothes has a fixed set of values (XS to XXXL). A command to set traffic lights to purple or trying to buy clothes that are A4 in size would be nonsensical.

The enumeration type in modern C++ is specified by the keyword combination **`enum class`**, sometimes referred to as the scoped enum. They provide a mechanism for introducing a new type with a name and a related set of constant values. Each *enum* definition is a new type that is separate from other *enum* types, and as such they cannot be mixed. A scoped *enum* value is represented like an integer but it is not automatically converted to one. Similarly, an integer cannot be automatically converted to an *enum* value, this would undermine the point of the type system. By default, the enumeration values are numbered from zero upwards in steps of one. We can specify a different numbering system by explicitly assigning numbers to the *enum* values (so long as it is greater than the previous element's number representation). To demonstrate let us write a scoped *enum* type for exam grades:

```
enum class grade {A, B, C, D, E, F, U=99};
```

In this example, `A` is represented by zero, `B` by one, and so on up to the grade `F` where we jump to the grade `U` that has been assigned the value 99. Initializing a variable to be of some scoped *enum* type

is the same as declaring any other variable; the type name followed by the identifier with the specific initialization value contained in curly braces (we have to use the scope resolution operator for scoped enums):

```
grade yours {grade::A};
```

To illustrate the point about not being able to mix *enum* structures, we could set up an *enum* structure for the seasons and the days-in-the-week:

```
enum class season {winter, spring, summer, autumn};
enum class day_week {mon, tues, wed, thurs, fri,
   sat, sun};
```

Then declare and initialize some variables based on these *enum* structures:

```
season now {season::summer};
day_week today {day_week::fri};
```

However, we cannot initialize (or assign) the variable `today` with the *enum* value `spring`, say, as today is of type `day_week` not `season`. This is obvious in our example but is something to be aware of in larger programs.

Please refer *item 10* of Scott Meyers' book *Effective Modern C++* (see Bibliography) for a more detailed discussion of scoped enumeration types and how to deal with them.

Control Flow

The control flow of a program can be categorized into four types:

- **Sequence**: Where program flow follows a simple sequential path executing one statement after another. The primary sequential structure is a compound block statement that is a series of statements inside curly braces: {}.
- **Selection**: Where only one path out of several possibilities is taken. Simple selection is conditional execution of a

statement or block of code guarded by an expression which will have the value true if the guarded code is to be executed: this is the **`if`** statement. An **`if else`** statement provides two alternative paths of execution: the true or false evaluation of a control expression determines which branch is taken. A **`switch`** statement supports a multi-way branch based on the value of a control expression known as the **`case`**.

- **Iteration**: Where one statement or block of code is repeatedly executed. A simple **`while`** structure executes the same code while a control expression has a true value (hence the name), terminating execution when the expression evaluates to false. C++ provides three forms of loops that are suited to the three most common iterative styles; **`while`**, **`do while`**, and **`for`**.
- **Transfer**: Where the point of execution jumps to a different point in the program. Although the point where execution jumps to is clearly defined, the use of transfer of control statements usually leads to programs that are difficult to understand. The simplest transfer structure is the **`goto`** statement which jumps to a labeled point in the code; we discuss the appropriateness of the simple **`goto`** statement in modern programming in the final paragraph of this section. The structured *goto* statements of **`break`** and **`continue`** jump to clearly defined points within other flow control structures and their use is okay but should be commented on if their intention is not immediately apparent in the source code.

Sequence

In C++, a simple statement is any semicolon terminated expression. Declarations are ordinary statements. This is unlike other languages such as Pascal and Fortran, which require the declarations to be specified before any executable code.

A statement can also be a compound statement; a sequence of statements delimited by an opening and closing brace, sometimes referred to as a block, and there is no terminating semicolon after the closing brace. The braces define the scope of the compound

statement and are equivalent to the begin and end keywords found in other languages. Variables declared in blocks are in scope from the point of their declaration until the corresponding closing brace. C++ does not use name-matched delimiters like languages such as Visual Basic, Ada, or Fortran where, for example, the "if" keyword is matched with an appropriate end keyword such as "endif."

Note that the definition of a statement and a compound statement is recursive (a compound statement contains statements that may themselves be compound statements which contain statements that may themselves be compound statements, and so on). A compound statement may also be empty.

Selection

The primary structure for decision-making is the **`if`** statement. A control expression is used to determine which branch of the two-way fork statement is taken. If the expression is true, the first branch (the **`if`** body) is taken but if the expression is false then the second branch (the **`else`** body) is taken. The **`if`** body and **`else`** body contain statements that are either a single ordinary expression statement or a compound statement delimited by curly braces. The **`else`** clause is optional and if it is omitted then code continues to execute from the closing brace of the **`if`** body should the control expression evaluate to false.

The **`if`** control expression expects a bool type expression; anything else is an error. For backward compatibility with previous versions of the C++ language and the C language, an integral expression can be narrowed into a **`bool`** by the compiler. Typically, a **`char`** or **`int`** expression may be used in place of the **`bool`**. When converting an integer to a Boolean a value of zero is false and any other value (positive or negative) is true.

Languages that use name-matched delimiters also support a multi-way if statement where additional branches are provided using "else if" constructs which are like the "if" component but occur between the "if" and "else" statements. Any number of these branches is allowed. C++ does not provide an "elseif," "elsif," or "elif" statement because the same effect can be achieved using multiple if statements. Note the **`else`** and **`if`** are separate keywords

so that the **else** clause of the first **if** statement can itself be another **if** statement. The tests are evaluated from top to bottom and as soon as an **if** or **else if** test evaluates to true that branch is taken.

```
if (some_condition) {
      //execute if some_condition is true
} else if (other_condition){
     //execute if some_condition is false and
     //other_condition is true
} else {
     //execute if both some_condition and
     //other_condition are false
}
```

The **switch** statement jumps to a chosen **case** label. The program execution continues from that point onwards until it hits a **break**. The **break** statement branches the control flow to the end of the **switch** statement. If no **break** is provided, control will drop into subsequent **cases**. This is known as fall through and is a valid programming design but can cause bugs in your program if it is not what was intended. Intentional fall-throughs should be commented as such otherwise other programmers could read it as a mistake. It is usual to include a **break** statement even after the last statement in a **switch** for consistency. The **default** label is selected if the expression evaluation does not exactly match one of the specified **case** labels. Note that a **default** label does not need to be specified. If the **default** is not specified and the expression evaluation does not match one of the case labels, then program execution continues from the end of the **switch** block.

```
switch (some_case) {
case 0:
       //code for case 0
       break;
case 1:
       //code for case 1
       break;
```

```
default:
        //code for default case
        break;
} //break transfers control here
```

The **break** statement can also be used within a loop structure to transfer control to the end of the loop. This use of **break** is discussed at the end of this section along with the keyword **continue**.

It is important to remember that you can only use simple types, like **int**, **char**, and **enum** (both scoped and un-scoped) as the **case** for **switch** statements; floating-point numbers, strings, and other complex types cannot be used.

The **switch** statement is generally more efficient than a complex **if** statement, particularly as the number of choices increases. If the branch requires three or more choices, consider whether the **switch** statement is more appropriate than multiple **if else** statements.

The **case** options can only be used to select exact values for the control expression: ranges of values (such as greater than zero, or 1 to 9) are not allowed. However multiple **case** labels can be specified for the same block of code; as discussed above the code falls through the cases until a **break** statement is encountered.

Iteration

The fundamental looping construct in C++ is the **while** loop. In this structure, a statement (called the *while body*) is repeatedly executed while a control expression evaluates to true. The repetition ends when this condition evaluates to false. Because the condition is checked before the loop body it is possible for the body to never be executed, the condition is false initially.

```
while (count++ < total) {
        //loop body
}
```

A classic mistake with a **while** loop is to forget to update the variable or variables used in the control expression in the loop body. This has the effect of creating an infinite loop: one that never

terminates. The **for** loop (described later) formalizes the update expression into its syntax so it is not as easy to make these kinds of mistakes.

Like the **if** statement, an integer expression can be narrowed to a **bool** where the value zero is false and any other value is true. An infinite **while** loop can be created intentionally by simply providing an argument of 1 or **true** in its control expression. Some programs are required to run indefinitely such that infinite loops are required, and we provide some sort of user exit command to terminate the loop (escape key, or a "quit" option). When developing programs, the key combination "ctrl-c" in an active terminal will send an interrupt signal to the program running in that terminal that, in general, will cause it to terminate.

C++ provides a generic looping construct with the **for** loop, rather than a traditional counted loop as in other languages. Here a classic **while** loop is formalized into a single construct. A classic **for** loop has an initialization clause, a continuation condition, and an update expression; the **for** loop provides placeholders for these three components separated by semicolons.

```
for (int i = 0; i < total; i++) {
       //loop body
} // i is destroyed once loop ends
```

We can in fact provide only the continuation condition leaving the other components of the loop blank. However, this undermines the point of the **for** loop; why not just use a **while** loop? An infinite loop can be created by leaving all three components blank (you still need to leave them in the separating semicolons). Note that variables declared in the initialization of the **for** loop are scoped to that construct; their lifetime ends when the loop ends.

C++11 introduced the range-based **for** loop used to provide a more readable equivalent to the classic **for** loop when operating over a range of values, for example the elements in a container type. The range-based for loop has a range declaration and a range expression, separated by a single colon. For example, we can loop over the elements in vector with the following syntax:

```
std::vector<int> v {1, 2, 4, 8, 16, 32};
for (auto i : v) {
     //loop body, i has type int, access by value
}
```

This can be read as "for each element `i` in vector `v` perform the loop body." Note that you can also access the elements of a container by reference and forwarding reference the details of which are beyond the scope of this brief discussion (see *cppreference.com* for more details).

An alternative loop is provided by the **`do while`** construct where the loop test is evaluated after the loop body has been executed. This means that the body must be executed at least once and is ideally suited to validating data before allowing program execution to proceed. It is far more common to use compound statements rather than a single statement for the body of a **`do while`** construct, but it depends on the application. Note a semicolon must follow the end of the **`while`** condition.

```
bool input_bad {true};
do {
     //loop body, validate input
} while (input_bad);
```

Transfer

Earlier I mentioned the **`break`** and **`continue`** keywords when applied to loops. The **`break`** command used within a loop will terminate the application of the loop, and any code located after the end of the loop will continue to execute. Whereas the **`continue`** command will cease the execution of the current iteration and move the loop to the start of the next iteration. The use of the **`break`** and **`continue`** commands within loop constructs can indicate poor logic that can be rewritten to remove their use, potentially making the code more readable. Where a **`break`** or **`continue`** absolutely must be used in a loop structure a comment about their intention is recommended.

One final control flow construct exists within the C++ language. The `label` and **`goto`** `label` expression are very much a hangover from the early days of programming. If you have ever programmed in FORTRAN 77 or BASIC, then you will have used this construct; it exists because early languages did not have a formal looping construct so programmers would have to build ones manually using the **`goto`** `label` command. In C/C++, you set up a named label somewhere within your code by writing an identifier for the label followed by a single colon. You can then branch control flow to that label at ***any point*** in your code by using the **`goto`** command followed by the label identifier. This non-continuous control flow does not suit modern structured programs and can make code quite unreadable to the un-trained eye, as well as frustrating to debug.

Next, we discuss exceptions in C++. Exceptions provide another way to transfer control from one part of a program to another but only under exceptional circumstances (typically errors).

Exceptions

An exception is something that happens that is out of the ordinary during code runtime. Typically, it is an error that arises during the execution of a program, such as an attempt to dereference a container object beyond its range or a system out of memory. When an error occurs, C++ will cease the normal execution of a program and report the error by generating an error message and passing it to the context that can handle the error. This is what is known as "throwing an exception."

Program errors can be split into two categories: logic errors that are caused by programming mistakes, and runtime errors that are beyond the control of the programmer, for example memory leaks in external libraries, lost network connections, missing databases, and so on. In general, the preferred way to report and handle both logic errors and runtime errors is to use exceptions. An exception makes a program acknowledge the occurrence of an error and gives it the ability to handle that error. Any unhandled exceptions stop program execution. When an exception is thrown the execution jumps to the point in the code that can handle that exception. We will explore classes and object shortly, but it is important to know that at the

point of an exception being thrown all objects in scope are destroyed via well-defined rules (object destructors)

We will also cover class inheritance in due course but it is enough to know for now that all exceptions generated by the standard library inherit from the *std::exception* class. This is an important feature in the way exceptions are handled in C++. Many of the standard exceptions can be used in your code by including the header *stdexcept* and you can use them as base classes for your own, custom exceptions. The message that the standard exceptions carry can be accessed by the member function "`what`" that returns a C-style character string pointer to the message.

Exception handling in C++ consists of three keywords: **`try`**, **`throw`**, and **`catch`**. The **`try`** statement allows you to define a block of code to be tested for errors while it is being executed. The **`throw`** keyword can be used when a problem is detected, and which allows us to report a custom error. The **`catch`** statement allows you to define a block of code to be executed if an exception is thrown in the **`try`** block; this is what we mean by the context that handles the error. A **`try`** block must have at least one corresponding **`catch`** block but could have several to catch different, specific types of error.

Although it is typical to throw an exception type that has inherited from *std::exception*, in C++ *any* type may be thrown and caught.

```
try {
          int legal_age {21};
          std::vector<std::string> the_list {
                "Elvis Presley",
                "Kayne West",
                "Judy Dench",
                "Lucy Liu"
          };
          int my_age {18};
          std:string my_name {"Joe Bloggs"};

          if (my_age < legal_age) throw(my_age);
          bool in_the_list {false};
          for (auto name : the_list) {
```

```
                in_the_list = my_name == name;
            }
            if (in_the_list == false) throw(my_name);
    }
    catch (int your_age) {
            std::cerr << "You must be 21 or over to
                        enter.\n"
            std::cerr << "Your age is " << your_age
                    << std::endl;
    }
    catch (std::string your_name) {
           std::cerr << your_name << " is not on the
                        list\n"
           std::cerr << "you're not coming in!"
                   << std::endl;
    }
```

Note that, unless the handling code terminates or does some other control transfer, code execution continues from the end of the last catch block. In the example above this means that `the_list` is **not** checked as an exception of type **`int`** has already been thrown. Bear this in mind when using try-catch blocks.

If an exception is thrown that does not match the type of any of the **`catch`** handlers, the program is terminated. C++ provides a means to catch any unspecified exceptions via the ellipsis `(...)` token.

```
    try {
           //code to be tested
    }
    catch (int e) {
           //handle "int" exception
    }
    catch (std::string e) {
           //handle "string" exception
    }
    catch (...) {
           //handle any other exception
    }
```

It is important to know those catch handlers are matched in the order they appear, so you want the most specific types first and the "catch all" handler should always appear last. This statement also applies when you are throwing standard exceptions (or custom exceptions) with class hierarchy; the order should follow the hierarchy from most derived (specific) class first to least derived (potentially base) class at the end. As always "catch all" should be the very last handler.

Before we move on to the next section discussing functions, we should mention that in the simple examples above we have thrown *and* caught exceptions by-value. This is okay for primitive or simple types but in general, we should throw exceptions by-value but catch by-reference and typically by **`const`** reference. These concepts are discussed in the next section.

C++ Functions

Functions written in C++ will generally require a prototype, which is usually called its declaration. You can think of the prototype function as a blueprint; it tells the compiler what type the function will return, what the function will be called, and what parameters it takes as arguments. The function does not have to return a value, in which case the return type is declared as **`void`**, and there can be more than one parameter or none. For the latter case, we use empty parentheses. To demonstrate, a prototype for a power function may be written as follows:

```
float power(float value, int m);
```

This tells us that the `power` function raises the floating-point `value` to the integer power `m`, returning the result as a floating-point value. Actually, it does not. All this prototype tells us is that the function `power` accepts a **`float`** type and an **`int`** type as inputs and returns a **`float`** type as an output. The implementation of the function that is, what the function actually does, is taken care of by the function definition which we describe shortly.

With this function prototype declared before the main function we can use it in our main by assigning the return value to a **`float`** variable, for example:

```
float power(float value, int m);
int main(){
      float x {2.f};
      int n {3};
      float y {power(x, n)};
}
```

Here we have "called" the power function and the main function is referred to as the calling environment (or the caller). Note that although the above example is syntactically correct it will not compile as we have yet to *define* the power function (you will get a linker error if you try, namely *unresolved external symbol*). The definition of a function is the block of code that provides the instructions as to what the function is to do. We can either provide the definition of the function where we declare the function or provide the definition in a separate location. It is a good habit to acquire to separate the declaration from the definition (something which is very much appropriate to classes, there it is referred to as interface and implementation). Remember that *main* is the *only* function in C++ that we must declare *and* define in the same place. A definition for the `power` function could look like:

```
float power(float value, int m) {
     return m == 0 ? 1 : value * power(value, m-1);
}
```

The power function here is recursive, that is, it calls itself, and you should satisfy yourself it does as advertised. As an aside, this function definition is not fully fledged or optimized, what happens if `m` is negative? Also, could this be rewritten as a loop?

The identifiers in the argument list of the declaration are merely placeholders for the actual argument identifiers used in the definition. In fact, the declaration does not even require you to provide identifiers for the arguments at all, only the types. However, unless is it noticeably clear from the context what the arguments are you should provide descriptive names for them in the prototype. The argument identifiers

in the definition can be as short and as cryptic as you like but we refer you back to the recommendations on identifiers for best practice. For larger and more complex programs that use several different functions, it is better to further separate declarations and definitions by placing them in their own header and source files, respectively.

How does C++ pass variables from the calling environment to a function? Default behavior of C++ is to pass arguments by value. That is, the function makes its own copy of the argument to work with and does not modify the variable in the calling environment. Pass-by-value is fine for primitive data types as they are small in terms of memory consumption and making a local copy of them is cheap and quick. However, what if the argument we pass is an object of a class that takes up a large portion of memory, a matrix for instance? Taking a local copy of that object could be prohibitive both in terms of memory consumption and computational effort. In C++ we get around this restriction by passing the argument by reference rather than value. In this case, the function does not make a local copy of the parameter but works directly on the variable in the function definition. To signify a reference to a variable the symbol & is added after the type declaration in the function parameter list. A reference can be thought of as an alias for the corresponding argument in the calling environment.

Astute readers will have spotted that by passing-by-reference gives the function permission to modify the contents of the variable in the calling environment. This might be desired. As a contrived example of this, we might want to be able to nullify matrices (make all elements zero) and we write a function to do this. Rather than having to copy the entire matrix into the function, modify its elements, and return the result to the calling environment, which would require another copy, we could just pass it by reference. The work would be done on the matrix as if the function definition were written directly in the calling environment.

What if we do not want the function to modify the variables declared in the calling function, but still need to pass them by reference for efficiency? C++ includes the keyword **`const`** for this purpose; **`const`** is a very hardworking keyword in C++ and pops in all kinds of contexts. When **`const`** is used in conjuncture with a variable declaration we mean that the contents of the variable are to

remain constant throughout the lifetime of the variable. Declaring a reference parameter **`const`** ensures the function cannot change the contents of the variable that is aliased by that reference. If you try the compiler will complain. You can also declare a pass-by-value parameter **`const`**, but this has a subtly different meaning; the local, function variable, that is, that declared by the function parameter, is *initialized* with the value of the argument variable from the calling environment, and that *local* variable then cannot be changed by the function.

Generally, you cannot return a variable from a function by reference. As soon as a function goes out-of-scope that is, it executes its **`return`** statement, all local variables are destroyed; when returning by value the **`return`** statement creates a temporary local variable that is copied back to the calling environment. If we were trying to pass a local variable back to the calling environment by reference, we would end up with a reference to nothing of relevance, known as a dangling reference. Return-by-reference is possible if the parameter was passed in by reference in the first place, an example of this can be found in overloaded versions of the output stream operator; we discuss overloading shortly in the next section.

Function parameters can be given a default argument in the function declaration. If a function with default arguments is called without passing arguments, then the default values are used. Not all parameters have to be given a default argument but after a parameter is given a default argument all subsequent parameters must also have default arguments. To illustrate this point let us look at our example `power` function declaration with a default argument:

```
float power(float value, int m = 2);
```

Our power function can now be called with only one argument and in this case, it would return the square of the `value`. If we were to declare the `value` parameter with a default argument, then we must also declare the m parameter with a default argument:

```
float power(float value = 1.0, int m); //error, won't
   compile
float power(float value = 1.0, int m = 2); //okay
```

Default arguments should be used with care and consideration as they can cause problems when updating and maintaining code. This is especially true when providing function overloads, which we will discuss next.

Function Overloads

C++ allows several functions with similar purposes to share the same name or identifier. We say these functions are overloaded. Function overloading is a feature that can be used effectively to simplify the intent of code by using the same name to refer to different physical functions that perform a conceptually similar role. Overloaded functions will differ by the number of parameters, or the parameter type, or both the number and type of parameters. Parameter order and default arguments are also considered; C++ will try to implicitly convert arguments if necessary. Overloaded class member functions can also differ on the `const` qualifier as explained later. Class constructor overloading is a good example of function overloading: there may be more than one way to initialize an object, again this is discussed in a later section.

The overloaded functions must have unique signatures so that the complier can distinguish between them. Each of these overloaded variants requires a separate function definition. Note that a function cannot be overloaded solely based on its return type: the return type is not considered part of the signature. Where the function signature is unique, the return type need not be the same as other functions of the same name. For example, it is possible to overload a function using a template, where the return type and argument type(s) are decided upon at compile time. We discuss template functions in detail shortly.

Using the function name, and the types, number, and order of the parameters the compiler can generate a unique name for each function. This is called the name mangling and is applied to all C++ function names. Errors from the linker often show these mangled names when reporting unresolved symbol errors which are usually attributable to missing function definitions or typing errors.

We mentioned previously that care and consideration should be taken when providing default arguments in function declarations

and here we provide a reason. If we are not careful when specifying default arguments, overloading a function could lead to ambiguous function calls. To demonstrate this, we return to our `power` function example with the default argument:

```
float power(float value, int m = 2);
```

and provide an overloaded declaration (admittedly contrived), also with a default argument:

```
float power(float value, float m = 2.0);
```

If we call the power function with only one argument, which power function gets called? The function call is said to be ambiguous and the compiler will give an error to that effect.

When the complier attempts to match an overloaded function against a function call it will insert type conversions of the arguments as necessary to obtain a match. The complier prefers the function that requires the fewest argument conversions when resolving the overloaded function to use.

Template Functions

Overloading functions is usually an attempt to allow us to use a particular function for different types; the overloaded functions will only differ by the type of the arguments and often, consequently, the return type. These overloaded functions will typically share identical code in their definition apart from the type of the variable that is being worked on. As an example, consider a function overloads that returns the larger of two input values:

```
int max(int a, int b) {return a > b ? a : b;}
float max(float a, float b) {return a > b ? a : b;}
double max(double a, double b) {return a > b ? a : b;}
```

Instead of needlessly repeating code we can use the template feature of C++ to write a single declaration and definition of a generic function that can be used autonomously with any acceptable type. Without the template feature, we would have to explicitly

overload the max function providing a declaration and definition for every conceivable type. However, using the template syntax we can write a possible function declaration and definition as

```
template<typename T> T max(T a, T b) { return a > b ?
   a : b;}
```

We can now use the max function for any type we like with the caveat that the template type **T** has the greater than operator defined, if not the code will fail to compile. Note that it is recommended that a template function be declared and defined together in the header file as explained shortly.

Here the template type **T** is a virtual placeholder for a concrete type the details of which will be provided at (a future) compile time, that is, when we produce a binary file. This essentially leaves a hole in the definition of the function. Machine code requires that a concrete type be specified to fill the hole that is left missing. Therefore, it is most convenient for the template function declaration and definition to reside together in a header file that we include to use the function. Otherwise, the definition must reside in each source file that calls the function; this would be clumsy programming and subverts the point of using a template function.

When calling a template function in general we should provide the complete type specification for the function. For example, to call our max template function with floating-point types we should invoke the function using

```
float max_val {max<float>(a, b)};
```

where a and b are of type **float** and have been initialized beforehand. However, modern C++ compliers can deduce the type specification from the arguments supplied such that we do not need to supply the type in the angled brackets:

```
std::string max_val {max<>(a, b)};
```

If a and b are *std::string* type, then the call instantiates the template function with *std::string* as the filler. In this case, if the

arguments are of any other type then, unless there exists a valid, accessible conversion of that type to a *std::string*, the compiler will complain. We can also omit the angle brackets entirely:

```
auto max_val {max(a, b)};
```

This allows overload resolution to examine both the template and normal function overloads if they exist. In C++11 the **auto** keyword can be used where the type can be deduced from context; here `max_val` will be initialized as the same type as `a` and `b`. Take note that due to the way we have written the `max` template function you cannot have arguments of different types.

The template parameter list delimited by the angled brackets can contain as many items as necessary, with each item separated by a comma. Each item can be a general class (using the **typename** or **class** keyword), a specific instance of a class, or a fundamental data type (or their alias). A concrete value of each template type must be provided at compile time. Note that the template parameters may also be given default argument types.

By the way, there is no need to write your own "max" template function as one is already implemented in the standard library in the *algorithm* header under the *std* namespace. Notice that it accepts arguments by **const** reference. Generally, when writing template functions, you will want the template arguments passed by **const** reference as the type is any potentially definable type and as such could be prohibitive to copy.

Note that the **typename** keyword in the template parameter list can be replaced with the **class** keyword; they are synonyms in this context. Using **typename** here rather than **class** is arguably better as it more precisely describes the functionality; class implies we can only use class names whereas we can use any defined or definable type.

Inline Functions

C++ supports simple macro **inline** expansion of functions. Any function *declared* as **inline** is not actually a function but a description of the code to be inserted at the point of the function

call. Inline functions are semantically the same as normal functions and are used as an attempt to improve program speed by avoiding the overheads of a function call and its associated stack adjustment. Note that the keyword **inline** is used only where the function is *declared* not where it is defined unless declaring and defining a function in the same place.

Inline functions generate more code than a normal function call. If an inline function itself calls another inline function that may call another inline function and so on, one line of code can expand into tens or even hundreds of lines of actual code. This is called bloat and is avoided by not using inline functions or restricting their use to simple functions, ideally ones that do not call other functions. If in doubt do not use inline functions. If the final program does not perform adequately then analyze the code to uncover any bottlenecks and redesign the program if possible. In many cases, well-chosen algorithms and good program design are more effective at speeding up programs than inline functions.

Note that with modern C++ the keyword `inline` is a suggestion to the complier that the function is made inline. It is not guaranteed that the function will be inline. The complier will make the final decision based on the code length and complexity of the function, among other considerations.

Pointers

Pointers play an essential role in the construction of any substantial C++ program. A pointer is a variable that holds the address in memory of another variable. When we declare a variable, we give it a memory address that uniquely identifies it within the memory structure. When we initialize or assign a value to a variable, we place that value at the location of the variable's unique address. When we use the variable, we are directly accessing the value stored at the address of the variable.

If we declare a pointer to the address of that variable, we then have a means of indirectly accessing the value stored at the address; we do not have to refer to the original variable to use or modify it. As an analogy, if you walk into a library and ask the Librarian "who was the King of England in 1137?" and they say "Stephen" then you are

directly accessing that data. However, if the Librarian gives you the location of a book about the monarchs of Great Britain then they are pointing you to where you can find that data. The location itself does not contain the answer you want but what is at the location does. Pointers become more beneficial when we start to consider object-orientated programming, but for now, we shall look at how they are implemented at a low level.

A pointer variable is declared as a pointer to type. A pointer can be initialized by or assigned from the address of any variable of the declared type or from another pointer of the same type. To declare a pointer to type we use the asterisk symbol, `*,` after the type name. For example,

```
int * ptr_int;
```

declares a pointer to an integer variable.

The space around the * is unimportant to the complier but is potentially important to the reader. Although it seems a petty, where to place the white space around the pointer operator is a contentious issue among programmers. Does the symbol belong to the type declaration or does it belong to the variable identifier? We will opt for treating the pointer symbol like a binary operator, whereby the type and the identifier are the "operands."

One pitfall to watch out for is declaring multiple pointers in a single statement. You might think that

```
int * first, second;
```

declares two pointers to integers called first and second. However, this is equivalent to declaring a pointer to an integer called `first`, and then declaring an integer variable called `second`. To declare `second` as a pointer to an integer on the same line as `first` you would need to prefix it with the asterisk. Hence, it is recommended to declare only a single pointer per line to avoid any confusion. Indeed, it is good practice to declare every variable on its own line.

To manipulate pointers and their contents we have the following operators `&` and * (the asterisk has different behavior here

to that just described above). The & operator when used in front of a variable identifier gives the address in memory of that variable, hence it is referred to as the address operator. The * operator when used in front of a pointer variable identifier returns the contents of the address; this is known as indirection or dereferencing. The * operator in this context is referred to as the dereference operator or the contents operator. If you compile and run the following example program the output in both cases is "a = 3 b = 7."

```
#include <iostream>

int main()
{
    //initialise a and b to zero
    int a {0};
    int b {0};

    //initialise pointers to a and b (address operator)
    int * ptr_a {&a};
    int * ptr_b {&b};

    //assign values to a and b via their pointers
    *ptr_a = 3;
    *ptr_b = 7;

    std::cout << "Direct access:\n";
    std::cout << " a = " << a << "\n";
    std::cout << " b = " << b << "\n";

    std::cout << "Indirect access:\n";
    std::cout << " a = " << *ptr_a << "\n";
    std::cout << " b = " << *ptr_b << "\n";

    return 0;
}
```

Because pointers in C++ are strongly typed (that is they must be declared as pointing to a particular type), the complier and user can tell what type is being pointed at and hence what will be dereferenced. This contrasts with other languages that have un-typed or plain pointers that could point to anything and thus require more careful use. C/C++ does allow you to declare a pointer to **void** type, which is mainly used to allow the passing of arbitrarily sized data to functions (this is an advanced topic and can be avoided using object-orientated methods).

In our code example, we have initialized the pointer when it was declared. This is good programming practice as any junk value given otherwise could be interpreted as a valid address anywhere in memory. This may lead to unforeseen consequences for your program if an attempt were made to dereference it.

In modern C++, you can initialize a null pointer with the **nullptr** type; this implies it points to nothing of relevance. However, be aware that a null pointer cannot be dereferenced, and attempting to do so will cause undefined behavior.

As a brief aside we mention basic or built-in arrays in C/C++ as they are relevant to the discussion on pointers. In C/C++, a closed pair of square braces following a variable name declaration identifies that variable as being an array of the type declared. For example,

```
int an_array[] = {2,4,8,16};
```

initializes an_array with 4 elements of the values specified. We can dereference an_array using the array access operator, [], with the index of the element we want to access placed in the brackets (indexing starts from zero).

```
int first_element {an_array[0]}; // == 2 in our
   example
```

The variable identifier an_array is a pointer to the first element in the array, such that the above statement is equivalent to:

```
int first_element {*an_array};
```

It is important to know here that the elements of an array in C/C++ are stored in contiguous memory locations, that is, in an unbroken block of memory, such that we can access subsequent elements using pointer arithmetic. To illustrate, both the following statements result in the variable being initialized with the value 8:

```
int third_element {an_array[2]};
int third_element {*(an_array + 2)};
```

Hopefully, you agree that the array access operator is a more elegant means of dereferencing an array. It is of note that when an array argument is passed to a function, it "decays" into a pointer to the type stored in the array.

Object-Orientated Programming (OOP)

Object-orientation bases its software model on behavioral, self-contained constructs. Typically, these constructs have some correspondence with real-world objects that often provide a logical starting point to object-orientated development. Traditionally, structured programming methods have separated the data from the functional code that operates on it. The object-oriented approach is to group the data and functions within a single unit called an object. The class of an object is effectively its type: it is the description of what operations are available on objects of that type, how these methods are implemented, and how the internal state of objects is represented.

Encapsulation, **inheritance**, and **polymorphism** are often called the "Big Three" of object orientation. Encapsulation literally means "to put something within a capsule," suggesting the close binding between data and the functions that act upon it. Information hiding, or the masking of internal representation, is another important feature of encapsulation. It allows the representation of a concept to be modified without affecting the interface that uses it. Inheritance defines new classes of objects in terms of extending existing classes. It corresponds to an "is a" kind of relationship, rather than "has a" kind of relationship. Polymorphism allows inheriting classes to specialize in the behavior they have inherited. These concepts are discussed in greater detail in due course.

C++ as an Object-Orientated Language

C++ is an object-orientated language. But what do we mean by object? An object is something tangible, something that can be seen or touched or felt, or something that can be alluded to, conceptualized, thought about. Objects can be

- Physical real-world concepts such as people, ATMs, smart phones, vehicles, etc.
- More abstract concepts we find in the real world such as bank accounts, dates, derivatives, laws, etc.;
- Interactive software abstractions such as windows, buttons, menus, etc.;
- Programming language constructs such as strings, arrays, I/O streams, etc.

Many objects in the world can be grouped together by their properties and behaviors; we say they have a class. This class system typically has a hierarchy. For example, the Samsung Galaxy and the iPhone are both a class of smart phone. Smart phones belong to a larger class of mobile phones, which in turn belongs to an even larger class of communication technology. As another example, a Seat Ibiza and a Citroen C3 are both classed as Hatchbacks. Hatchbacks are cars, and cars are vehicles. As we go up the hierarchy the concept of the class becomes increasingly abstract. A vehicle, in general, gets you from A to B but the method by which it gets you there can vary wildly depending on the vehicle's specific class; compare cars, ships, and planes, for example.

In C++, an object is a runtime instance of a class. There can be many objects of a single class and many classes within a program. But how does the language provide the class feature? The **`class`** keyword can be used to define a new data type, providing it with a name, a set of operations expressed as member functions, and an internal representation expressed as member data. We call this encapsulation. A class is typically defined in a header file that constitutes a kind of contract with the user; it tells us what the class is and how we can operate with objects of that class. As an example, *std::cout* is an object of the output stream class *ostream* defined in

the *iostream* header file. We know we can use the operator >> with it to stream an output sequence to standard output, typically the terminal, but it has many more member functions that can be used to affect its behavior. And that is the point of using objects within software; interacting with them is completely defined by their behavior. The internal state representation of an object must exist but is not a consideration when interacting with the object. In other words, objects are what we can do with them rather than the parts that make them. As simple examples of this in the real-world think about pens and cars as objects. A pen has a "write" function. You as the user simply pick up the pen and write with it with no thought about how the pen implements putting ink on the paper. Similarly, a car has a "travel" function that gets you from A to B. Unless you are a car mechanic or mechanical engineer you likely have little idea of the car's internal workings.

We call the pen-write "function" or the car-travel "function" their interface or how we interact with them. The details of how those functions work are called the implementation. When writing applications, we are typically only interested in the interface of objects; usually referred to as the Application Programming Interface (API). However, if writing libraries for future and/or widespread use we will be concerned with the implementation details: algorithm correctness; computational efficiency; speed; memory optimization; storage concerns, and so on (as well as providing a clean and simple interface for application programmers).

Simply put a class type can be split into two parts: the class definition serving as a contract to the user and the class implementation containing its functionality.

Class Definition

The class definition is placed in a header file (*.h* or *.hpp*) and contains the class identifier (or type name) and its members that consist of data and function declarations, including constructors.

```
class SomeClass {
private:
          //data members, internal representation
```

```
public:
          //the class constructor(s), initialisation
public:
          //member functions, class interaction or
             interface
};
```

The header file name is not required to match the class definition it contains (indeed a header file may contain several, different class definitions) but it is considered good practice to have a coherent and consistent naming policy (and to have one class definition per header where appropriate).

The **`private`** section declares the data members that make up the internal representation of the class. Although data members can be declared **`public`** this goes against the idea of encapsulation and an object being responsible for its internal state. The **`public`** section(s) declares the constructors and member functions that can be used by client code; member functions can be declared **`private`** but are then only accessible through other member functions of the specific class. As an aside, member functions can generally be split into two types: query functions and modifier functions. Query functions simply return the current internal representation of an object for inspection. Modifier functions change the internal representation of an object in some way.

The specifiers **`public`** and **`private`** may be repeated in the class definition, the order in which these sections occur is unimportant (unimportant to the compiler but potentially important to a person). It is my preference to put the representation details (private section) before the class user interface (public section) but this is not set in stone. So long as it follows a logical and readable format then the choice is yours. However, it pays to be consistent so once you have a preferred style you should stick to it. Please note that the default access to members of a class that is, if you do not specify one explicitly, is **`private`**. A third access specifier exists called **`protected`**. This is typically used in conjuncture with the inheritance feature of OOP in C++ and will be discussed shortly.

When an object is initialized from a (poorly programmed) user-defined class its initial state may be undefined. If the object is used later, then it may give undefined or meaningless behavior; the class is said to be not fully encapsulated. A constructor is a special member function that initializes an object automatically and, if professionally written, fully. Constructors have the same name as the class and no return type, not even void. You can think of the constructor as the initialization sequence for an object and we can provide different ways to initialize an object by overloading the constructor for a class. A constructor typically takes arguments that are used to initialize the classes' data members. The default constructor will either take no arguments or have all arguments with default values, the overloaded variants will perhaps only take certain arguments. Take note that if you do not provide any constructors for a class the compiler will automatically generate a set for you. More on this later but of importance here is the automatically generated destructor. The destructor is a special member function that is called whenever an object of a class goes out-of-scope and serves to clean up the object fully. Typically, the automatically generated destructor is fit for purpose, and you rarely have to provide one explicitly. We will discuss destructors in more detail in the sections on inheritance and polymorphism.

Member functions are declared in the same way as normal functions but with some optional qualifiers. To illustrate let us create an example C++ class definition for the date and time:

```
class DateTime {
private: //data members
   double _posix_time;
public: //constructors
   DateTime();
   DateTime(double posix_time);
   DateTime(std::string date_time);
   DateTime(int year, int day_in_year);
public: //member functions
   std::string dateTimeString() const noexcept;
   void incrementTime(double amount = 1.0) noexcept;
};
```

The date and time can be represented by a single floating-point number that tells us the number of seconds that have passed since a well-defined point in time. POSIX time represents the number of seconds that have passed since The (Unix) Epoch, namely midnight on January 1, 1970. This is the internal representation of our `DateTime` class, or at least that is the intention. The interface also tells us there are four ways to initialize a `DateTime` object. The default constructor (no parameters), a constructor that accepts an argument of **`double`** type, another that accepts an argument of *std::string* type, and a final constructor that accepts two **`int`** type arguments. How these constructors are implemented is not yet relevant to the discussion. Finally, the interface tells us what we can do with objects of our `DateTime` class, the member function declarations. First a member function, `dateTimeString`, accepts no arguments and returns a *std::string* object, presumably a string representation of the `_posix_time` data member but this is an implementation detail. And a second member function, `incrementTime`, can accept a **`double`** type argument that presumably increases the internal representation of the POSIX time by the value of the argument, again an implementation detail. Notice that a class definition is a statement so must be terminated by a semicolon.

The qualifying keywords **`const`** and **`noexcept`** that follow the member function declarations in our `DateTime` class inform the user (and the compiler) about the implementation of those functions in general. First, **`const`** tells the user that the member function does not modify the internal representation of the object, the data members are unaffected by a call to this function. Typically, **`const`** member functions are query functions. Objects and references to objects declared **`const`** can only call **`const`** member functions. This is logical. Any member function not declared **`const`** is considered a modifier function and a **`const`** object cannot be changed; its internal representation must remain constant. An attempt to call a non-**`const`** member function on an object or reference to an object declared **`const`** will result in a complier error. It is more than a programming convention as it is both useful to the complier that checks the code and the human programmer who uses the code. Appropriate use of **`const`** is a good habit to acquire. Second, **`noexcept`** is used to tell the user that the member function is guaranteed not to

throw an exception during its operation. The compiler can also use this information to apply certain optimizations when a **noexcept** member function is called.

A member function may be overloaded on **const**. That is, you can have two member functions that have identical prototypes except for the **const** qualifier. The **const** version will be called for objects declared **const** and the other version for modifiable objects. An example of this can be seen in the std::vector class where the array access operator (`operator[]`) is overloaded on the **const** qualifier to provide read and write versions of that operator.

Class Implementation

The class implementation is placed in a source file (*.cpp*, *.cxx*) that includes the class definition file (*.h*, *.hpp*). The naming of this source file is flexible, but it is logical to match the name of the header file containing the class definition. The implementation file contains the constructor definitions and member function definitions that will be compiled into machine code. For a client to use the class they must include the class definition header file in their code and link in the complied implementation code.

Where the class definition provides an interface that a class user can include and use, the class implementer plays a different role, that of the supplier of functionality. In addition to defining the bodies of the member functions declared for use on objects of a class, the class implementer must also be aware of object identity and initialization issues.

Let us return to our `DateTime` example class to see how we write these definitions in practice:

```
//constructors
DateTime::DateTime() : _posix_time(0.0) {}
DateTime::DateTime(double posix_time) : _posix_
          time(posix_time) {}
DateTime::DateTime(std::string date_time) : _posix_
          time(0.0) {
   //code to convert date_time to _posix_time
}
```

```
DateTime::DateTime(int year, int day_in_year) : _
          posix_time(0.0) {
   //code to convert year and day_in_year to _posix_
     time
}
//member functions
std::string DateTime::dateTimeString() const {
   //code to convert _posix_time to string and
     return result
}
void DateTime::incrementTime(double amount) {
   //code to increment _posix_time by amount
}
```

Constructor definitions are written using the class scope operator, which is the class name followed by a double colon, followed by the constructor "signature," which consists of the class name and parameter list in parentheses. The member initializer list provides an elegant means for passing initial values to data members; the list starts with a colon (referred to as the delegation operator) and is a comma separated for multiple data members. The members are initialized with the values in parentheses. For multiple data members, regardless of the order of the members in the list, initialization occurs in the order they were declared in the class definition. In our default constructor for the `DateTime` class, the member initialization list simply initializes the data member, `_posix_time`, to zero. The constructor is completed by providing the constructor body, delimited by curly braces. Quite often the body will contain no code because the object can be completely initialized by the member initialization list. It is important to know that an object is not actually created until the end of the constructor body is reached. This is a language feature and avoids partial objects from existing; either the object is created, or it is not. It does not guarantee that a partially initialized object could exist, it is up to the person who wrote the class implementation to completely initialize an object. When data members cannot be directly initialized in the member initialization list, we must use the constructor body. In this case, it is considered good practice to initialize the data member with some "default" value in

the initialization list. Technically, the data member is not initialized in the constructor body but assigned some value. We generally do not have to be concerned with this distinction, but it is something to be aware of. For example, a member initializer must be present for any data members that are declared `const`, or that cannot be reassigned at a later point, as this is the only opportunity to initialize them. If the code in the constructor body throws an exception or exits abnormally in any way, the object is not created; running code must reach the closing curly brace and return to the calling environment for an object to be created.

Member function definitions are like normal (global) function definitions, with the addition of class scope and the ability to access other class members. The class scope resolution operator must be used for member function definitions otherwise the compiler assumes the function is global (normal). If declaring a `const` member function in the class definition, then the `const` keyword must also be included in the function definition after the parameter list as this forms part of the function signature. The `noexcept` qualifier is not required in the function definition.

Automatically Generated Constructors and Operators

In modern C++, the compiler automatically generates the default constructor, copy constructor, copy assignment operator, move constructor, move assignment operator, and destructor for a type if they have not been explicitly defined by the programmer. These functions are known as the *special member functions*.

Automatic generation of special member functions is convenient for simple classes, but complex classes often define one or more of these functions, and this can prevent other special member functions from being automatically generated. There are several rules to bear in mind if explicitly providing definitions of the special member functions:

- The default constructor is auto-generated if there is no user-declared constructor (of any kind).
- The copy constructor is auto-generated if there is no user-declared move constructor or move assignment operator.

- The copy assignment operator is auto-generated if there is no user-declared move constructor or move assignment operator.
- The destructor is auto-generated if there is no user-declared destructor.
- The move constructor is auto-generated if there is no user-declared copy constructor, copy assignment operator or destructor, and if the generated move constructor is valid.
- The move assignment operator is auto-generated if there is no user-declared copy constructor, copy assignment operator or destructor, and if the generated move assignment operator is valid.

As a rule of thumb if you explicitly *declare* any of these special member functions, save perhaps the default constructor, then you should explicitly *declare* the others. Note also that we emphasize the word *declare*. In general, you probably will not have to *define* these functions because C++11 (and upwards) specifies the use of the keywords **`default`** and **`delete`** in conjunction with these special member functions (and indeed any other member functions).

As ever an example of this feature is illustrative. Here we just show you the syntax for the various functions we have not discussed.

```
class Demo {
private:
   //internal representation
public:
   ~Demo(); //destructor
   Demo() = default; //default constructor
   Demo(Demo&) = delete; //copy constructor
   Demo& operator=(Demo&) = delete; //copy assignment
   Demo(Demo&&) = default; //move constructor
   Demo& operator=(Demo&&) = default; //move
      assignment
public:
   //member functions
};
```

Here we imagine that the Demo class requires an explicit destructor to be declared and defined because it requires additional functionality that the auto-generated destructor cannot provide. In this case, we would not get an auto-generated default constructor because the compiler assumes that if the auto-generated destructor is not adequate then neither is the auto-generated default constructor. However, we know that in fact the auto-generated default constructor is fine for this class so we can reinstate it using the **`default`** keyword. For whatever reason, we want to prevent objects of this class from being copied. With only a destructor and default constructor declared we would still get the auto-generated copy special members. We must remove them by explicitly declaring them with the **`delete`** keyword. For yet another reason we would like objects of this class to be moveable. As we have declared an explicit destructor the auto-generated move functions are no longer supplied. However, just like the default constructor, we are happy that the auto-generated move functions are adequate, so we reinstate them using the **`default`** keyword.

Using Objects

The simplest way to create an object is to declare it as a variable. In this case, the type of the variable is the class name and the variable itself is the object. Remember to include the relevant header file that contains the class definition in your code. We say that an object is an instance of its class.

```
DateTime posix_epoch; //uses default constructor
std::cout << "The Epoch: " << posix_epoch.date
   TimeString();
```

Once created an object of a class may be manipulated via its public interface, with public member functions being accessed using the dot operator. When an object is used indirectly via a pointer the member access operator, `operator->`, must be used instead.

```
void someFunction(DateTime * date_time_ptr) {
   Date_time_ptr->incrementTime(5.0);
}
```

Behavior, state, and identity are three defining aspects of an object. The first two have been covered in our discussion of member functions (behavior) and data members (state). The last express the idea that one object is distinct from another. The simplest way to handle object identity in C++ is by the (memory) address of the object. This will be a unique number for the object and distinct from other objects. This is how we can get a handle on the object's identity outside of the object itself but what about inside the object? What happens when the program is executing a member function? To answer these questions, we note that each member function has a pointer parameter that is implicitly present. The **`this`** pointer points to the object on which the member function is being called. When a member function is called on an object, the address of that object is automatically passed to the function. This can be thought of as a hidden or silent first argument to the member function. The **`this`** pointer is not normally used by the programmer but can be referred to explicitly when needed. In essence, an object is aware of its own address in memory through the **`this`** pointer.

Inheritance

We have discussed previously that objects in the real world can be grouped together by their properties and behaviors, and those classification systems have a hierarchy. Typically, the class at the top of the hierarchy is the most abstract of the system, and the class at the bottom is the most specific. For instance, the animal kingdom is a hierarchical class system. The term animal is the most abstract concept. Based on certain properties and behaviors animals can be categorized into more specific groups: mammals, fish, birds, reptiles, amphibians, and insects. Those groups can be further separated into even more specific groups, for example, dogs, cats, cows, dolphins, bats, humans all fall under the category of mammal. And again, we can split those into more specific types; dogs can be Terriers, or Spaniels, or Retrievers, or Labradors, and the list goes on. The point here is the hierarchy tree: A Terrier is a dog, a dog is a mammal, and a mammal is an animal.

Somebody once said that abstraction is like selective ignorance; you discard the specifics and focus on generalities. Abstraction is a

good thing as it allows a look at the bigger picture without getting bogged down in the details. The hard part is implementing abstraction while allowing lower levels to still be efficient and accessible where necessary.

C++ attempts to reflect the ideas of class hierarchy, and abstraction, through the concepts of inheritance and polymorphism. A base class is (typically) the highest, and therefore the most abstract class. We derive sub-classes from base classes that fill in the specifics of what that derived class is meant to do. Typically, we refer to this structure as base class and derived class, but it can also be known as super-class and sub-class. C++ supports multiple inheritances, that is, having more than one base class per derived class, as well as multiple levels of inheritance; a derived class can be a base class, the derived class of which can be another base class, and so on. If we go back to our example of the animal class system and we could have categorized animals based on their diet, carnivores only eat meat, herbivores only eat plants, and omnivores eat anything. A Terrier for example is a dog, which is a mammal, but it is also a carnivore.

As a demonstrative example in C++ let us think about two-dimensional shapes. In general, a 2D shape has a width, and a height. They also have an area and a perimeter. It has these properties regardless of the details of the shape; the shape could be a polygon, circle, or random squiggle. Though a random squiggle cannot be defined in terms of its width and height alone, they would probably define the "bounding-box" of the squiggle. This gives us a starting point to define a shape base class in C++ based on its mathematical properties:

```
//Shape2D definition – e.g. in shape_2d.h
class Shape2D {
protected:
    double _width;
    double _height;
    std::string _name;
public:
    Shape2D(double w, double h, std::string name =
        "Rectangle");
```

```
public:
   double area() const;
   double perimeter() const;
};
//Shape2D implementation - e.g. in shape_2d.cpp
Shape2D::Shape2D(double width, double height,
         std::string name):
   _width(width), _height(height), _name(name) {}

double Shape2D::area() const {
   return _width * _height;
}

double Shape2D::perimeter() const {
   return 2 * (_width + _height);
}
```

The `Shape2D` class requires two **`double`** data members, one for width, and the other for height. For convenience, we also include a *std::string* data member to record the name of the shape. We also give it two member functions, one to compute the area of the shape, and the other to compute its perimeter. We assume the shape base class takes the "simplest" form, that of a rectangle. Its area is then computed by the product of its width and height, and its perimeter by twice the sum of those values.

To declare a class as being derived from another we use a single colon after the derived class' identifier followed by an access qualifier: **`public`**; **`private`**; or **`protected`**, and the base class we wish to use. We can declare multiple base classes using a comma-separated list; each base class requires its own access qualifier. A derived class will inherit any members of the base class that are not specified as being **`private,`** that is, a derived class does not have (direct) access to its base class **`private`** members (access could be indirect via **`public`** or **`protected`** member functions in the base class if they exist). When we inherit from a class via **`public`** access anything that is **`public`** in the base class becomes **`public`** in the derived class and anything that is **`protected`** in the base class becomes **`protected`** in the derived class. When we inherit from a base class via **`private`**

access, both **public** and **protected** members of the base class become **private** members of the derived class. And when deriving a class via **protected** access, **public** and **protected** members of the base class become **protected** in the derived class. In the majority of use cases, **public** inheritance is the appropriate method to employ.

Getting back to our example, with the Shape2D base class defined we can then design derived classes for more specific shapes. The square, for example, is the case where the height and width of the base shape are equal.

```
//Square definition
class Square : public Shape2D {
public:
   Square(double side);
};
//Square Implementation
Square::Square(double s) : Shape2D(s, s, "Square") {}
```

We declare the Square class as having **public** inheritance from the Shape2D class. The constructor for the Square takes just one argument that specifies the length of a side. The Square class does not need to declare the data members that represent width and height as it inherits these from the Shape2D base class. The Square class also inherits the base class functions that compute the area and perimeter of the shape. It does not need to redefine these functions as they already compute the correct values. When we implement the constructor for the Square class, we use the member initialization list syntax but the first "member" that we initialize is the base class itself. We do this by calling the base class constructor with the required arguments. Professionally written C++ code will always construct the base class first before initializing any data members of the derived class. This reflects the philosophy (or is it practicality) that we build a house up from the foundations, rather than down from the roof. Note that the (default) destructor will then "tear down" the derived class from the roof down to its foundations. It will clean up any data members in the derived class before calling the destructor for the base class. You could of course provide your

own definition for the destructor that changes this order, but this is not recommended.

We can also derive a circle class from the shape base class. Like the square class, we would like a constructor that takes one argument, but in this case, the parameter specifies the radius of the circle. We must also provide new declarations and definitions for the functions that compute the area and perimeter, as these are now different from the base class.

```
//Circle definition
class Circle : public Shape2D{
public:
   Circle(double radius);
public:
   double area() const;
   double perimeter() const;
};
//Circle implementation - width and height are diameter
circle::circle( double r ) : shape(2*r, 2*r, "Circle") {}

double circle::area() const {
   return PI * _width * _width / 4.;
}

double circle::perimeter() const {
   return PI * _width;
}
```

By specifying new area and perimeter functions for the `Circle` class we hide or shadow those inherited from the base class. When an object of `Circle` type calls a member function, the compiler checks the most local namespace first, that is, the `Circle` class name, for a match and uses the function found there. Whereas, with the `Square` class the compiler will not find a match within the local class namespace and so searches the base class `Shape2D` for that member function. If a match cannot be found in the base class, this is then flagged as an error by the compiler.

A triangle seems such a simple shape. It consists of three straight lines that connect at three vertices. But we have a problem. Our general shape class has a width and a height that defines a rectangular border. Even with the constraint that the triangle's base edge runs along the bottom border of the rectangle, there remains an infinite number of triangles that will fit this rectangle. While this does not affect the computation for the area of the triangle, which is always one-half base times height, it does affect the calculation for the perimeter. How might you go about designing a `Triangle` class (or classes) using the `Shape2D` class as a base?

Of course, our initial design for an abstract base shape class may not be optimal. Readers that have looked at graphical programming (with OpenGL or DirectX, for example) will know that a shape can be described as a collection of vertices, points in either a two- or three-dimensional space. These vertices are implicitly connected by straight lines and three or more vertices connect to form a loop that describes a face of the shape. For example, a cube is a collection of eight vertices that are connected to form six square faces. In this way, a shape is an array of vertices or points with a corresponding description of how these vertices are joined together to form the shape.

As a final thought on inheritance, see it as a "is a" relationship rather than a "has a" relationship; a derived class type is its base class type. Also, consider whether the "is a" relationship is appropriate. For example, a car *is a* vehicle, but it *has* various components, wheels, an engine, a chassis, seats, and so on. Let us say we had a class representing an engine, it most certainly should not be a base class for a car class and equally not be derived from a car class. A car is not an engine, an engine is not a car, rather a car has an engine. While this relationship is obvious in this contrived example it might be more subtle when designing more abstract concepts in code.

Polymorphism

It essentially means that you can change the behavior of a class via the same pointer or reference to its base type. This allows us to set up standard containers (vectors, lists, etc.) with pointers (or references) to the same (base) type, but where each element may have different behavior depending on the complete derived type to

which they point. It also allows us to write general functions that take base class pointers or references that change behavior depending on the complete derived class to which they point or refer. Without polymorphism, we would have to provide overloaded variants of the same function for each derived class. With polymorphism, we can provide a single function that takes a pointer or reference to the base class and that works for all derived classes.

To provide polymorphism in our classes C++ uses the keyword **`virtual`**. In our inheritance example, the base `Shape2D` class provides an area and perimeter function that is inherited by the derived classes. Typically, we shadow these base class functions by declaring the same functions within the derived class. When we instantiate a direct object of a derived class and use the member functions from this object, it is those functions that are declared in the derived class that is called. However, if we have indirect access to an object via a pointer or reference to its base class, it is the base class versions of the member functions that are called.

To make the `Shape2D` class polymorphic we should declare the member functions in the base class virtual.

```
//Shape2D definition - e.g. in shape_2d.h
class Shape2D {
protected:
   double _width;
   double _height;
   std::string _name;
public:
   Shape2D(double w, double h, std::string name
      = "Rectangle");
public:
   virtual double area() const;
   virtual double perimeter() const;
};
```

This is all that is required to make the `Shape2D` base class polymorphic. Derived classes that provide their own definitions of the base class member functions need to add the **`override`** attribute at the end of the member function declaration in the class definition.

```
//Circle definition
class Circle : public Shape2D{
public:
    Circle(double radius);
public:
    double area() const override;
    double perimeter() const override;
};
```

The **override** keyword tells both the compiler and the human programmer that the derived class is providing a different definition for the member function, and it is this definition that should be used when indirectly accessing an object via a pointer or reference to its base. Note that the **override** keyword is not required when writing the member function implementation.

Sometimes we may want to create a base class that should not have any concrete objects of the base class constructed. For example, we may want to make the shape class from our discussion above an abstract base class because the concept of a shape is abstract.

To give another example let us say we are trying to build a role-playing game. One part of the game is that the player character can pick up, use, and discard items within the game world. The items could be weapons, armor, potions, food, trinkets, or any other item we would like. The hero should be able to store these items in a backpack, a literal container. Clearly, we need a base class to represent the items in the game world, but the concept of an item is an abstract one. We certainly would not want to create an item object directly; what would that mean?

To make an abstract base class in C++ we provide the **class** with what is called a pure virtual function. A pure virtual function has no definition within the base class. The syntax to set up a pure virtual function is

```
virtual   return-type   identifier(argument-list)
    [attributes] = 0;
```

The equals zero at the end of the virtual function declaration reads that this function has no implementation in the base class, and as such makes the base class abstract. You will not be able to make direct instances of this class as this function has no definition; the class is incomplete. For consistency, we should make the access qualifier for the constructor(s) of an abstract base class **`protected`**. This also has the effect of preventing direct instances of the class being created, the compiler will complain if you try, but allows derived classes (with **`public`** or **`protected`** inheritance) to use the base constructor in their initialization list. We can only refer to the base class via indirect methods, that is, pointers and references. Every concrete derived class of an abstract base class is required to declare and define an **`override`** for the pure virtual function as this then completes the class definition. If the derived class is itself abstract, either via its own pure virtual function or protected constructor it does not need to provide an override.

As a rule of thumb, a base class with at least one pure virtual member function should provide a **`virtual`** destructor. This ensures entire objects, including the abstract base class, are cleaned up when they are destroyed (go out-of-scope).

The header and source files *cpp_primer.h* and *cpp_primer.cpp*, found at the GitHub page *github.com/DJWalker42/laserRacoon*, show a practical example of how to write a class hierarchy with polymorphism in C++. These files also contain examples of many of the major features of modern C++ language, some of which we have not discussed here, including smart pointers and lambda expressions, so we urge you to study the content within. We also urge you to edit those files, play with the code, change it, and add it to so you have a thorough understanding of both the code's syntax and semantics. The source code can be compiled with:

```
g++ -std=c++11 cpp_primer.cpp -o cpp_primer.
```

INDEX

A

Adaptive quadrature
  global error, 88–89
  Lorentzian line-shape function, 87, 90
  trapezoid rule, 88
Advanced numerical quadrature
  Gauss-Laguerre quadrature, 217–219
  Gauss-Legendre quadrature
    Legendre polynomials, 213
    positive definite, 210
    programming, 214–217
    quotient polynomial, 211
    weights and abscissas, 210
  Lagrange remainder, 204
  lowest polynomials, 206–207
  mid-ordinate rule, 206
  orthogonal polynomials, 207–210
  Simpson's rule, 205
Advanced quadrature
  adaptive quadrature, 86–90
  Euler-McClaurin integration, 85–86
  multidimensional integration, 90–93
Amdahl's law, 269, 270
Application Programming Interface (API), 315

B

Backward integration, 244, 245
Bisection-Newton-Raphson, 62–64
Boole's rule, 85
Bourne-Again shell (bash), 3
Brute force search, 64–65

C

Center-of-mass (COM), 238
Central difference formula, 166
Central processing unit (CPU), 16, 255-259
Coding, 2
Command-line arguments, 10
Computers
  hardware
    CPU, 16
    input and output devices, 17
    RAM, 17
    SSD, 16, 17
  mathematics
    matrices, 27–32
    Taylor series, 25–27

number representation and precision
binary format, 20
integer values, 20
mantissa, 22, 23
pseudo code, 23–24
software
guidelines, 18–19
Linux, 18
C++ programming
automatically generated constructors and operators, 321-323
class definition, 315-319
class implementation, 319-321
coding guidelines, 284-285
command-line compilation, 279-284
description, 285
enumeration types
control flow, 291-298
exceptions, 298-301
functions, 301-305
function overloading, 305-306
inline functions, 308-309
template functions, 306-308
Object-Orientated Programming (OOP)
encapsulation, 313
inheritance, 313, 324-329
polymorphism, 313, 329-332
objects, 314, 323-324
operators, 286-290
pointers, 309-313
primitive types, 286
Crank-Nicolson method, 183–184
C shell (tcsh), 3
Cubic spline, 43–45
Cygwin, 2
Bonjour Tout Le Monde, 6

D

Data fitting
matrix form
GE, 49
LAPACK, 50
linear least squares, 48
Millikan's experiment, 50–53
regression, 45–48
Debugging, 11
DFT. *See* Discrete Fourier transform (DFT)
Differential equations, classification of
solution and initial conditions, types of, 98–99
types of
homogeneous and nonhomogeneous equation, 97
ODEs and PDE, 96
Dirichlet's theorem, 124
Discrete Fourier transform (DFT)
N data points, 127, 128
real and imaginary parts, 128

E

Earth-Moon system, 238
Euler-McClaurin integration, 85–86
Even parity wavefunctions, 246

F

Fast Fourier transform (FFT)
brief history and development, 129–130
implementation and sampling
aliasing, 134
frequency spectrum, 131
interweaving strategy, 130
leakage problem, 134

positive and negative frequencies, 131
Van der Pol oscillator, 228–230
Finite difference methods (FDM)
difference formulas
application of, 168–174
backward difference approximations, 166
central difference formula, 166, 168
differential equation, 165
five-point formula, 167
forward difference approximation, 166
multi-step formula, 167
Runge-Kutta method, 167
heat equation, dirichlet boundaries, 184–190
Finite element method (FEM), 199–200
Finite square well
regions, 69–71
Schrödinger equation, 70
wavefunction, 71, 72
Firmware. *See* Hardware
Fletcher's experiment, 51
Forward integration, 244, 245
Fourier analysis
Fourier transforms (*see* Fourier transforms)
series (*see* Fourier series)
Fourier series
coefficients of, 120
Dirichlet's theorem, 124
harmonic tone, 120
square wave, 122
Fourier transforms
components, 124
DFT (*see* Discrete Fourier transform (DFT))
FFT (*see* Fast Fourier transform (FFT))
Fourier integral, 124
momentum-space, 126
non-periodic function, 124

G

Gaussian Elimination (GE), 49
Gauss-Laguerre quadrature, 217–219
Gauss-Legendre quadrature
Legendre polynomials, 213
positive definite, 210
programming, 214–217
quotient polynomial, 211
weights and abscissas, 211
Gauss-Seidel scheme, 172, 173
Graphical processing unit (GPU), 17
Ground state function, 243
Gustafson's law, 271, 272

H

Halley's Comet, 235–237
Hard disk drive, 255, 256
Hardware
CPU, 16
input and output devices, 17
RAM, 17
SSD, 16, 17
Heat equation
Dirichlet boundaries
Crank-Nicolson method, 183–184
explicit method, 180–181
general finite difference method, 184–190
implicit method, 181–182
Neumann boundaries, 190–193
steady state heat equation, 193–196

Hello World program, 6
High performance computing
Amdahl *vs*. Gustafson, 268–272
blocking
block size, 261
matrix multiplication, 259, 260
Strassen's algorithm, 262
sub-matrix multiplications, 262
computer memory
HDD, 255, 256
RAM and CPU, 255, 256, 257
heap and stack
advantages and disadvantages, 253
LIFO, 253
segments, 252
stack frame, 253, 254
loop unrolling, 262–263
loopy indexing, 257–259
parallel programming
hyper-threading technology, 265
OMP library, 264, 265
parallel regions, 265
vector summation
dynamic assignment, 268
serial code, 266
static assignment, 268
Hybrid methods
Bisection-Newton-Raphson, 62–64
brute force search, 64–65

I

Infinite square well
box model, 66
eigenfunctions, 67
Newtonian equations, 65
Schrodinger's equation, 66
wavefunctions and probability functions, 68
Integrated development environment (IDE), 2, 3
Interpolation
cubic spline, 43–45
linear
straight line, 36, 37
symmetrical form, 37
polynomial
Lagrange formula, 40
n ordered polynomial interpolation, 40, 41
symmetrical form, 39

J

Jacobi scheme, 171, 172, 173

K

Kepler's third law, 239

L

Laplace equation, 193
Linear interpolation
straight line, 36, 37
symmetrical form, 37
Linux, 18
Lorentzian line-shape function, 87, 90

M

Mathematics
matrices
diagonal elements of, 28
Hermitian conjugate, 31
null matrix, 28
square matrix, trace of, 29
Taylor series, 25–27
Matrix form
GE, 49

LAPACK, 50
linear least squares, 48
Mid-ordinate rule, 82–83
Millikan's experiment, 50–53
Monte Carlo integration
advantage, 146
dart throwing
Pythagorean theorem, 139
random number, 139
shooting method, 137
single integration, 142
geometrical interpretation of, 143
importance sampling, 146–148
multidimensional integration, 144
trapezoidal rule, 145, 146
Monte Carlo methods
numerical integration (*see* Monte Carlo integration)
simulation (*see* Monte Carlo simulation)
Monte Carlo simulations
radioactive decay, 154–155
random walk
mean distance, drunken walk, 149, 150
spherical coordinate system, 151
stochastic process, 149
uniform distribution, 152, 153
Multidimensional integration, 90–93

N

Namespaces, 8
Newton-Raphson method, 58–60, 215, 216
Non-linear equations, 52
Null matrix, 28
Numerical quadrature
advanced quadrature
adaptive quadrature, 86–90
Euler-McClaurin integration, 85–86
multidimensional integration, 90–93
simple quadrature
mid-ordinate rule, 82–83
Simpson's rule, 84–85
trapezoidal rule, 83–84

O

Odd parity states, 242
Oil drop experiment, 50
OpenMP (OMP), 264
Ordinary differential equations (ODEs)
differential equations, classification of
solution and initial conditions, types of, 98–99
types of, 96–98
solving 2nd ordered ODES
coupled 1st order ODEs, 108–110
one dimension, 116–117
oscillatory motion, 110–116
solving 1st order ODEs
adaptive Runge-Kutta, 107–108
modified and improved Euler methods, 102–104
Runge-Kutta method, 104–107
simple Euler method, 99–102
Over-relaxation, 174

P

Partial differential equations (PDEs), 96

boundary conditions, type of
Cauchy boundary condition, 163–164
Dirichlet conditions, 163
mixed boundary conditions, 164
Neumann conditions, 163
conic sections, 160
FDM (*see* Finite difference methods (FDM))
FEM, 199–200
numerical methods
heat equation (*see* Heat equation)
wave equation, 196–199
Poisson's equation, 161
Richardson extrapolation
central difference formula, 176, 177
error behavior, 174-176
round off error issues, 178
trapezoidal rule, 177
Phase space, 225–226
Planck's constant, 73
Polynomial interpolation
Lagrange formula, 40
n ordered polynomial interpolation, 40, 41
symmetrical form, 39
Pythagorean theorem, 139

R

Random access memory (RAM), 17, 255, 256, 257
Region of interest (ROI), 64
Residual norm, 52
Richardson extrapolation
central difference formula, 176, 177
error behavior, 174-176
round off error issues, 178
trapezoidal rule, 177
Root-finding algorithm
bisection method, 56–58
even and odd parity states, 74
Newton-Raphson method, 58–60
Planck's constant, 73
Secant method, 60–62
tangent function, 74
wavefunctions and probability functions, 77
Runge-kutta-fehlberg
higher ordered method, 222
intermediary function evaluations, 223
lower ordered method, 222
Runge-Kutta method, 104–107
Runtime errors, 11

S

Schrodinger's equation, 66, 70, 245
Scope resolution operator, 8
Secant method, 60–62
Sequence acceleration method, 174
Shooting method, 137
Simple Euler method, 99–102
Simple harmonic motion (SHM), 96
Simple pendulum
finite amplitude, 231–233
utter chaos, 233–235
Simple quadrature
mid-ordinate rule, 82–83
Simpson's rule, 84–85
trapezoidal rule, 83–84
Simpson rule, 84–85, 106, 205
Software
guidelines, 18–19
Linux, 18
Solving 2nd ordered ODES
coupled 1st order ODEs
component vectors, 109

Newton's second law of motion, 109
pair of, 109
one dimension, 116–117
oscillatory motion
driving force, 115, 116
numerical solution, 112
SHM, 111, 112
steady-state region, 116
transient region, 116
Solving 1st order ODEs
adaptive Runge-Kutta, 107–108
modified and improved Euler methods, 103–105
Runge-Kutta method, 105–107
simple Euler method, 99–103
Spectral analysis. *See* Fourier analysis
Spherical coordinate system, 151
Special member functions, 321
Steady state heat equation, 193–196
Strassen's algorithm, 262
Successive Over-Relaxation (SOR), 174
Symmetrical form, 37

T

Taylor polynomial, 25
Trapezoidal rule, 83–84, 144, 146

V

Van der Pol oscillator
FFT, 228–230
non-linear differential equation, 227
phase space, 227–228