

Joshua Izaac  
Jingbo Wang

# Computational Quantum Mechanics

# **Undergraduate Lecture Notes in Physics**

Undergraduate Lecture Notes in Physics (ULNP) publishes authoritative texts covering topics throughout pure and applied physics. Each title in the series is suitable as a basis for undergraduate instruction, typically containing practice problems, worked examples, chapter summaries, and suggestions for further reading.

ULNP titles must provide at least one of the following:

- An exceptionally clear and concise treatment of a standard undergraduate subject.
- A solid undergraduate-level introduction to a graduate, advanced, or non-standard subject.
- A novel perspective or an unusual approach to teaching a subject.

ULNP especially encourages new, original, and idiosyncratic approaches to physics teaching at the undergraduate level.

The purpose of ULNP is to provide intriguing, absorbing books that will continue to be the reader's preferred reference throughout their academic career.

#### **Series editors**

Neil Ashby  
University of Colorado, Boulder, CO, USA

William Brantley  
Department of Physics, Furman University, Greenville, SC, USA

Matthew Deady  
Physics Program, Bard College, Annandale-on-Hudson, NY, USA

Michael Fowler  
Department of Physics, University of Virginia, Charlottesville, VA, USA

Morten Hjorth-Jensen  
Department of Physics, University of Oslo, Norway

More information about this series at <http://www.springer.com/series/8917>

Joshua Izaac · Jingbo Wang

# Computational Quantum Mechanics

Joshua Izaac  
Department of Physics  
The University of Western Australia  
Perth, Australia

Jingbo Wang  
Department of Physics  
The University of Western Australia  
Perth, Australia

ISBN 2192-4791

ISBN 2192-4805 (electronic)

Undergraduate Lecture Notes in Physics

ISBN 978-3-319-99929-6

ISBN 978-3-319-99930-2 (eBook)

<https://doi.org/10.1007/978-3-319-99930-2>

Library of Congress Control Number: 2018955445

© Springer Nature Switzerland AG 2018

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG  
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

# Contents

<b>I Scientific Programming: an Introduction for Physicists</b>	<b>1</b>
<b>1 Numbers and Precision</b>	<b>3</b>
1.1 Fixed-Point Representation . . . . .	4
1.2 Floating-Point Representation . . . . .	5
1.3 Floating-Point Arithmetic . . . . .	10
Exercises . . . . .	15
<b>2 Fortran</b>	<b>17</b>
2.1 Variable Declaration . . . . .	18
2.2 Operators and Control Statements . . . . .	23
2.3 String Manipulation . . . . .	29
2.4 Input and Output . . . . .	34
2.5 Intrinsic Functions . . . . .	38
2.6 Arrays . . . . .	40
2.7 Procedures: Functions and Subroutines . . . . .	49
2.8 Modules . . . . .	64
2.9 Command-Line Arguments . . . . .	73
2.10 Timing Your Code . . . . .	78
Exercises . . . . .	81
<b>3 Python</b>	<b>83</b>
3.1 Preliminaries: Tabs, Spaces, Indents, and Cases . . . . .	85
3.2 Variables, Numbers, and Precision . . . . .	88
3.3 Operators and Conditionals . . . . .	93
3.4 String Manipulation . . . . .	100
3.5 Data Structures . . . . .	106
3.6 Loops and Flow Control . . . . .	118
3.7 Input and Output . . . . .	126
3.8 Functions . . . . .	130
3.9 NumPy and Arrays . . . . .	136
3.10 Command-Line Arguments . . . . .	155
3.11 Timing Your Code . . . . .	158
Exercises . . . . .	162

<b>II Numerical Methods for Quantum Physics</b>	<b>163</b>
<b>4 Finding Roots</b>	<b>165</b>
4.1 Big-O Notation . . . . .	165
4.2 Convergence . . . . .	167
4.3 Bisection Method . . . . .	168
4.4 Newton-Raphson Method . . . . .	171
4.5 Secant Method . . . . .	173
4.6 False Position Method . . . . .	176
Exercises . . . . .	178
<b>5 Differentiation and Initial Value Problems</b>	<b>181</b>
5.1 Method of Finite Differences . . . . .	181
5.2 The Euler Method(s) . . . . .	186
5.3 Numerical Error . . . . .	196
5.4 Stability . . . . .	197
5.5 The Leap-Frog Method . . . . .	202
5.6 Round-Off Error . . . . .	205
5.7 Explicit Runge–Kutta Methods . . . . .	209
5.8 Implicit Runge–Kutta Methods . . . . .	213
5.9 RK4: The Fourth-Order Runge–Kutta Method . . . . .	218
Exercises . . . . .	224
<b>6 Numerical Integration</b>	<b>229</b>
6.1 Trapezoidal Approximation . . . . .	229
6.2 Midpoint Rule . . . . .	236
6.3 Simpson’s Rule . . . . .	238
6.4 Newton–Cotes Rules . . . . .	244
6.5 Gauss–Legendre Quadrature . . . . .	246
6.6 Other Common Gaussian Quadratures . . . . .	252
6.7 Monte Carlo Methods . . . . .	254
Exercises . . . . .	259
<b>7 The Eigenvalue Problem</b>	<b>265</b>
7.1 Eigenvalues and Eigenvectors . . . . .	265
7.2 Power Iteration . . . . .	269
7.3 Krylov Subspace Techniques . . . . .	278
7.4 Stability and the Condition Number . . . . .	284
7.5 Fortran: Using LAPACK . . . . .	286
7.6 Python: Using SciPy . . . . .	298
Exercises . . . . .	304
<b>8 The Fourier Transform</b>	<b>309</b>
8.1 Approximating the Fourier Transform . . . . .	309
8.2 Fourier Differentiation . . . . .	316

8.3	The Discrete Fourier Transform . . . . .	319
8.4	Errors: Aliasing and Leaking . . . . .	334
8.5	The Fast Fourier Transform . . . . .	337
8.6	Fortran: Using FFTW . . . . .	348
8.7	Python: Using SciPy . . . . .	351
	Exercises . . . . .	354
<b>III</b>	<b>Solving the Schrödinger Equation</b>	<b>357</b>
<b>9</b>	<b>One Dimension</b>	<b>359</b>
9.1	The Schrödinger Equation . . . . .	359
9.2	The Time-Independent Schrödinger Equation . . . . .	360
9.3	Boundary Value Problems . . . . .	361
9.4	Shooting method for the Schrödinger Equation . . . . .	368
9.5	The Numerov–Cooley Shooting Method . . . . .	377
9.6	The Direct Matrix Method . . . . .	380
	Exercises . . . . .	383
<b>10</b>	<b>Higher Dimensions and Basic Techniques</b>	<b>387</b>
10.1	The Two-Dimensional Direct Matrix Method . . . . .	387
10.2	Basis Diagonalization . . . . .	397
10.3	The Variational Principle . . . . .	402
	Exercises . . . . .	407
<b>11</b>	<b>Time Propagation</b>	<b>415</b>
11.1	The Unitary Propagation Operator . . . . .	416
11.2	Euler Methods . . . . .	418
11.3	Split Operators . . . . .	420
11.4	Direct Time Discretisation . . . . .	422
11.5	The Chebyshev Expansion . . . . .	425
11.6	The Nyquist Condition . . . . .	428
	Exercises . . . . .	435
<b>12</b>	<b>Central Potentials</b>	<b>439</b>
12.1	Spherical Coordinates . . . . .	439
12.2	The Radial Equation . . . . .	443
12.3	The Coulomb Potential . . . . .	445
12.4	The Hydrogen Atom . . . . .	447
	Exercises . . . . .	455
<b>13</b>	<b>Multi-electron Systems</b>	<b>459</b>
13.1	The Multi-electron Schrödinger Equation . . . . .	459
13.2	The Hartree Approximation . . . . .	460
13.3	The Central Field Approximation . . . . .	465

13.4 Modelling Lithium . . . . .	469
13.5 The Hartree–Fock Method . . . . .	477
13.6 Quantum Dots (and Atoms in Flatland) . . . . .	480
Exercises . . . . .	483
<b>Index</b>	<b>487</b>

# Preface

Programming is becoming an increasingly important skill in twenty-first-century life, and nowhere more so than in physics research. Problems that were previously intractable and avoided are now becoming commonplace as computational power increases. As a result, the ability to program, understand common numerical techniques, and apply these to the physical world around us lends an unparalleled advantage. Numerical skills and computational ability also allow us, more than ever, to cross fields and perform interdisciplinary studies, as techniques such as numerical differentiation are applicable across physics, mathematics, economics, biology, psychology, and many more.

Undergraduate life, however, is adapting at a much slower rate. Quantum mechanics, as commonly taught in undergraduate physics courses, focuses mostly on systems with known analytic solutions, for example, the infinite square well, the finite square well, the simple harmonic potential, and the hydrogen atom. This belies modern research, where most practical problems in quantum mechanics do not have known analytical solutions. As such, being armed solely with techniques in analytics (often supplemented by computer algebra systems such as Mathematica or Maple) often leads to being ill-equipped to study real-life quantum systems!

This textbook introduces numerical techniques required to tackle problems in quantum mechanics, alongside numerous examples taken from various quantum systems. By the end of the book, you should be confident in applying these techniques to solving the Schrödinger equation for arbitrarily complex potentials in one or higher dimensions, and to simulating quantum dynamics under the influence of time-independent or time-varying external fields. You will also be exposed to the concepts of self-consistent fields, indistinguishable particles, anti-symmetrised wave function, and the basic formalism of Hartree and Hartree–Fock theory to study multi-electron systems.

As this textbook is designed for senior undergraduate physics students in the third or fourth year of their studies, some familiarity with quantum systems is expected. However, as is often the case with physics undergraduate courses, no prior programming knowledge is assumed. We begin in Part I of the book by introducing core concepts of programming for scientific

computation. Here, we follow a dual approach, discussing and introducing two programming languages that have had (and are still having!) a massive impact on physics research — Fortran and Python. Feel free to learn either or both; throughout the book, examples are presented using both programming languages.

Fortran 90/95 is recommended for this book for the following reasons:

- Its history: designed specifically for scientific computation, Fortran is focused on array and matrix computations, and many highly optimised mathematical and scientific libraries are still written in Fortran.
- Its speed: as a result, array and matrix computations are much faster than other programming languages such as Java, MATLAB, and Mathematica, even faster than C in most situations.
- It can be easily parallelised for use in high-performance computing environments, an increasingly important resource in the twenty-first century.

While Fortran remains pervasive in scientific computation, the field is slowly moving towards more modern programming languages such as Python, which are being extensively developed with the scientific community in mind. This increase in popularity is well justified, as can be seen from the following:

- It is easy to learn: Python has a relatively simple syntax; as such, it is reasonably straightforward to read and learn.
- Its ecosystem: more and more scientific libraries are being written in Python, making for a rich ecosystem, even for niche fields. Some of these libraries are even written in Fortran, providing speeds in Python almost comparable to native Fortran.
- It makes problem-solving faster: Python does not need to be compiled and can be used in a web-based notebook interface.

Finally, both Python and Fortran are open source and freely available: this allows accessibility for all, without the licence fees required by Mathematica, Maple, or MATLAB. If you choose to learn both Python *and* Fortran, you will find yourself well prepared to enter the world of scientific computation. While not touched upon in this text, Python and Fortran work well together in practice, with Python often forming the ‘front-end’ interface to facilitate ease of use, with Fortran used in the background to perform the intensive number crunching computations.

Beyond serving as an introduction to scientific programming, this text also introduces essential computational and numerical methods in Part II — all techniques introduced are firmly grounded in the pursuit of solving the Schrödinger equation in order to study quantum systems and related phenomena. With a hands-on approach and both Fortran and Python code examples,

numerical techniques discussed include finding roots, integration, differential equations, linear algebra, and Fourier transformations. Unfortunately, no numerical technique is perfect; some are more suited to certain applications than others. This textbook also introduces concepts such as numerical error and algorithm stability, allowing you to make an informed decision on which one to choose. Throughout the book, we offer examples and questions based on problems in quantum systems, finally finishing in Part III by solving the Schrödinger equation using a purely numeric and procedural approach, with no symbolic or functional assistance. This approach provides a solid base for further studies in high-performance scientific computation with more sophisticated applications in the era of quantum information and technology.

This textbook evolved out of several lecture courses taught at the University of Western Australia, including Computational Physics (2002-2005), Computational Quantum Mechanics (2007-2013), and Scientific High Performance Computation (2015-2018). We are in debt to many students attending these courses for testing the multitude of code examples, especially Thomas Loke, Scott Berry, Michael Swaddle, Sam Marsh; to past research students in the Quantum Dynamics and Computation group for developing, evaluating, and analysing algorithms for quantum simulation, including Stuart Midgley, Peter Falloon, Shane McCarthy, Ranga Muhandiramge, Chris Hines, Kia Manouchehri, Brendan Douglas, Bahaa Raffah, Anuradha Mahasinghe, Jeremy Rodriguez, Filip Welna, Gareth Jay, Richard Green, Matthew Katz, Neil Riste, Bradley McGrath, Georgina Carson, Douglas Jacob; and in addition to all those who proofread, in particular Bruce Hartley, Jacob Ross, Tania Loke, Will McIntosh, and Nicholas Pritchard.

Our deepest gratitude goes to our families. Joshua would like to especially thank Debbie, Kath, Will, and Ash, without whom this work would not have been possible. Thank you for the laughs, many teas, and constant support. Jingbo thanks Jie, Edward, Eric, and Lin for their love and continuous support, and also all those who have been a true source of wonder and inspiration.

## How to get the most out of this text

Operating systems, software tools, and learning approaches have a lot in common — they are very personal choices, what works for one person might not work for others. Here, we provide a list of recommended software tools, as well as suggestions to get the most out of this text.

### Software tools

An essential tool for using Fortran is a compiler; this converts the Fortran source code into machine language instructions that can be executed directly on the CPU. Python, on the other hand, is an interpretive language — once a Python interpreter is installed, Python scripts can be executed directly.

- **Fortran:** the most widely used compiler for Fortran is `gfortran`. It is commonly available on most Linux distributions as part of the packaging system and is available for download for Windows and MacOS. Another common Fortran compiler is the Intel Fortran Compiler (`ifort`).
- **Python:** Python 3 is pre-installed on most Linux distributions and otherwise is easily installable via the relevant package managers. On MacOS, only the older Python 2.7 is installed by default, which is not compatible with this text; and on Windows, Python does not come pre-installed at all. In these cases, Python 3.7 can be downloaded and installed directly from the Python website. Another alternative is the Anaconda 3 Python distribution — this comes with common scientific packages such as NumPy, SciPy, and Jupyter, so is a good choice for those new to Python.
- **Text editors and IDEs:** a good text editor can make or break your programming experience; we recommend modern and extendable editors such as Sublime Text, Visual Studio Code, or Atom. Another option is an integrated development environment or IDE that includes built-in debugging and other tools to make your life easier. Choices here include PyCharm for Python, and Code::Blocks for Fortran.
- **Plotting and visualisation:** Matplotlib is the definitive plotting library for Python, and its syntax should be familiar to anyone coming from MATLAB. In fact, a majority of the figures in this book are created using matplotlib. Alternatively, you may use any plotting software you are most familiar with, for example Excel, Mathematica, MATLAB, ggplot2, or gnuplot.

In addition to the tools suggested above, another highly recommended Python tool is Jupyter. Jupyter provides a notebook interface for Python via the Web browser that is highly reminiscent of Mathematica. Using Jupyter notebooks, you can mix and match cells containing text and Python code. Furthermore, there is a Jupyter extension named Fortran-magic that allows you to add Fortran cells to the mix as well!

### Tip and tricks

In addition to the recommended software tools above, we have a couple of notes regarding the layout and how to approach this text.

- Tips, suggestions, and additional information are provided in the form of margin notes — this helps to clarify and provide additional context, without overloading the main text.
- Important equations are highlighted with a blue background; these are equations that will be repeatedly used and referenced throughout the text.

- In addition, there are several digressions labelled throughout the text.
  - Asides: these green boxes provide in-text tangential discussions that are more extensive than the margin notes.
  - Important: like the aside, but in red — these detail important subtleties that should not be skipped!
  - Derivations: marked by a blue line on the left-hand side of the page, these delineate sections containing mathematical derivations.
  - Examples: marked by a red line on the left-hand side of the page, these provide numerical and computational examples.
  - Further reading: provided at the end of each chapter, listing additional resources that may be of interest, providing more detailed coverage of topics covered within the chapter.

Finally, and most importantly, play around with the provided code, make changes, and do not be afraid to get your hands dirty! Try running the provided examples, verifying the results within the text, and have a go modifying them as you wish. Programming is a skill that is best learnt by diving straight in and coding from the get go; you'll be amazed at how quickly you begin to pick it up.

Perth, Australia

Joshua Izaac  
Jingbo Wang

Part I:

Scientific Programming: an  
Introduction for Physicists



# Chapter 1

## Numbers and Precision

You should be quite familiar with base-10 arithmetic<sup>1</sup>; digital computers, on the other hand, use base-2 or **binary** representation for arithmetic. Like base-10, binary is also written using positional notation, however restricted to only allowing the digits 0 and 1. For example, the binary number  $10110.1_2$  is converted to base-10 as follows:

$$\begin{aligned} 10110.1_2 &= 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} \\ &= 16 + 4 + 2 + 1/2 \\ &= 22.5 \end{aligned} \tag{1.1}$$

Similarly to base-10, the position of the digits relative to the decimal point determine the value of the digit — if a digit is  $n$  places to the *left* of the decimal point, it has a value of  $2^{n-1}$ , whilst if it is  $n$  places to the *right*, it has a value of  $2^{-n}$  (see Fig. 1.1).

Converting from base-10 to binary is a little more involved, but still quite elementary. For example, to convert the number 23.625 to binary, we start with the integer component 23, and continuously divide by 2, storing each remainder until we get a result of 0:

$$\begin{aligned} 23/2 &= 11 \text{ remainder } 1 \\ \Rightarrow 11/2 &= 5 \text{ remainder } 1 \\ \Rightarrow 5/2 &= 2 \text{ remainder } 1 \\ \Rightarrow 2/2 &= 1 \text{ remainder } 0 \\ \Rightarrow 1/2 &= 0 \text{ remainder } 1 \end{aligned}$$

The remainder of the first division gives us the **least significant digit**, and the remainder of the final division gives us the **most significant digit**. Therefore, reading the remainder from bottom to top gives us  $23 = 10111_2$ .

To convert the fractional part 0.625, we instead *multiply* the fractional part by 2 each time, storing each integer until we have a fractional part of

- ▶ If working outside of base-10, we use the notation  $X_B$  to denote that  $X$  is a number represented in base- $B$

$$\underline{2^{n-1}} \dots \underline{2^1} \underline{2^0} . \underline{2^{-1}} \underline{2^{-2}} \dots \underline{2^{-n}}$$

integer bits | fractional bits

**Figure 1.1** The value of binary digits in fixed point representation

<sup>1</sup>We are assuming that your physics undergraduate course was taught in base-10!

zero:

$$\begin{aligned} 0.625 \times 2 &= 1.25 \\ \Rightarrow 0.25 \times 2 &= 0.5 \\ \Rightarrow 0.5 \times 2 &= 1.0 \end{aligned}$$

The first multiplication gives us the **most significant digit**, and the final multiplication gives us the **least significant digit**. So, reading from top to bottom,  $0.625 = 0.101_2$ .

Putting these both together, we have  $23.625 = 10111.101_2$ .

## 1.1 Fixed-Point Representation

- ▶ Each **bit** stores one **binary digit** of 0 or 1 (hence the name)

As computers are limited in the amount of memory they have to store numbers, we must choose a finite set of ‘bits’ to represent our real-valued numbers, split between the integer part of the number and the fractional part of the number. This is referred to as **fixed-point representation**. For example, a 16-bit fixed-point representation might choose to use 4 bits for the integer, and 12 bits for the fractional part. Since the location of the decimal place is *fixed*, we can write out the base-10 value of this example fixed-point number convention as follows:

$$\text{base-10 value} = \sum_{i=1}^N M_i 2^{I-i} \quad (1.2)$$

where  $M$  is our binary fixed-point representation,  $M_i$  represents the  $i$ th digit,  $N$  is the total number of bits, and  $I$  the number of integer bits. In our 16-bit convention, we’d set  $N = 16$  and  $I = 4$ .

Let’s consider some real numbers in base-10, their conversion to base-2, and their resulting 16-bit fixed-point representation (using 4 bits for the integer, and 12 bits for the fractional part).

- (a)  $10 = 1010_2 \rightarrow 1010.000000000000_2$
- (b)  $-0.8125 = -0.1101_2 \rightarrow -0000.110100000000_2$
- (c)  $9.2 = 1001.0011001100110011\dots_2 \rightarrow 1001.001100110011_2$

Now, let’s convert the fixed-point representations *back* into base-10.

- (a)  $1010.000000000000_2 = 2^3 + 2^1 = 10$
- (b)  $-0000.110100000000_2 = -2^0 - 2^{-1} - 2^{-2} - 2^{-4} = -0.8125$
- (c)  $1001.001100110011_2 = 2^3 + 2^0 + 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + 2^{-12} = 9.19995$

What is happening here? In the first two examples, 10 and  $-0.8125$  were able to be represented *precisely* using the 16 bits available to us in our fixed-point representation. On the other hand, when converted to binary, 9.2 requires more than 16 bits in the fractional part to properly represent its value. In fact, in the case of 9.2, the binary decimal is non-terminating! This is similar to how  $1/3 = 0.3333\dots$  is represented in decimal form in base-10. As we can't construct a computer with infinite memory, we must truncate the binary representation of numbers to fit within our fixed-point representation. This is known as **round-off error** or **truncation error**.

## 1.2 Floating-Point Representation

In general usage, computers do not use fixed-point representation to encode real numbers. Not being able to control exactly *where* the decimal point goes in fixed-point representation is inconvenient in most cases. For example, to represent 1034832.12 in fixed point, we could reduce truncation error by having a large number of *integer* bits, whereas to represent 0.84719830 we would reduce truncation error by having a larger number of *fractional* bits.

The solution to this problem is to use **floating-point representation**, which works in a similar manner to scientific notation in ensuring that all numbers, regardless of their size, are represented using the same number of significant figures. Using the previous examples, lets write these in scientific notation in base-2, using 8 significant figures:

- (a)  $10 = 1010_2 = 1.0100000_2 \times 2^3$
- (b)  $-0.8125 = -0.1101_2 = -1.1010000_2 \times 2^{-1}$
- (c)  $9.2 = 1001.001100110011\dots_2 \approx 1.0010011_2 \times 2^3$

Unlike base-10, we use exponents of 2 in our base-2 scientific notation; this is because binary positional notation gives a digit  $n$  places left of the decimal point a value of  $2^n$ . Even more importantly, as we are restricting the number of significant figures in our base-2 scientific notation, there is *still* truncation error in our representation of 9.2.

Floating-point representation as used by digital computers is written in the following **normalised** form:

$$(-1)^S \times \left(1 + \sum_i M_i 2^{-i}\right) \times 2^E, \quad E = e - d \quad (1.3)$$

where

- $S$  is the **sign** bit:  $S = 0$  for positive numbers,  $S = 1$  for negative numbers,

► You sometimes might see  $(1 + \sum_i M_i 2^{-i})$  written as  $1.M$ , signifying the implied leading one and the fractional significant digits of the mantissa

- $M$  is the **mantissa**: bits that encode the *fractional* significant figures (the significant figures to the right of the decimal point — the leading 1 is implied),
- $E$  is the **exponent** of the number to be encoded,
- $d$  is the **offset** or **excess**: the excess allows the representation of *negative* exponents, and is determined by the floating-point standard used,
- $e$ : bits that encode the exponent after being shifted by the excess.

► For example, if we have the binary number  $1.01_2 \times 2^5$  and an excess of 16, then

- $s = 0$
- $M = 01_2$   
(drop the leading 1)
- $E = 5$
- $e = E + d = 21 = 10101_2$

► 24 bits of binary precision gives  $\log_{10}(2^{24}) \approx 7.22$  significant digits in base-10

► With only 8 bits, the largest integer we can represent in binary is  $2^8 - 1 = 255$ :

$$1111111_2$$

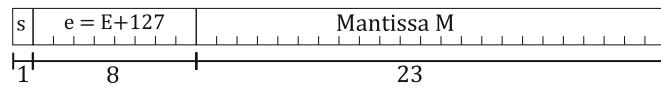
This gives a range of  $[0, 255]$ .

By introducing an **excess** of  $\frac{1}{2}2^8 - 1 = 127$ , we shift this range to  $[-127, 128]$ , allowing us to encode negative exponents

The most common standard for floating-point representation is IEEE754; in fact, this standard is so universal that it will be very unlikely for you to encounter a computer that doesn't use it (unless you happen to find yourself using a mainframe or very specialised supercomputer!).

### 1.2.1 IEEE754 32-Bit Single Precision

The IEEE754 32 bit single precision floating-point number is so-called because of the 32 bits required to implement this standard. The bits are assigned as follows: that is, 1 bit for the sign bit, 8 bits for the exponent, and 23 bits for



the mantissa. Because of the implicit leading digit in the mantissa, IEEE754 32 bit single precision therefore has 24 bits of precision, while the 8 bits in the exponent have an excess of  $d = 127$ .

To see how this works, let's have a look at a quick example.

#### Example 1.1

What is the IEEE754 single precision floating-point representation of (a)  $-0.8125$ , and (b)  $9.2$ ?

#### Solution:

(a) We know that  $-0.8125 = -0.1101_2 = -1.101_2 \times 2^{-1}$ .

Since this is a negative number, the sign bit has a value of 1,  $S = 1$ .

The exponent is  $-1$ , accounting for the excess, we therefore have  $e = -1 + 127 = 126 = 0111110_2$ .

Finally, we drop the leading 1 to form the mantissa — we can drop the leading 1. Therefore,  $M = 1010000000000000000000000_2$ .

Thus, in single-precision floating point representation,  $-0.8125$  is given by 1 0111110 10100000000000000000000000000000

(b) We know that  $9.2 = 1001.0011001100110011\dots_2 \approx 1.0010011_2 \times 2^3$ .

Since this is a positive number, the sign bit has a value of 0,  $S = 0$ .

The exponent is 3, accounting for the excess, we therefore have  $e = 3 + 127 = 130 = 10000010_2$ .

Finally, we drop the leading 1 to form the mantissa. Therefore,  $M = 00100110011001100110011_2$ .

Thus, in single-precision floating point representation, 9.2 is given by  
 $\textcolor{blue}{0 10000010 00100110011001100110011}$

### Problem question

Convert the single precision floating-point representation of 9.2, calculated above, back into decimal. What is the value *actually* stored in  
 $\textcolor{blue}{0 10000010 00100110011001100110011}$ ?

What is the percentage difference between the actual stored value and 9.2? This is the truncation error.

Before we move on, let's try encoding 0 as a single precision floating-point number. It doesn't really matter what sign we pick, so we'll let  $S = 0$ . For the exponent, we want the smallest possible value — this corresponds to  $e = 0$  or  $2^{-127}$ . So that gives us the normalised floating-point representation of

$$1.M \times 2^{-127}$$

But wait, we've forgotten about the leading 1 in the mantissa — so even if we choose  $M = 0$ , we are still left with  $1 \times 2^{-127}$ , or  $5.88 \times 10^{-39}$  in base-10. A very small number for sure, but definitely not zero!

In order to make sure that it *is* possible to encode zero in floating-point representation, IEEE754 mandates that if the exponent bits are all zero  $e = 0$ , then the implied leading 1 in the mantissa is replaced by an implied leading 0, and the represented exponent is  $E = -d + 1$ , giving  $E = -126$  in the single precision case. This is known as **denormalised form**. We can therefore use denormalised form to encode zero as a IEEE754 single precision floating point number:

$$\textcolor{blue}{0 00000000 00000000000000000000000000} \equiv (-1)^0 \times 0 \times 2^{-126} = 0$$

So, in summary:

- If  $1 \leq e \leq 2d$  (or  $-126 \leq E \leq 127$ ), we use **normalised form**, and the mantissa has an implied leading 1:

$$(-1)^S \times 1.M \times 2^{e-d}$$

- ▶ Actually, IEEE754 allows a **signed zero**; i.e. there is slightly differing behaviour for positive zero  $+0$  and negative zero  $-0$

- If  $e = 0$ , we use **denormalised form**. We set  $E = 1 - d = -126$ , and the mantissa has an implied leading 0:

$$(-1)^S \times 0.M \times 2^{1-d}$$

### Example 1.2

What are the (a) largest and (b) smallest possible **positive** single precision floating point numbers? (Ignore the case of zero.)

#### Solution:

- (a) The smallest possible positive single precision number has  $E = 0$ , and so must be *denormalised*. As it is larger than zero, it therefore has a mantissa of 1 in the least significant bit position, and an exponent of  $E = -126$ :

0 00000000 00000000000000000000000000000001

Converting this into base-10:

$$0.M \times 2^{-126} = 2^{-23} \times 2^{-126} \approx 1.40 \times 10^{-45}$$

- (b) The largest possible positive single precision number has  $E = 127$ , and so must be *normalised*. It has a mantissa of all 1s:

0 11111110 11111111111111111111111111111111

Converting this into base-10:

$$1.M \times 2^{127} = \left( 1 + \sum_{n=1}^{23} 2^{-n} \right) \times 2^{127} \approx 3.40 \times 10^{38}$$

#### Problem question

Calculate (a) the largest possible **denormalised** single precision number, and (b) the smallest possible **normalised** single precision number. What do you notice? Do they overlap, or is there a gap?

### 1.2.2 Non-numeric Values

The eagle-eyed reader might have noticed something odd in the above section.

*“Hang on, if the exponent has 8 bits, then*

$$1111111 = \sum_{i=0}^7 2^i = 255$$

*is the largest possible exponent. Subtracting the defect gives  $255 - 127 = 128$ . Then why did you say that the largest exponent in normalised form is only 127?!”*

Well, uh... you caught me. Turns out, the case where the exponent is all 1s is reserved in IEEE754 for non-numeric values.

- **Infinity:** If the exponent is all 1s, and the mantissa is all zeros, this represents positive or negative infinity, depending on the sign bit:

This behaves like you'd expect; add 1 to the largest possible floating-point number, and you'll get  $+\infty$  as a result. Adding or multiplying infinities will result in further infinities.

- **Not a Number (`NaN`):** If the exponent is all 1s, and the mantissa contains 1s, this represents NaNs:

0 11111111 0000001000000100001001 ≡ NaN

NaNs are the result of invalid operations, such as taking the square root of a negative number (IEEE754 floating-point numbers represent **real** numbers), or attempting to divide by zero.

### 1.2.3 IEEE754 64 Bit Double Precision

While single-precision floating point numbers are extremely useful in day-to-day calculation, sometimes we need more precision (that is, more significant digits — requiring a higher number of bits in the mantissa), or a larger range (more bits in the exponent). The IEEE754 64 bit double precision floating-point standard helps with both of these issues, as we now have 64 bits to work with; this is divided up to give 1 sign bit (as before), 11 exponent bits, and 52 mantissa bits. With 11 exponent bits, we can form integers from 0 to  $2^{11} - 1 = 2047$ ; thus, the excess or offset for double precision floating-point numbers is  $2^{11}/2 - 1 = 1023$ . Taking into account the denormalised case, this allows exponents to range from  $2^{-1022}$  to  $2^{1023}$ .

To see what the double precision floating-point range is, consider the largest possible positive number:

In binary, this is equivalent to

$$\left(1 + \sum_{i=1}^{52} 2^{-i}\right) 2^{1023} \approx 1.80 \times 10^{308} \quad (1.4)$$

This is  $10^{342}$  times larger than the maximum possible single precision value of only  $3.40 \times 10^{38}$ .

Furthermore, the 53 binary bits of precision (52 from the mantissa, and 1 from the implied leading digit) results in  $\log_{10} 2^{53} \approx 15.9$  significant figures in base-10, more than double the number of significant digits in the single precision case. You can see how double precision numbers can be much more useful when you need to deal with large numbers and many significant digits!

We can go even higher than double precision if we'd like; an IEEE754 standard also exists for **128 bit quadruple (quad) precision** — containing 1 sign bit, 15 exponent bits, and 112 mantissa bits, for 128 bits total. A summary of single, double, and quad precision is provided in Table 1.1.

- If the IEEE754 floating point standard has an excess of  $d$ , then the allowed exponent range is  $[1..d; d]$

Name	Sign bit	Exponent bits	Mantissa bits	Total bits	Excess	Base-10 sig. figs.
Single	1	8	23	32	127	~7.2
Double	1	11	52	64	1023	~15.9
Quad	1	15	112	128	16383	~34.0

Table 1.1 IEEE754 floating-point number standards

### Problem question

Using the identity

$$\sum_{i=1}^N 2^{-i} = 1 - 2^{-N},$$

derive expressions for the largest and smallest positive IEEE754 floating-point number with  $e$  exponent bits and  $m$  mantissa bits. Assume that when the exponent bits are all zero, the floating-point number is denormalised.

Hint: if there are  $e$  exponent bits, then the excess is given by  $d = 2^e/2 - 1$ .

- Property of commutativity:

$$x + y = y + x$$

$$x \times y = y \times x$$

- Property of associativity:

$$(x + y) + z = x + (y + z)$$

$$(x \times y) \times z = x \times (y \times z)$$

- Property of distributivity:

$$x \times (y + z) = x \times y + x \times z$$

Table 1.2 Standard properties of arithmetic

## 1.3 Floating-Point Arithmetic

When working through theoretical calculations using our standard mathematical toolset, we are used to applying the standard properties of arithmetic. For example, multiplicative and additive commutativity, associativity, and distributivity. But do these properties still apply in the case of arithmetic performed on computers using floating-point representation? Commutativity still holds, as the order we multiply or add floating-point numbers should not affect the result. But what about associativity? Let's work through a quick example to see what happens.

**Example 1.3**

Consider the expression  $-5 + 5 + 0.05$ . Using the property of associativity, we could calculate this using either

- (a)  $(-5 + 5) + 0.05$ , or
- (b)  $-5 + (5 + 0.05)$

These are equivalent if we can represent these numbers exactly, with no loss of precision.

Without evaluating (a) or (b), convert them into single precision floating-point representation. Are they still equivalent?

**Solution:** We'll start by converting the numbers into single precision floating-point representation; that is, converting them into binary with 24 significant digits.

- $5 \rightarrow 1.01000000000000000000000_2 \times 2^2$
- $0.05 \rightarrow 1.10011001100110011001101_2 \times 2^{-5}$

Using this to evaluate the two expressions (a) and (b):

- (a) For the first expression, we have  $5 - 5 = 0$  regardless of whether we use exact or floating-point representation — a number minus itself should always give zero. Therefore, after calculating the result, we are left with our floating-point number representing 0.05. Converting this back into base-10,

$$1.10011001100110011001101_2 \times 2^{-5} \rightarrow 0.050000000745058059692\dots$$

The error we see in the 10th decimal place is round-off/truncation error, due to the limited precision.

- (b) For this case, we must first add 5 and 0.05 in single precision floating-point representation. In order to add them together, we need 0.05 to have the *same exponent* in floating-point form as 5:

$$1.10011001100110011001101_2 \times 2^{-5} \rightarrow 0.0000001100110011001_2 \times 2^2$$

Note that we must truncate the rescaled value after 24 significant figures, as we only use 24 bits in single-precision. Adding 5 and then subtracting 5 results in no change. Converting to base-10,

$$0.00000011001100110011001_2 \times 2^2 \rightarrow 0.049999713897705078125\dots$$

So, in the above example, we can see that associativity *doesn't* hold for floating point numbers — by changing the order in which we add floating-point numbers together, we change the amount of truncation error that occurs! This can't be eliminated entirely; all we can do is to try and perform operations on numbers of similar magnitude first, in order to reduce truncation error.

Like associativity, the property of distributivity also does not always hold for floating-point arithmetic. In fact simple equations such as

$$0.1 + 0.1 + 0.1 = 0.3$$

are no longer perfectly guaranteed in floating-point arithmetic. To see why, let's convert 0.1 to a single-precision floating-point:

$$0.1 \rightarrow 1.10011001100110011001101_2 \times 2^{-4}$$

Converting back to base-10, we see that truncation error has occurred — the floating-point representation slightly *overestimates* the value of 0.1:

$$1.10011001100110011001101_2 \times 2^{-4} = 0.10000000149011611938\dots$$

Therefore, in single-precision floating point arithmetic,

$$0.1 + 0.1 + 0.1 = 0.30000000447034835815429688\dots$$

But what is 0.3 in single precision floating-point representation? Doing a quick conversion, we find that

$$0.3 \rightarrow 0.30000001192092895508$$

which is a different result from  $0.1 + 0.1 + 0.1$ . Therefore, in single precision floating-point arithmetic,  $0.1 + 0.1 + 0.1 \neq 0.3$ .

### 1.3.1 Machine Epsilon

In order to get a rough ‘upper bound’ on the relative round-off error that occurs during floating-point arithmetic, it is common to see standards (such as IEEE754), programming languages (such as Fortran and Python), and software packages (such as Mathematica) quote something called the **machine epsilon**.

#### Machine Epsilon and Round-Off Error

Machine epsilon is defined by the difference between the floating-point representation of 1 and the next highest number. In single precision, we know that

$$1 \equiv 1.000000000000000000000000000000_2$$

The next highest number is represented by changing the least significant bit in the mantissa to 1, giving

$$1.000000000000000000000000000001_2$$

Therefore, in single precision IEEE754 floating-point representation,

$$\epsilon = 1.000000000000000000000000000001_2 - 1.000000000000000000000000000000_2 = 2^{-23}$$

Repeating this process for double precision floating-point numbers results in  $\epsilon = 2^{-52}$ . In general, if the mantissa is encoded using  $M$  bits — resulting in  $M + 1$  significant figures — then the machine epsilon is  $\epsilon = 2^{-M}$ .

The important take-away here is that this value represents the *minimum possible spacing* between two floating point numbers, and as such provides an *upper bound* on the relative truncation error introduced by floating-point numbers.

To see how this works, lets consider a floating-point representation of a number  $x$ ; we'll denote this  $fl(x)$ . To calculate the **relative error** of the floating-point representation, we subtract the original exact value  $x$ , and divide by  $x$ :

$$\text{relative error} = \frac{fl(x) - x}{x}$$

Now, we know that the relative error **must** be bound by the machine epsilon, so substituting this in to form an inequality:

$$-\epsilon \leq \frac{fl(x) - x}{x} \leq \epsilon \quad (1.5)$$

By taking the absolute value and rearranging this equation, we also come across an expression for an upper bound on the **absolute error** of the floating-point representation:

$$|fl(x) - x| \leq \epsilon|x|$$

Ultimately, however, while the IEEE754 floating-point numbers have their quirks and setbacks, alternative proposals and standards have their own oddities. The sheer dominance of the IEEE754 standard in modern computing means that, although we sometimes still have to worry about precision, accuracy, and round-off errors, we can do so using a standard that will work the same way on almost all modern machines.

In the next two chapters, we'll start putting our knowledge of floating-point representation into practice — by actually bringing them into existence on a computer near you — using either Fortran (Chap. 2) or Python (Chap. 3).<sup>2</sup> It is up to you which programming language you would like to delve into, as Chap. 4 and onwards will include both Fortran *and* Python examples.

### Further reading

This chapter introduces the topic of floating-point arithmetic, and forces you to consider a topic (number representation) that normally does not appear in most analytic work. As we will see repeatedly in this text, however, you could write a whole textbook on this topic — and people have! If you would like to read more on floating-point representation, how it's implemented, and relevant applications to computational science and mathematics, you cannot go past the *Handbook of Floating-Point Arithmetic*, the definitive textbook on the subject.

- ▶ This result also generalises to *operations* of floating-point numbers. For example, the square root

$$|fl(\sqrt{x}) - \sqrt{x}| \leq \epsilon|\sqrt{x}|$$

and addition

$$|fl(x+y) - (x+y)| \leq \epsilon|x+y|$$

<sup>2</sup>This is a *choose your own adventure* textbook!

- Muller, J.M., Brunie, N., de Dinechin, F., et al. (2018). Handbook of Floating-Point Arithmetic. New York: Springer International Publishing, ISBN 978-3-319-76526-6.
-

## Exercises

- P1.1** Convert the following binary numbers into base-10:
- (a)  $110110_2$
  - (b)  $0.000111_2$
  - (c)  $-110.011011_2$
  - (d)  $1.01001_2 \times 2^{-2}$
- P1.2** Convert the following numbers into binary:
- (a) 146
  - (b) 0.8375
  - (c) -23.9
  - (d) 2.1
- P1.3** Repeat P1.2, but this time write out the 32-bit single precision floating point representation of each of the real numbers.
- P1.4** What decimal values do the following single precision floating-point binary numbers represent?
- (a) **0 00000101 00101010000101001011110**
  - (b) **1 01111011 10011001100110011001101**
  - (c) **1 00000000 0000000000000000000000000000000**
  - (d) **0 01111101 01010101010101010101011**
  - (e) **1 11111111 0000000000000000000000000000000**
  - (f) **0 00000000 00100010000010000001000**
  - (g) **0 11111111 00100010010010010011010**
- P1.5**
- (a) Convert  $1/3 = 0.3333\dots$  into:
    - (i) binary
    - (ii) scientific binary notation
    - (iii) a 32-bit floating-point representation
    - (iv) a 64-bit floating-point representation
  - (b) Convert the 32-bit floating-point representation *back* into base-10, and calculate the absolute round-off error. Check that it satisfies the machine epsilon inequality.
  - (c) Repeat part (b) for the 64 bit floating-point representation. Compare this to the absolute round-off error in the single precision case.
- P1.6** In Example 1.3, we saw that the property of associativity fails in floating point arithmetic. Confirm that the property of distributivity also fails in the expression

$$2.7(3.5 + 0.8) = 2.7 \times 3.5 + 2.7 \times 0.8$$

by evaluating both sides using floating-point arithmetic.



## Chapter 2

# Fortran

Fortran, originally standing for **F**ormula **T**ranslating System, is a programming language first developed in 1957 at IBM. Whilst the early versions of Fortran might seem archaic and obsolete to us today (they required fixed form input and were entered onto punched cards using a keypunch keyboard), its ease of use compared to assembly programming and features such as complex data types sparked a revolution in scientific and numeric computation. With an emphasis on speed and efficiency, Fortran quickly became the language of choice for those in science and engineering, and remains in use to this day for highly-intensive scientific computation.

Over the years, Fortran has changed significantly, and on the surface bears little resemblance to its original incarnation. For example, modern Fortran now has free-form input with case-insensitive commands, inline comments, and array operations (introduced in Fortran 90), object-oriented programming (Fortran 2003), and parallel processing (Fortran 2008).

In this chapter, we'll provide a brief introduction to Fortran, setting you on the path to solve simple problems in physics.

### Example 2.1

Hello World!

**Solution:** hello\_world.f90

```
program first
    implicit none
    ! write 'Hello World!' to stdout
    write(*,*)"Hello world!"
end program first
```

and then compile and run the code using

```
$ gfortran hello_world.f90 -o hello_world
$ ./hello_world
```

In Example 2.1, we illustrated one of the simplest Fortran programs; upon running, the statement ‘Hello World!’ is outputted to the terminal.

Note the code structure – all code takes place between the **program** first and **end program** first statements, where **first** is the name given to the program.

Meanwhile starting a line with ! indicates a comment (the line is to be ignored when compiling), whilst the **write(\*,\*)** instructs the computer to write the statement "Hello world!" to the screen (indicated by the first asterisk) using free-format (the second asterisk). The **implicit none** statement turns out to be a vitally important statement — we will see why in more detail in Sect. 2.1.3.

## 2.1 Variable Declaration

### 2.1.1 Using Scalar Variables

#### Example 2.2

Write a program that converts temperatures in Celsius to Fahrenheit

**Solution:**

```
program temperature
    implicit none

    ! declare variables
    real :: DegC, DegF

    write(*,*)"Please type in temp in Celsius"
    ! the read statement reads input from the keyboard,
    ! and stores it in variable DegC
    read(*,*)DegC

    DegF = (9./5.)*DegC + 32.

    ! The write statement accepts multiple
    ! strings or variables, separated by commas
    write(*,*)"This equals to",DegF,"Fahrenheit"
end program temperature
```

- ▶ Note that unlike other static languages such as C and C++, Fortran requires **all** variable declarations to take place in a block at the beginning of the program.  
**You cannot declare variables in the middle of a Fortran program.**

In this example, we have defined **DegC** and **DegF** to be **real** or **real(4)** variables; that is, they can store single-precision real numbers (i.e. with absolute values ranging from  $10^{-37}$  to  $10^{37}$  and a precision of 7 decimal digits). Other numeric data types available in Fortran include single and double precision integers, reals, and complex numbers – you can even set the precision arbitrarily if you would like. See Table 2.1 for a full list and description of the variable types available in Fortran.

Note that Fortran also includes several numerical inquiry functions (see Table 2.2) – these functions act on the data type of variable **x** (they are

independent of the value of  $x$ ) to provide information about the range and precision of the variables data type. Try playing around with these function on various data types and see what values they output.

<code>integer</code>	<b>single precision integer</b> with values ranging from $-2^{31} = -2147483648$ to $2^{31} = 2147483647$
<code>integer(4)</code>	
<code>integer(8)</code>	<b>double precision integer</b> with values ranging from $-2^{63} = -9,223,372,036,854,775,808$ to $2^{63} - 1 = 9,223,372,036,854,775,807$
<code>real</code>	<b>single precision real number</b> with absolute values ranging from $10^{-37}$ to $10^{37}$ and precision of 7 decimal digits
<code>real(4)</code>	
<code>real(8)</code>	<b>double precision real number</b> with absolute values ranging from $10^{-308}$ to $10^{308}$ and precision of 15 significant digits
<code>complex</code>	<b>single precision complex number</b> , written as a pair of <code>real(4)</code> numbers; e.g. <code>(5.234, -9.432e-1)</code> represents $5.234 - 0.9432i$
<code>complex(8)</code>	
<code>character</code>	a <b>single alphanumeric character</b> , for example ' <code>a</code> '
<code>character(len=X)</code>	<b>string containing X alphanumeric characters</b> , e.g. ' <code>abcdefg123</code> '
<code>logical</code>	<b>boolean literal</b> which can only take two values; <code>.true.</code> or <code>.false.</code>

Table 2.1 Fortran intrinsic data types

## 2.1.2 Implicit (and Explicit) Data Conversion

### Integers

Say you have a variable  $N$ , declared as an `integer` – that is, the value of  $N$  must be a whole number in the range  $-2^{31} \leq N \leq 2^{31}$ . If you define  $N$  in terms of real numbers, for example

```
integer :: N
N = 3.41
N = 5./2.
```

▶ Note that the presence of a decimal point indicates a `real` number rather than an `integer`, even if no decimal places are present. So implicit data conversion still occurs for `N = 3.`, but not for `N = 3`.

<code>digits(x)</code>	number of significant digits
<code>huge(x)</code>	largest number
<code>maxexponent(x)</code>	maximum exponent possible for x <code>real</code>
<code>minexponent(x)</code>	minimum exponent possible for x <code>real</code>
<code>precision(x)</code>	decimal precision for x <code>real</code> or <code>complex</code>
<code>tiny(x)</code>	smallest positive number for x <code>real</code>
<code>range(x)</code>	decimal exponent range

**Table 2.2** Numerical inquiry functions

then Fortran will do what is called an **implicit data conversion** and drop all decimal places. In line 2, the real number 3.41 is converted into the integer 3 and thus `N=3`, whilst in line 3 we are trying to assign `N = 2.5`, resulting in an implicit conversion and we actually end up with `N = 2`.

One case where you have to be especially vigilant is **integer division**; for example, consider

```
real :: x
x = 5/2
```

Although this time we have defined `x` as a `real` variable, the division taking place is between two *integers* (note the lack of decimal places), and so the result is also an integer – this is called integer division. So instead of assigning the value 2.5 to variable `x`, as we might expect, implicit conversion and truncation still occurs. Try running this program yourself, and then write the value of `x` to the screen. Instead of `x = 2.5`, what do you get instead?

In order to avoid integer division, we could make sure that at least one of the numbers in the division is a *real* number. Note, however, that it is **good practice** to ensure *all* constants used are the same type to avoid the idiosyncrasies surrounding implicit conversion — this is essential to avoid unintentional precision loss and inefficiency.

```
real :: x
x = 5./2
x = real(5)/2. ! or we can do an explicit conversion
```

Explicit data type conversions can be used when mixing variable types in an arithmetic statement; for example

```
real    :: x
integer :: n
n = 2
x = real(n)/5.
```

See Table 2.3 for more type conversion functions.

<code>int(x)</code> : truncated integer
<code>nint(x)</code> : nearest integer
<code>real(x)</code> : nearest real
<code>real(x,8)</code> : nearest <code>real(8)</code>
<code>cmplx(x,y)</code> : complex $x + iy$
<code>cmplx(x,y,8)</code> : double complex

**Table 2.3** Intrinsic Fortran data type conversion functions

- In order to keep track of precision, **avoid implicit conversion** by never mixing different types in Fortran arithmetic. Instead, use explicit conversion functions where you need to.

The one exception is for exponents of real numbers; for instance,  $2.7^3$  is fine as it is simply the same as writing  $2.7 \times 2.7 \times 2.7$ , and no precision is lost.

### Precision and Round-Off Errors

Implicit conversion also needs to be considered when working with double precision reals or `real(8)`.

#### Example 2.3

```
program precision
    real :: x
    real(8) :: y, z
    x = 1.1
    y = 1.1
    z = 1.1d0
    write(*,*)"x =", x, " , y =", y, " , z =", z
end program precision
```

*Output:*  $x = 1.10000002$  ,  $y = 1.10000002$  ,  $z = 1.1$

In the above program, what causes the error in variable  $x$ ? How do you explain the difference between variables  $y$  and  $z$ ?

#### Solution:

The error in variable  $x$  is **round-off error**, and occurs due to the fact that the binary representation of 1.1 cannot be exactly represented as a decimal – it has an infinite number of decimal places (similar to how  $1/3 = 0.3333333\dots$  in base 10). For example,

$$1.1_{10} = 1.0001100110011\dots_2$$

As single precision real only preserves a finite number of decimal places, this gets truncated, and thus, when converted back into base 10 round off error is introduced in the final decimal place.

The reason we also see round-off error in the final *displayed* decimal place of  $y$  (note that as  $y$  is double precision, it is stored in memory with more significant figures than  $x$  – we just aren't seeing all of them here) is that we are assigning to  $y$  a *single precision real!* In order to avoid this, we must instruct the compiler that the literal `1.1` is exact to double precision – in effect padding the right hand side with zeros to fill all 17 significant figures. This is done by writing `1.1d0`, which can be read like

$$1.1d0 \equiv 1.100000000000000 \times 10^0$$

Note that round-off error is still present in the value of  $z$ , but this time is much less significant, occurring in the 16th decimal place.

As can be seen from Example 2.3, it is important to use the syntax

$$0.43dX \equiv 0.430000000000000 \times 10^X$$

when working with double precision reals (where  $X$  refers to the exponent in scientific notation), in order to avoid single precision round-off errors affecting your double precision calculations.

#### ► Machine epsilon

Provides quantitative upper bound on the **relative** round-off error.

#### Single precision:

$$\epsilon = 2^{-23}$$

#### Double precision:

$$\epsilon = 2^{-52}$$

#### ► Scientific notation can also be used to enter **single precision** reals; in this case you use the syntax `0.43eX`

### 2.1.3 Implicit None

You may have noticed that up to now we haven't explained the statement **implicit none** statement present in Exercises 2.1 and 2.2; this statement is so important that it deserves its own section. In the original Fortran specification, variables did not need to be explicitly declared; in fact, Fortran by default treats all variables that start with the letters i, j, k, l, m, and n as integers, and all other variables as floats.

This can cause huge issues when debugging your code! For example,

- potential confusion between variable types – your naming might cause your variable to unknowingly be an integer!
- typos in variable names will not stop your code compiling, as Fortran happily accepts your new, undeclared variable (!! ) – leading to incorrect output

► In summary: *always use implicit none!*

The **implicit none** statement always comes straight after the **program** statement, and simply instructs the Fortran compiler to require all variables be declared before use, avoiding the aforementioned pitfalls. It is good practice to make sure that the **implicit none** statement is **always present** in any Fortran code.

## 2.2 Operators and Control Statements

Fortran has four main categories of operators; arithmetic, character, relational, and logical.

### 2.2.1 Arithmetic

In order of priority, the arithmetic operators are:

- **Exponentiation ( $\star\star$ )**

Note that, somewhat nonintuitively, exponentiation in Fortran is right to left associative; that is,  $2\star\star3\star\star2 = 2^{(3^2)} = 2^9 = 512$ , and **not**  $2\star\star3\star\star2 \neq (2^3)^2 = 8^2 = 64$ .

► To make your code easier to read and to avoid ambiguities, always use brackets in arithmetic operations!

- **Multiplication ( $\star$ ) and division ( $/$ )**

Unlike exponentiation, these are left to right associative;  $a\star b\star c = (a \times b) \times c$ .

- **Addition ( $+$ ) and subtraction ( $-$ )**

Like multiplication and subtraction, these are also left to right associative. Note that  $-$  can also be used as a unary operator (it only acts on one operand). For example,  $-3\star\star2 = -(3^2)$  acts to negate the result of the exponentiation.

### 2.2.2 Character

Fortran only has one character operator, the concatenation operator  $//$ . This allows multiple strings to be joined or concatenated.

#### Example 2.4

The following Fortran program writes ‘Hello world’ to the screen by concatenating the two strings “Hello ” and “world”:

```
program concat
    implicit none
    character(len=6) :: str1 = "Hello "
    character(len=5) :: str2 = "world"
    character(len=11) :: str3

    str3 = str1 // str2
    write(*,*)str3
end program concat
```

Output: Hello world

### 2.2.3 Relational

The relational or comparative operators allow you to compare the values of different expressions or variables. These will become a lot more useful once `if` statements are covered, but they are relatively simple in construction.

Note that the relational operators have a lower precedence than the arithmetic operators, but a higher precedence than the logical operators. See Table 2.4 for the full list of intrinsic relational operators.

.eq. or <code>==</code>	equal to
.ne. or <code>/=</code>	not equal
.lt. or <code>&lt;</code>	less than
.gt. or <code>&gt;</code>	greater than
.le. or <code>&lt;=</code>	less than or equal to
.ge. or <code>&gt;=</code>	greater than or equal to

Table 2.4 Relational/comparison Fortran operators

### 2.2.4 Logical

Logical operators have the lowest precedence, and act on logical variables and the two logical constants `.true.` and `.false.`. There are three main logical operators, `.not.`, `.or.` and `.and.` (in order of descending precedence) – these are described in Table 2.5.

<code>.not.</code>	logical negation (acts from right to left)
<code>.or.</code>	<code>.true.</code> if at least one operand is <code>.true.</code>
<code>.and.</code>	<code>.true.</code> if and only if both operands are <code>.true.</code>

Table 2.5 Relational/comparison Fortran operators

- ▶ When combining logical operators, make sure there is a space between them  
 $x .and. .not. y$   
 or ‘share’ the dots, like so  
 $x .and.not. y$

### Example 2.5

In the program below, what do you expect the value of  $y$  to be?

```
program logops
    implicit none
    logical :: x, y
    x = .true.
    y = x .and. 3.5 <= 5.
end program logops
```

**Solution:** Following the order of precedence, the expression for  $y$  will be evaluated as  $x .and. (3.5 <= 5.)$ . As 3.5 is obviously less than 5, the bracketed statement is true. Since  $x$  is also `.true.`, thus

$$y = x .and. 3.5 <= 5. \Rightarrow y = .true. .and. .true. \Rightarrow y = .true.$$

### 2.2.5 Control Statements

Control statements are an essential component of programming, allowing you to control which section of code is executed based on the state of the program, and how many times certain sections should repeat. There are two main control statements we will consider here; **selection** (**if** and **select case** statements) and **iterations** (**do** and **do while** loops).

#### The **if** Statement

The **if** statement enables a choice of code to execute depending on various logical conditions. It is structured as follows:

```
if (condition1) then
    ! Fortran code 1
else if (condition2) then
    ! Fortran code 2
else if (condition3) then
    ! Fortran code 3
else
    ! Fortran code else
end if
```

When the above **if** structure is encountered, the conditions are evaluated sequentially (i.e. from the top down). When the first condition that evaluates to `.true.` is found, the corresponding Fortran sequence will be executed. Otherwise, the **else** code sequence is executed.

Note that the **else if** and **else** statements are completely optional and do not need to be included; for example, when **else** is not included, then no code is executed unless the **if/else if** conditions are met.

► The **endif** and **end if** statements can be used interchangeably, as can **elseif** and **else if**.

► **if** statements can be infinitely nested inside of other **if** statements when needed.

#### Example 2.6

Calculate the smallest of three real numbers  $a$ ,  $b$ , and  $c$ .

**Solution:**

```
if (a < b .and. a < c) then
    Result = a
else if (b < a .and. b < c) then
    Result = b
else
    Result = c
end if
```

Another useful **if** construction is the **logical if** or **one-line if** statement – this can be written in one line (making your code more concise) as follows

```
if (condition) statement
```

Note that the **then** statement is **not** present between the condition and statement in this case!

### Case Selection

If a selection is required which depends solely on the value of a single **logical**, **integer**, or **character** variable, then the **select case** or **case-switch** statement may be used:

```
select case (variable)
  case(value1)
    ! Fortran code 1
  case(value1)
    ! Fortran code 2
  case default
    ! Fortran code default
end select
```

Note that the catch-all **case default**, which is only executed if no match is found, is again optional, similar to **else**. However it is good practice to always include it regardless as a fail-safe.

### Example 2.7

Return the age of Tom (29) and Sally (23) from their name.

Solution:

```
select case (name)
  case('Tom')
    age = 29
  case('Sally')
    age = 23
  case default
    write(*,*)'Error! Name not recognised'
end select
```

### 2.2.6 do loops

► **end do** and **end do** can be used interchangeably.

► Like **if** statements, **do** loops can be infinitely nested.

The **do** loop allows you to run a particular section of code for a specific number of iterations;

```
do i = istart, iend, istep
  ! fortran code
end do
```

where `i` is an arbitrary variable to iterate over (which must have been declared as an integer), `istart` is the initial value of the iteration variable, `iend` the final value of the iteration variable, and `istep` the step size of each iteration. Note that `istep` is optional – if not included, the step size is 1 (i.e. `i` increments by 1 each iteration). Alternatively, if `isize` is *negative*, then the loop counts *downwards*.

### Example 2.8

Calculate the squares of all numbers between 5 and 10

**Solution:**

```
program squares
    implicit none
    integer :: i

    do i=5, 10
        write(*,*)i**2
    end do
end program squares
```

Two flow control statements also provide for additional control in `do` loops; these are the `cycle` and `exit`. See Table 2.6 for more details on using these statements.

<code>cycle</code>	all remaining statements up to the <code>end do</code> are to be skipped, and the next loop iteration begins
<code>exit</code>	the do loop is terminated and the program continues

Table 2.6 Loop control statements

### do while loops

`do while` loops are used to repeat a section of code until a certain condition has been met:

```
do while (condition)
    ! fortran code
end do
```

**Example 2.9 do while loop**

List all square numbers less than 60

**Solution:**

```
program squares2
    implicit none
    integer :: i

    i = 1

    do while (i**2 < 60)
        write(*,*)i**2
        i = i+1
    end do
end program squares2
```

**2.2.7 General do loops**

In a general do loop, no arguments are provided to the **do** statement whatsoever – full control of the loop is instead provided via the **cycle** and **exit** statements.

- ▶ If using a general do loop, make sure to always have an exit statement somewhere, otherwise the loop will run forever!

- ▶ There is one more form the do loop can take, known as an **implied do loop**. These are useful when working with arrays and reading/writing data, and we'll come across them later.

**Example 2.10 General do loop**

List all square numbers between 10 and 60

**Solution:**

```
program squares3
    implicit none
    integer :: i, isq

    i = 0

    do
        i = i + 1
        isq = i**2

        if (isq < 10) then
            cycle
        else if (isq > 60) then
            exit
        end if

        write(*,*)isq
    end do
end program squares3
```

## 2.3 String Manipulation

Fortran provides some intrinsic functions for manipulating strings, as detailed in Table 2.7, which you might find very useful.

<code>len(string)</code>	returns an integer specifying the length of the string
<code>index(string,substring)</code>	searches for the first occurrence of <code>substring</code> in <code>string</code> , from left to right. If <code>substring</code> is found, it returns an integer specifying the start position of <code>substring</code> in <code>string</code> ; otherwise <code>0</code> is returned
<code>adjustl(string)</code> <code>adjustr(string)</code>	left/right justifies the characters in the variable <code>string</code> , by removing blank padding from the left/right. For example <code>adjustl(" abc") = "abc "</code> (note that the length of the string is unchanged)
<code>trim(string)</code>	removes trailing blanks from the string; i.e. <code>trim("abc ") = "abc"</code>

Table 2.7 Intrinsic character manipulation functions

Of the character manipulation functions listed above, `adjustl` and `trim` are especially useful when it comes to concatenating strings, as a method of removing annoying ‘trailing blanks’ and misplaced spaces.

- ▶ It is good practice to always `adjustl` and `trim` any strings you are concatenating, to avoid the chance that forgotten trailing blanks result in unexpected spaces in your concatenated string.

### Example 2.11

```
program concat2
  implicit none
  character(20) :: str1 = "Hello", str2 = "world!"

  write(*,*) str1 // " " // str2
  write(*,*) adjustl(trim(str1)) // " " &
             // adjustl(trim(str2))
end program concat2
```

Output:

```
Hello          world!
Hello world!
```

- ▶ If a line in your Fortran code becomes too long and unwieldy, you can break it up into multiple lines using `&` as a delimiter (see Example 2.11).

This is known as a **continuation line**.

### 2.3.1 Substrings

Another neat string manipulation gives us the ability to extract consecutive parts of a string, otherwise known as a substring. This can be done in Fortran via the syntax

```
string(istart:iend)
```

where `istart` is an integer indicating the first position of the substring, and `iend` is an integer indicating the end position of the substring.

Note that if `istart` is missing, the beginning of the substring is assumed to coincide with the beginning of the string (i.e. `istart = 1`); and if `iend` is missing, the end of the substring coincides with the *end* of the string (i.e. `iend = len(string)`).

For example, consider the string `str = "arbitrary string"`:

```
str(5:) = "trary string"
str(3:8) = "bitrar"
str(:6) = "arbitr"
```

### 2.3.2 Format Statements

#### Data types:

F	real	, fixed point format
E	real	, exponential notation
ES	real	, scientific notation
EN	real	, eng. notation
I	integer	
A	character	
L	logical	

#### Positioning:

X	space
/	newline
T	tab

#### Sign control:

SP	print positive (+) signs
SN	don't print positive signs

So far, we have been using the statements `read(*,*)` and `write(*,*)` to read/write **free-form strings**, as indicated by the second asterisk (\*). To have finer control on the display format, we can make use of Fortran's **format** statement.

Using the format code letters (see Table 2.8), we can precisely define how we would like particular data type to be displayed (referred to as the **field**). For example, using the following symbol convention,

- *w*: total width of the field
- *d*: number of digits to the right of the decimal point
- *e*: number of digits in the exponent (*optional, default 2*)
- *c*: number of times to repeat the field (*optional*)

we are able to construct the following format statements:

**Table 2.8** Format code letters.

Note that these are not case-sensitive.

Field	Format Statements
<b>real</b> decimal form	<i>cFw.d</i>
<b>real</b> exp. form	<i>cEw.d</i> , <i>cEw.dEe</i>
<b>real</b> sci. form	<i>cESw.d</i> , <i>cESw.dEe</i>
<b>real</b> eng. form	<i>cENw.d</i> , <i>cENw.dEe</i>
<b>integer</b>	<i>cIw</i>
<b>character</b> string	<i>cA</i> , <i>cAw</i> ( <i>w</i> is optional)
<b>logical</b>	<i>cLw</i>
positioning	<i>cX</i> , <i>c/</i> , <i>cT</i>

**Table 2.9** Available format statements for various display fields

There are two main approaches to using format statements. They can be entered directly into the **read/write** statement, like so,

```
integer :: i = 28
real    :: x = 3.14159, y = 2.7182818
write(*,'(A,i3,2f8.3)')"The values are: ",i,x
```

or the **format** statement can be used, along with an (arbitrary) numeric label:

```
integer :: i = 28
real    :: x = 3.14159, y = 2.7182818
write(*,100)"The values are: ",i,x,y
100 format(A,i3,2f8.3)
```

Regardless of how we enter the format statement, since we are using the same format statement **(A,i3,2f8.3)**, the outputs are identical:

```
The values are: 28 3.1416 2.7183
```

Some things to note:

- we have not provided a width specification for the character field, as it is optional – by default, the entire string is printed
- when the total width of the field *w* is larger than the number of significant digits to print, the field is **right justified**, padded with blank spaces
- the real variables *x* and *y* are **rounded** rather than truncated

► **Important:** if the total width of the field *w* is too small to fit the value in the specified format, Fortran will just print a string of asterisks instead (**\*\*\*\*\***). If this occurs, simply increase *w* for the affected field until the value is properly displayed.

For example, '**(E8.4)**', only being 8 characters in width, will not have room to display all 4 decimal places and will output as  
**\*\*\*\*\***

Instead, a field width of 10 is the minimum required to display **0.XXXXE+XX**, or '**(E10.4)**'

**Example 2.12** Exponential, scientific, and engineering notation

```

real :: x = 18947.81274593
! output x using E, ES, and EN formats
write(*,'(E10.4,ES10.4,EN10.4)')x,x,x
! force exponent to contain 3 digits
write(*,'(E10.4E3,ES10.4E3,EN10.4E3)')x,x,x

```

Output:

0.1895E+05	1.8948E+04	18.9478E+03
0.1895E+005	1.8948E+004	18.9478E+003

### Example 2.13 Sign control

```

real :: x = 18947.81274593, y = -0.2154212
! do not print positive signs (default)
write(*,'(2ES10.4)')x,y
! print positive signs for every field to the right of SP
write(*,'(SP,2ES10.4)')x,y

```

Output:

1.895E+04	-2.154E-01
+1.895E+04	-2.154E-01

### Example 2.14 Complex numbers

As complex numbers are a ‘tuple’ of two real numbers, their output format can be a little ambiguous. Have a look at the code and the output in this example, for various ways in which we can format complex numbers.

```

1 complex :: x = (2.6,-0.23)
2 ! default format
3 write(*,*)x
4 ! fixed point format
5 write(*,'(2f8.3)')x

```

Output:

( 2.5999990 , -0.230000004 )
2.600 -0.230

### Explanation:

- Line 3: the complex number is displayed using default formatting – this results in displaying the parenthesis that indicate data type **complex**, and printing the real and imaginary components to full single precision (note the rounding error in the least significant digit).

- Line 5: this time, we are instructing the compiler to print both the real and imaginary component separately, as fixed 3-digit decimals. Note that the **2** in the format statement is required, as the real and imaginary components are treated as two independent **real** numbers.

As with real numbers, the exponential, scientific, and engineering notation can also be used with complex numbers.

## 2.4 Input and Output

At some point, hard-coding data into a program becomes too lengthy and takes focus away from the algorithm itself, making it difficult to change input data on-the-go. This is where considering various methods of input and output becomes important – for example, by reading the input data from a file, a program can be run multiple times on different data sets without having to re-compile the code. Additionally, writing the output data to a file is significantly more convenient and less prone to error when a large amount of data is involved (whilst also providing the flexibility to use external data analysis software).

### 2.4.1 Opening and Closing Files

- ▶ The unit number can be any integer in the range  $1, 2, \dots, 99$ , however do not use `unit=5` or `unit=6`, as these are reserved for standard input and standard output respectively.

In order to make a file available to Fortran, it first needs to be ‘opened’, using the aptly named `open` statement,

```
open(unit=12, file='dir/filename')
```

where the `unit` number (in this case `12`) is used by Fortran to refer to the open file. For more options accepted by `open`, see Table 2.10

<code>iostat=ios</code>	<b>input/output status:</b> if no error occurred, integer variable <code>ios</code> is set to 0, otherwise <code>ios &gt; 0</code> .
<code>err=LABEL</code>	<b>error branching:</b> specifies a <i>label</i> (i.e. a line labelled by an integer) to which the program will <code>goto</code> if any error is detected.
<code>status</code>	<ul style="list-style-type: none"> <li><code>'old'</code>: an error occurs if the file doesn't exist</li> <li><code>'new'</code>: an error occurs if the file exists</li> <li><code>'replace'</code>: the file is first deleted (if it exists)</li> <li><code>'scratch'</code>: the file is deleted when closed</li> </ul>
<code>position</code>	<ul style="list-style-type: none"> <li><code>'asis'</code>: position the cursor at previously used position (<i>default</i>)</li> <li><code>'rewind'</code>: position the cursor at the start of the file</li> <li><code>'append'</code>: position the cursor at end of the file</li> </ul>
<code>action</code>	values are <code>'read'</code> , <code>'write'</code> , or <code>'readwrite'</code> ( <i>default</i> )

Table 2.10 `Open` optional arguments

When the file is no longer needed, it can be closed by using the `close` statement:

```
close(unit=12)
```

- ▶ You can close multiple files by separating unit numbers by commas, e.g. `close(12,15,8)`

Whilst it is not strictly necessary to close files (this is done automatically when the program terminates), closing files when they are no longer needed allows other programs to access them, and may avoid potential memory issues if working with large quantities of data. Note that, like `open`, the `close` statement also accepts the `iostat` and `err` optional arguments.

### 2.4.2 Writing to a File

Once the file is open, we can begin writing data to it. This is achieved via the `write` statement; however rather than writing to the terminal with `*` as we have done so far, instead we now write to the open file's unit number:

```
write(12,'(a,3f8.3)')"A new line of variables:",x,y,z
```

After writing a line to a file, `write` will automatically advance the file cursor to the next line – thus, every additional `write` statement will automatically write to a new line of the file. If for some reason you want to change this default behaviour, you can specify the optional argument `advance='no'`:

```
write(12,'(3f8.3)',advance='no')x,y,z
```

this will continue writing its output to the *same* line as the previous write statement.

Apart from `advance`, `write` – like `open` and `close` – also accepts the `iostat` and `err` optional arguments.

#### Example 2.15

Write a Fortran program that calculates the 5 times table, and outputs it to a file `5times.txt`

#### Solution:

This can be done by using a `do` loop to iterate over the times table and writing each line to the file:

```
program writefile
  implicit none
  integer :: i

  open(unit=10, file='5times.txt')

  do i=1, 10
    write(10,'(a,i2,a,i2)')'5 * ',i,' = ',5*i
  end do

  close(10)
end program writefile
```

Looking at the first four lines of `5times.txt`:

```
head -n4 5times.txt
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
```

### 2.4.3 Reading from a File

The `read` statement is used similarly to the `write` statement when working with external files; once the file is opened, each subsequent use of the `read` statement will read successive lines from left to right, and store separated values in specified variables.

- ▶ `read` also supports tab-separated and comma-separated values, in addition to space-separated values.

For example, if a file contains 3 columns of space-separated data of the form

```
1    0.72      Jerry
6    0.176     Larry
7    0.0048    Terry
7    0.0056    Garry
```

then the statement

```
read(12,*),s,p,name
```

- ▶ Note: failing to match the variable type with the data type you are trying to assign will lead to errors.

will assign the data to the variables `integer s`, `real p`, and `character name`. After the first `read` statement, we will have:

```
s=1, p=0.72, name="Jerry"
```

However note that subsequent `read` statements will overwrite these values; for instance after repeating the above `read` statement, the variables will now hold the values

```
s=6, p=0.176, name="Larry"
```

In order to avoid this pitfall, you will need to use arrays – we will come across these in a later chapter.

### Reading to the End of a File

If you keep calling `read` statements, you'll find that, once you have more `read` statements than lines in the file you're reading, you will get the following error when running your program:

```
Fortran runtime error: End of file
```

To avoid this, you can use pass the argument `end=LABEL`, which instructs `read` to `goto` the specified labelled line number once the end of the file is reached. See Example 2.16 for how this works in practice.

Like `write` and `open`, `read` also supports the `iostat` and `err` arguments (useful when troubleshooting, as you don't want the potentially unknown formatting of the external file to crash your program!).

**Example 2.16**

Write a Fortran program that reads in a one-column list of data, and calculates the sum.

**Solution:**

This can be done by using a `do` loop to iterate through the file:

```
program readsum
    implicit none
    real :: x, sum

    sum = 0

    open(unit=12, file='test.dat', action='read')

    do
        read(12,*,end=100)x
        sum = sum + x
    end do

    100 close(10)
    write(*,*)sum
end program readsum
```

## 2.5 Intrinsic Functions

So far in this chapter, we have introduced three small (but important) categories of intrinsic Fortran functions; the data type conversion functions (Table 2.3), the numerical inquiry functions (Table 2.2) and the character functions (Table 2.7).

However, this barely scratches the surface – and we are still missing one of the most important categories in scientific computation, mathematical functions! In this section, we will list the mathematical intrinsic functions (Table 2.11) in addition to complex (Table 2.12) and numeric intrinsic functions (Table 2.13).

Function	Arg.	Return	Description
<code>sqrt(x)</code>	<code>real</code> <code>complex</code>	<code>real</code> <code>complex</code>	square root $\sqrt{x}$
<code>exp(x)</code>	<code>real</code> <code>complex</code>	<code>real</code> <code>complex</code>	exponential function $e^x$
<code>log(x)</code>	<code>real</code> <code>complex</code>	<code>real</code> <code>complex</code>	natural logarithm $\ln(x)$
<code>log10(x)</code>	<code>real</code>	<code>real</code>	base 10 logarithm $\log(x)$
<code>sin(x)</code>	<code>real</code> <code>complex</code>	<code>real</code> <code>complex</code>	sine (in radians)
<code>cos(x)</code>	<code>real</code> <code>complex</code>	<code>real</code> <code>complex</code>	cosine (in radians)
<code>tan(x)</code>	<code>real</code>	<code>real</code>	tangent (in radians)
<code>asin(x)</code>	<code>real</code>	<code>real</code>	inverse sine $\sin^{-1}(x)$
<code>acos(x)</code>	<code>real</code>	<code>real</code>	inverse cosine $\cos^{-1}(x)$
<code>atan(x)</code>	<code>real</code>	<code>real</code>	inverse tangent $\tan^{-1}(x)$
<code>sinh(x)</code>	<code>real</code>	<code>real</code>	hyperbolic sine $\sinh(x)$
<code>cosh(x)</code>	<code>real</code>	<code>real</code>	hyperbolic cosine $\cosh(x)$
<code>tanh(x)</code>	<code>real</code>	<code>real</code>	hyperbolic tangent $\tanh(x)$

Table 2.11 Mathematical functions

Function	Arg	Return	Description
<code>real(z)</code>	<code>complex</code>	<code>real</code>	real part of a complex number
<code>aimag(z)</code>	<code>complex</code>	<code>real</code>	imaginary part of a complex number
<code>conj(z)</code>	<code>complex</code>	<code>complex</code>	complex conjugate $\bar{z}$
<code>abs(z)</code>	<code>complex</code>	<code>real</code>	For $z = x+iy$ , $ z  = \sqrt{x^2 + y^2}$
<code>atan2(y,x)</code>	<code>real</code>	<code>real</code>	returns $\arg(x + iy)$

Table 2.12 Complex functions

Function	Arg.	Return	Description
<code>abs(x)</code>	<code>real integer</code>	<code>real integer</code>	absolute value $ x $
<code>aint(x)</code>	<code>real</code>	<code>real</code>	truncated whole number
<code>anint(x)</code>	<code>real</code>	<code>real</code>	rounding nearest whole number
<code>mod(x,y)</code>	<code>integer real</code>	<code>integer real</code>	remainder due to $x/y$ $\text{mod}(x,y) = x - \text{int}(x/y)*y$
<code>modulo(x,y)</code>	<code>integer real</code>	<code>integer real</code>	returns $x$ modulo $y$ $\text{modulo}(x,y) = x - \text{floor}(x/y)*y$
<code>floor(x)</code>	<code>real</code>	<code>real</code>	round down to the nearest whole number
<code>ceiling(x)</code>	<code>real</code>	<code>real</code>	round up to the nearest whole number
<code>sign(x,y)</code>	<code>real</code>	<code>real</code>	sign transfer from $y$ to $x$ $\text{sign}(x,y) = (y/ y ) x $
<code>max(x1,x2,...)</code>	<code>integer real</code>	<code>integer real</code>	returns the maximum of a list of numbers (minimum 2 arguments)
<code>min(x1,x2,...)</code>	<code>integer real</code>	<code>integer real</code>	returns the minimum of a list of numbers (minimum 2 arguments)

Table 2.13 Numerical functions

## 2.6 Arrays

So far, we have only considered scalar variables; if we would like to save an ‘array’ of scalar variables using the methods already discussed, we would have to write the value of each variable to a file, and read it back in when needed. This would result in a significantly slower program than one that keeps an array of variables in memory! Furthermore, what if we would like to do vector and matrix operations?

This is where arrays come in – a Fortran data type in their own right.

### 2.6.1 Declaration and Indexing

Arrays are declared in the variable declaration block at the beginning of the program, by specifying the data type of each **element** along with the array shape. For example, to declare an array of ten integers and then a  $3 \times 4$  array of real values, we can use the **dimension** statement on the left-hand side,

```
integer, dimension(10) :: a
real(8), dimension(3,4) :: b
```

or we can specify the array shape on the right hand side:

```
integer :: a(10)
real(8) :: b(3,4)
```

The latter format has the advantage that we can declare both scalar and vector variables on the same line:

```
integer :: i, j, a(10)
real(8) :: b(3,4), x0, x1, dt
```

Note that in this example **a** is a **rank-1** array, and **b** is a **rank-2** array – the **rank** of an array refers to the number of dimensions. For convenience, we will refer to rank-1 arrays from now on as **vectors**, and rank-2 arrays as **matrices**. We can also declare higher rank arrays; for instance **c(2,2,5,4,3)** is a rank-5 array.

### 2.6.2 Array Constructors

Array constructors are symbols provided by Fortran to enable you to type an array directly into your code. You have a choice of using either **( / )** or **[ ]**:

```
a = (/1, 6, -2, 3, 5, 1, 7, 11, 0, -5/)
a = [1, 6, -2, 3, 5, 1, 7, 11, 0, -5]
```

- ▶ Array dimension size must always be specified using integers!

- ▶ When assigning a constructed arrays, make sure it has the same size/shape as the array variable!

Whilst the `[]` syntax is more concise (and my preferred syntax), you should be aware that it was only introduced in the Fortran 2003 revision. Most compilers (including `gfortran`) now have included this syntax, as well as several other Fortran 2003 features. However, in the rare case you are using an older compiler version or need to stick strongly to pre-2003 standards, you can use the more awkward `(//)` syntax.

### 2.6.3 Implied do loops

Arrays can also be constructed by implied `do` loops, which use round brackets `()` to quickly and easily apply an implied `do` loop in place:

```
a = [(i, i=-4, 14, 2)] ! i.e. [-4, -2, 0, 2, 4, 6, 8, 10, 12, 14]
a = [(i**2, i=1, 10)] ! i.e. [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Like the regular `do` loop, the iterating variable (in this case `i`) must be declared, and the syntax is the familiar `i=istart,iend,stride`.

### 2.6.4 Slicing

When working with arrays, there are several ways we can access the stored data:

- we can access individual **scalar elements** by using brackets to indicate the element index. For example, `a(2)` will give the second element of `a`, which will be of type `integer` (since `a` was initialised as an `integer` array)
- access a range of elements, or an **array section**. This can be done by using the **triplet subscript** notation `istart:iend:stride` where `istart` is the starting index, `iend` the ending index, and `stride` is the increment between elements.

For example,

```
a(2:4)      ! returns array [a(2), a(3), a(4)]
a(1:10:3)   ! returns array [a(1), a(4), a(7), a(10)]
a(7:6:-1)   ! returns array [a(7), a(6)]
```

Array slicing can also be performed on arrays with multiple dimensions. Consider array `b(3,4)`, defined earlier:

Note that the slice lower bound `istart`, upper bound `iend`, and `stride` are all *optional* – if not specified, the following defaults are used:

- `istart` default: the declared lower bound of the array dimension
- `iend` default: the declared upper bound of the array dimension
- `stride` default: 1

Thus, the following are *equivalent*:

$$\mathbf{b(2:,:) \equiv b(2:3:1,1:4:1), \quad a(:,:3) \equiv a(1:10:3)}$$

**Index numbering bounds**  
The starting index of an array dimension (or **lower bound**) can be specified when declaring an array by using the syntax `istart:iend`.

For example, the rank-1 array  
`real :: vec(0:4)`  
has length 5 and has elements indexed by  $\{0, 1, 2, 3, 4\}$ ; we say it has a **lower bound** 0 and **upper bound** 4.

If not specified, the lower bound default is 1.

### Vector substrings

In addition to accessing individual elements by passing their index, we can access an **array of elements** by passing an **array of indices**. You may even repeat indices in this case! For example,

`a([2,7,4,7])`

will return the rank-1 array

`[a(2),a(7),a(4),a(7)]`

Note that you can also use integer array variables as vector substrings:

`integer :: c(4) = [2,7,4,7]`

Now using `a(c)` should give the same result as previously

`b(2,:)` returns vector  $[b(2,1), b(2,2), b(2,3), b(2,4)]$

`b(2:3,1:2)` returns the 2x2 array  $\begin{bmatrix} b(2,1), b(2,2) \\ b(3,1), b(3,2) \end{bmatrix}$

`b(2:1:-1,4)` returns the 2x1 array  $\begin{bmatrix} b(2,4) \\ b(1,4) \end{bmatrix}$

## 2.6.5 Assignment

We can assign values to arrays/arrays sections using the various techniques described above – for example by using array constructors, looping over elements, or by passing sub-arrays of other variables.

- ▶ When assigning arrays, make sure the arrays on the left and right side of the equals sign are **conformal** – i.e. they have the same shape.

```
integer :: i
real    :: a(5), b(2,5,3), c(4,4)

! set a using an array constructor
a = [1.,5.,-0.43,1.23,0.]

! set the first row of b=a
b(1,:) = a
! set the 2nd row to be the square of the first row
b(2,:) = [(b(1,i)**2, i=1,5)]

! set all elements of c to 0.
c = 0.
! set diagonal elements of c to 1.
do i=1,4
  c(i,i) = 1.
end do
```

- ▶ Note that we can also mix and match the different methods of constructing arrays! For example

`c=[4,a(1:5),(i,i=1,4),7]`

### Important aside

When working with large matrices, looping over all elements can be a time-consuming process, and often leads to **bottlenecks**. If you can, only loop over indices or array sections that you know will have specific values you must set. This helps to reduce the number of loops needed, and thus the number of operations performed by the program.

For example, a loop over `i`, `j`, and `k` for `if (condition) A(i,j,k)` could be replaced by a loop over only two variables, if there exists a *dependence* between the looping variables.

### 2.6.6 Array Operations

All the mathematical operators discussed in Sect. 2.2 apply between arrays in an element-by-element fashion, whilst scalars operating on an array apply to every element. Note that when dealing with an expression involving several arrays, all arrays must be **conformal** (i.e. have the same shape):

```
real :: x, a(5),b(5),c(2,3),d(2,4)
```

```
! some array on array operations:  
a = a + b           ! element by element addition  
c = c * d(:, :3)   ! element by element multiplication  
c(:, 1) / b(1:3:2) ! element by element division  
! Scalar on array operations:  
a = (2*a)/3.       ! every element multiplied by 2/3.  
c = c + x          ! add variable x to every element in c
```

In addition to the mathematical operators, arrays can also be compared for equality, element by element, by using the `==` relational operator.

### 2.6.7 Intrinsic Array Functions

Many of the intrinsic functions we saw back in Sect. 2.5 accept array variables as arguments, with the function then **broadcast** over all elements in the array (although the array must be the correct data type – since `sqrt` only accepts **real** or **complex** arguments, you cannot apply it to an **integer** array).

In addition, there are intrinsic functions that *only* accept arrays as arguments – these fall into several categories: **vector and matrix mathematics**, **array reduction** (return a single value based on the elements in the array), **inquiry** (returns information based on the array itself), **manipulation and construction** (alters arrays), and **array location** (returns indices).

#### Important note

Fortran uses **column-major order** when storing arrays in memory.

For example, the matrix  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$  is stored in contiguous memory by

Fortran as **1 4 2 5 3 6**, whereas a row-major ordered language (such as C) would store it as **1 2 3 4 5 6**. This has implications in optimisation, where iterating over elements in column order can be orders of magnitude faster than iterating row by row in Fortran.

It must also be kept in mind when applying array functions; `dim=1` will refer to the *column* dimension, not rows; further, functions like `reshape` and `unpack` will apply sequentially down columns rather than across rows.

A note on some of the conventions used in this section:

**Required arguments** are capitalised

**Optional arguments** are given in lowercase

**Masks** are **logical** vectors or arrays, that can be created manually or via an array conditional statement. For example, if `a=[0.5, -1, -2]`, then the condition `a<0` creates the mask

```
[.false., .true., .true.]
```

If an optional mask is provided, the function will only apply to the array elements that correspond to a `.true.` element in the mask.

Hence, masks must be the same shape as any provided arrays.

**Dimensions** can also sometimes be specified, via the optional `dim` argument. By passing an integer corresponding to a particular dimension, the function will only be applied to the elements in that dimension.

For example, if `a(2, 2)` is defined as

```
a(1,:)=[1, 2]
```

```
a(2,:)=[3, 4]
```

then

```
maxval(a)=4
```

```
maxval(a, dim=1)=[3, 4]
```

```
maxval(a, dim=2)=[2, 4]
```

(i.e. max value, max value *per column*, and max value *per row* respectively).

If `dim` is not specified, then all dimensions are considered – the function is applied to *all* elements.

Function	Returns	Description
<code>dot_product(a,b)</code>	vector	For vectors $a$ and $b$ , $a \cdot b$ . If both arguments are complex arrays, this returns $a^* \cdot b$
<code>matmul(a,b)</code>	array	matrix product of arrays $a$ and $b$ , which must be compatible

**Table 2.14** Vector and matrix functions

Function	Returns	Description
<code>maxloc(ARRAY,mask)</code>	<code>integer</code>	returns index of largest element, subject to optional <code>mask</code>
<code>minloc(ARRAY,mask)</code>	<code>integer</code>	returns index of smallest element, satisfying optional <code>mask</code> . For example, <code>maxloc(a,a&gt;0)</code>

**Table 2.15** Array location functions

Function	Returns	Description
<code>all(MASK,dim)</code>	<code>logical</code>	indicates whether the condition is true for all elements. E.g. <code>all(a&gt;5)</code> is .true. if all elements are $>5$
<code>any(MASK,dim)</code>	<code>logical</code>	indicates whether the condition is true for any element. E.g. <code>any(a&gt;5)</code> is .true. if any element is $>5$
<code>count(MASK,dim)</code>	<code>integer</code>	counts number of elements that satisfy the condition
<code>maxval(ARRAY, dim,mask)</code>	<code>real(8)</code> or <code>integer</code>	returns the maximum value of an array along an optional dimension, or of those elements satisfying an optional condition
<code>minval(ARRAY, dim,mask)</code>	<code>real(8)</code> or <code>integer</code>	returns the minimum value of an array along an optional dimension, or of those elements satisfying an optional condition. E.g. <code>minval(a,mask=a&gt;0)</code> returns minimum value out of the positive elements of $a$
<code>product(ARRAY, dim,mask)</code>	<code>real(8)</code> or <code>integer</code>	returns the product of all array elements along an optional dimension, or of those elements satisfying an optional condition
<code>sum(ARRAY, dim,mask)</code>	<code>real(8)</code> or <code>integer</code>	returns the sum of all array elements along an optional dimension, or of those elements satisfying an optional condition. E.g. <code>sum(a)</code> sums all elements of $a$

**Table 2.16** Array reduction functions

Function	Returns	Description
<code>lbound(ARRAY, dim)</code>	vector or <b>integer</b>	returns the lower bound index (i.e. starting index <code>istart</code> ) for each dimension <i>or</i> for a particular dimension. E.g. for <code>a(9, 2 : 4)</code> , <code>lbound(a)</code> returns <b>[1, 2]</b>
<code>ubound(ARRAY, dim)</code>	vector or <b>integer</b>	returns the upper bound index (i.e. ending index <code>iend</code> ) for each dimension <i>or</i> for a particular dimension
<code>size(ARRAY, dim)</code>	<b>integer</b>	returns the number of elements in the array
<code>shape(ARRAY)</code>	<b>integer</b> vector	returns array shape (length of each dimension)
<code>allocated(ARRAY)</code>	<b>logical</b>	returns <code>.true.</code> if the array is allocated

**Table 2.17** Array inquiry functions

Function	Returns	Description
<code>cshift(ARRAY, SHIFT, dim)</code>	array	shifts each element along dimension <code>dim</code> in a <i>circular</i> fashion, by <code>SHIFT</code> positions left if <code>SHIFT &gt; 0</code> or <code>SHIFT</code> positions right if <code>SHIFT &lt; 0</code> . By default <code>dim=1</code>
<code>eoshift(ARRAY, SHIFT, bval, dim)</code>	array	similar to <code>cshift</code> , but instead shifted edge values come from the <code>bval</code> array (optional – by default the boundary has value <code>0</code> or <code>.false.</code> )
<code>transpose(MATRIX)</code>	matrix	returns the transpose of a matrix (rank-2 array) by swapping the rows and columns
<code>reshape(SOURCE, SHAPE, pad, order)</code>	array	reshapes the source array according to the vector <code>shape</code> , which lists the size of each dimension. If no padding values are provided ( <code>pad</code> ), then we must have <code>size(SOURCE)==product(SHAPE)</code> . Optional parameter <code>order</code> is an <b>integer</b> vector, specifying the <i>order</i> with which the dimensions should be filled.
<code>merge(TSOURCE, FSOURCE, MASK)</code>	array	the returned array is constructed from <code>TSOURCE</code> elements where <code>MASK=.true.</code> , and <code>FSOURCE</code> elements where <code>MASK=.false.</code> . All three input arrays must be the same shape.
<code>pack(ARRAY, MASK, vector)</code>	vector	‘flattens’ the array into a vector, ignoring elements that do not satisfy the mask (i.e. where <code>MASK=.false.</code> ). If an optional vector is provided, the array values are ‘placed’ sequentially into the vector.
<code>unpack(VECTOR, MASK, ARRAY)</code>	array	places the elements of the vector that satisfy the mask (i.e. where <code>MASK=.true.</code> ) into the array, element by element.

**Table 2.18** Array manipulation and construction functions

**Example 2.17** Normalisation

Normalise the quantum mechanic state vector  $|\psi\rangle = |0\rangle + (1+2i)|1\rangle - 0.2|2\rangle$  using the relationship  $\langle\psi|\psi\rangle = \sum_i |\psi_i|^2 = 1$ , and then return the probability vector  $|\psi_j|^2 = |\langle j|\psi\rangle|^2$ .

**Solution:**

```
program normalise
    implicit none
    complex :: psi(3)
    real     :: N, P(3)

    psi = [(1.,0.), (0.,1.), (-0.2,0.)]

    N = 1. / sqrt(sum(abs(psi)**2)) ! normalisation constant
    psi = N*psi                      ! normalise psi
    P = abs(psi)**2                  ! probability vector

    write(*,'(3f8.3)')P
end program normalise
```

*Output:* 0.166 0.828 0.007

**Example 2.18** Array reshaping

The following Fortran program uses `reshape` to create a multidimensional array from a vector, and then outputs some properties of the new array.

```
program reshapetest
    implicit none
    integer :: i, a(10), b(2,5), c(5,2)

    a = [(i,i=1,10)]

    ! reshape a to a 2x5 matrix, using default
    ! Fortran ordering (i.e. column-ordered)
    write(*,'(a)')'column-ordered reshape:'
    b = reshape(a,[2,5])
    do i=1,ubound(b,1)
        write(*,'(5i3)')b(i,:)
    end do

    ! reshape a to a 2x5 matrix, using row-ordering
    ! i.e. we direct the array to be reshaped starting
    ! from the second dimension
```

```
write(*,'(a)')'row-ordered reshape:'
b = reshape(a,[2,5],order=[2,1])
do i=1,ubound(b,1)
  write(*,'(5i3)')b(i,:)
end do

! some properties
write(*,10)'size: ',size(b)
write(*,10)'shape: ',shape(b)
write(*,10)'num. of elements in domain (1,5): ', &
  & count(1<b.and.b<5)
10 format(a,t40,2i2)

! transpose of b
write(*,'(a)')'transpose:'
c = transpose(b)
do i=1,ubound(c,1)
  write(*,'(2i3)')c(i,:)
end do
end program reshapatetest
```

*Output:*

```
column-ordered reshape:
1 3 5 7 9
2 4 6 8 10
row-ordered reshape:
1 2 3 4 5
6 7 8 9 10
size: 10
shape: 2 5
num. of elements in domain (1,5): 3
transpose:
1 6
2 7
3 8
4 9
5 10
```

### 2.6.8 Dynamic Arrays

Sometimes, the size of an array might not be known when we are declaring the array variables – for instance, the length of a vector containing a wavefunction depends on the grid size and spacing, which may require user or file input to be determined.

To get around this, we can use **dynamic storage allocation**; that is, rather than assign memory storage to the array variables via the compiler (**static storage allocation**), we allocate memory storage whilst the program is running. To do this, we must use the **allocatable** specifier when declaring the variables:

```
integer          :: N, ierr
real            :: a(2,10)
real, allocatable :: b(:, :)
```

In this example, **a** is a rank-2 array with memory allocated statically by the compiler, whereas **b** is a rank-2 array that has yet to have memory allocated. Let's assume the size of the array depends on the variable **N**:

**N = 3**

To dynamically allocate memory on program execution, we use the **allocate** statement:

**allocate(b(0:N-1, 2\*N))**

The array **b** now has shape **b(0:2, 6)**, and from here on can be used like any other array.

Once the array **b** is no longer needed, we can **deallocate** its memory allocation like so:

**deallocate(b)**

This frees the memory that was previously used to store the allocatable array, which may be required when working with particularly memory intensive code.

### Exception Handling

Occasionally, you may run into the issue where not enough memory exists to allocate an array of a particular size. The **allocate** and **deallocate** statements can return the status of the memory allocation process, providing a way to easily identify and resolve such issues:

```
allocate(b(N,N), stat=ierr)
if (ierr /= 0) stop "Error: Memory allocation failed"
```

The returned integer **ierr** has value **0** only if the memory allocation was successful; otherwise, the program terminates (via the **stop** statement) with a useful error message.

- ▶ Note that whilst we can avoid specifying the bounds of the allocatable array at declaration, we still have to indicate the number of dimensions or **rank**.

- ▶ Deallocation of array occurs automatically when the array variable goes **out of scope**, that is, the program/subroutine/function containing the array ends. However, it is good practice to always use the **deallocate** statement.

## 2.7 Procedures: Functions and Subroutines

Sometimes, a calculation must be computed numerous times in the same program. In order to increase code readability and maintainability, we can define custom **functions** and **subroutines** – together, these are known as **procedures**. Here, we will discuss the various approaches available for user-defined procedures in Fortran.

► **Functions** are used when we need to perform a calculation or sub-procedure numerous times that produces a single result.

### 2.7.1 External Functions

External functions occur at the end of the main program procedure, after the line containing the **end program** statement. External functions have the following structure:

```
program main
    implicit none
    real :: myFunc
    ! program code
end program main

function myFunc(x,y)
    implicit none
    real, intent(in) :: x, y
    real             :: myFunc
    ! additional function code
    myFunc = ! provide a return value
end function myFunc
```

Some things to note:

- As the function is placed after the end of the main program, it needs its own **implicit none** statement
- The data type of the function itself must be declared – in this case, **myFunc** returns a value with data type **real**
- **In addition**, the external function must *also* be declared in the main program, otherwise it will not know the correct data type of the function
- The dummy argument variables **x** and **y**, have the **intent(in)** attribute; this means that they *cannot* be changed inside the function – the presence of an assignment **x=** or **y=** will result in a compilation error.
- The function can contain as many lines of code as required, however at least one line must include the assignment **myFunc=**, which is the value that gets *returned* when the function is called.

► In contrast, **subroutines** are used when we need to repeat a block of code numerous times, with the possibility of returning no result or multiple results by directly altering the provided arguments.

Rather than using the function name to return the result, you can alternatively use the **result** statement:

```
function myFunc(x,y) result(f)
```

Then, instead of using the function name, the result is then returned in variable **f**:

```
real :: f
f = ! return value
```

Note that this is just an alternative **syntax**, either method of specifying the return value is fine.

**Keyword arguments**

Functions can also be called from the main program using keyword arguments, for example instead of

```
myFunc(5.,9.)
```

we can use

```
myFunc(x=5.,y=9.)
```

Keyword arguments also allow arguments to be passed in *any* order:

```
myFunc(y=9.,x=5.)
```

- And finally, the function is called in the main program just like any other intrinsic function: `myFunc(x,y)`

**Example 2.19** External normalisation function

Write an external function that calculates the norm of a state vector

$$\langle \psi | \psi \rangle = \sum_i |\psi_i|^2$$

and use it to normalise the two-basis state vector  $|\psi\rangle = |0\rangle + i|1\rangle$ .

**Solution:**

```
program main
    implicit none

    complex :: psi(2)
    real     :: N, normEF

    psi = [(1.,0.),(0.,1.)]

    ! normalisation constant
    N = 1./sqrt(normEF(psi,size(psi)))
    write(*,*)N*psi
end program main

function normEF(psi,n)
    implicit none
    integer, intent(in) :: n
    complex, intent(in) :: psi(n)
    real                 :: normEF

    normEF = sum(abs(psi)**2)
end function normEF
```

**Output:** ( 0.707106769 , 0.00000000 ) ( 0.00000000 , 0.707106769 )

### 2.7.2 External Subroutines

Perhaps instead of returning a single value, we would like to manipulate multiple variables, or maybe even return *no* result and just re-run a block of code for various inputs. In this case, we can use **external subroutines**. These have a very similar structure to external functions:

```
program main
    implicit none
    ! program code
```

```

end program main

subroutine mysub(x,y,z)
  implicit none
  real, intent(in) :: x, y
  real, intent(out) :: z

  ! subroutine code
end function mysub

```

Some things to note:

- **functions** are used to return a single result without changing input variables, so we generally only use **intent(in)**. In contrast, subroutines allow us to define three types of arguments (Table 2.19). To reduce unintended variable modifications, it is highly recommended to always specify the variable intent.

- ▶ You can also define subroutines that accept *no* arguments – this is useful if you have blocks of code that need to be repeated with no changes based on the program state.

<b>intent(in)</b>	the variable <i>cannot</i> be changed by the subroutine. The argument can be a literal <i>or</i> a variable
<b>intent(out)</b>	the subroutine modifies and returns the value of this variable, whilst ignoring its initial value – the argument <i>must</i> be a variable
<b>intent(inout)</b>	the variable passes an initial value to the subroutine, which can then be modified and returned by the subroutine – the argument <i>must</i> be a variable. This is the <b>default</b> behaviour if intent is not stated.

**Table 2.19** Variable intent

- Again, as external procedures occur after the end of the main program, it needs its own **implicit none** statement
- *Unlike* external functions, the external subroutine does *not* need to be declared in the main function
- The subroutine is called in the main program via the **call** statement; for this example, we would use

```
call mysub(x,y,z)
```

where **x**, **y**, and **z** are **real** variables declared in the main program. Note that the **call** statement must occur on its own line, with nothing preceding or following it; after calling the subroutine, the variable **z** (defined with **intent(out)** in **mysub**) will now contain the result.

**Example 2.20** External normalisation subroutine

Write an external subroutine that normalises a state vector *and* returns the normalisation constant. Apply it to the state vector  $|\psi\rangle = |0\rangle + i|1\rangle$ .

**Solution:**

```
program main
    implicit none

    complex :: psi(2)
    real     :: N, NC

    psi = [(1.,0.),(0.,1.)]

    call normalize(psi,size(psi),NC) ! call the subroutine
    write(*,*)psi ! write the normalised wavefunction
    write(*,*)NC ! write the normalisation constant
end program main

subroutine normalize(psi,n,NC)
    implicit none
    integer, intent(in)   :: n
    complex, intent(inout) :: psi(n)
    real, intent(out)     :: NC

    NC = 1./sqrt(sum(abs(psi)**2))
    psi = NC*psi
end subroutine normalize
```

**Output:**

```
( 0.707106769      ,  0.00000000 ) (  0.00000000      ,  0.707106769  )}
```

### 2.7.3 Internal Procedures

- The **contains** statement must always occur at the **end** of the program.

```
program main
    implicit none
    ! variable declarations
    ! program code

contains
    function myInternalFunc(x,y)
        ! internal function code
```

### 2.8 Summary

```

end function

subroutine myInternalSub(x,y,z)
    ! internal function code
end myInternalSub
end program main

```

The same rules apply when working with internal procedures or external procedures, however with some slight differences involving **scope**, the visibility of a procedure or variable to other procedures.

- The scope of an internal procedure is the program in which it is declared – so for instance, an internal function can only be called by the program that contains it. This differs from an external procedure, that can be called from any program.
- Similarly, the scope of a variable is the procedure in which it is declared. This means, rather than just manipulating **locally declared variables**, internal procedures can also manipulate and access **global variables**, that is variables defined in the main program.

For example, consider the following:

```

program main
    implicit none
    ! global variables
    integer :: a

    a = 2; write(*,*)myFunc(7) ! gives 29
    a = 1; write(*,*)myFunc(7) ! gives 8(!!)

contains
    function myFunc(x)
        integer, intent(in) :: x
        integer :: myFunc

        !local variables
        integer :: y

        y = (a**2) * x
        myFunc = y + 1
    end function
end program main

```

In this example, **x** is a **dummy argument** accepted by the internal function **myFunc**, and **y** is a **local variable** whose scope is **myFunc** – it cannot be accessed by the main program.

Note that a variable declared in an internal procedure is *always* a local variable, even if it has the same name as a global variable. In this case, if we define **a** as a local variable to **myFunc**,

```

integer :: y, a
a = 1

```

then this **a** is *different* from the global **a**, and any changes to **a** in **myFunc** will not affect **a** in the main program.

In contrast, `a` is a **global variable** – it is accessible to both the main program (where it is declared) *as well as* any contained procedures, in this case `myFunc`. If we allow `myFunc` to depend on `a`, then changing the value of `a` in the main program will change the value of `myFunc`, *even when called with the same arguments*.

Furthermore, internal procedures can even *change* the values of global variables – leading to side effects and code which may be hard to maintain. **Always try and avoid using global variables in internal procedures!**

#### 2.7.4 Advanced Procedure Features

##### Optional Arguments

Using the **optional** attribute, arguments can be made optional, i.e. may or may not be passed from the main program. For example,

```
function myFunc(x,y,z)
    implicit none
    real, intent(in)          :: x, y
    logical, intent(in), optional :: z
    real :: myFunc
    ! function code
end function myFunc
```

The argument `z` is now optional, and we can call the function with either two (`myFunc(0.,2.)`) or three (`myFunc(0.,2.,.true.)`) arguments. To set the *default* value of any optional arguments, we can then use the **present** intrinsic function to check if it was passed or not:

```
logical :: temp_z
if (present(z)) then
    temp_z = z
else
    temp_z = .false.
end if
```

Note that we are using a new, temporary local variable `temp_z` to store the default or provided value of `z`; `temp_z` must therefore be declared as a local variable within the functions declaration block. The reason we must do this is because `z` has attribute **intent(in)** – we cannot assign it a new value!

You might be wondering, what happens if an optional argument is nested in between two compulsory arguments? Try and avoid this if you can – stick to the convention of placing all optional arguments after compulsory arguments. However, if for some reason you *must* break this rule, you can always use keyword arguments; for example, if `x` and `z` are now compulsory, and `y` is optional, then we can use

```
myFunc(x=0., z=.false.)
```

### Assumed-Shape Array Arguments

In Example 2.19, we passed an array to a function. Note that the function `normEF` accepted *two* arguments – the array itself, and an integer specifying the size of the array. This was needed so that we could declare the function dummy variable `psi(n)`. To avoid doing this, we can use what is called **assumed-shape arrays** – that is we don't declare the dummy array sizes, instead they are 'assumed' (and calculated exactly) when an array is passed to the function.

Assumed-shape arrays have multiple advantages over explicit-shape arrays, for example:

- No array copying is done between the main program and the procedure
- We can pass array strides directly to a procedure
- The shape and size of the array is passed automatically to the procedure, without the need to specify them as separate arguments
- The compiler automatically checks the shape at compilation, producing errors if there are mismatches

As such, assumed-shape arrays should always be used in **subroutines**, where possible, over explicit-shape arrays .

To use an assumed-shape array dummy argument, simply declare it within the function **with no upper-bound** in any of its dimensions; for example, `psi(2:)` or `psi(:)`. In the former case, the elements of a size  $n$  vector argument will be indexed by the function as

`[psi(2),psi(3),...,psi(n+1)]`

whereas in the latter case, the default lower bound of the assumed array will be 1:

`[psi(1),psi(2),...,psi(n)]`

**Example 2.21** Internal normalisation function with assumed-shape arrays

Write an external function that calculates the norm of a state vector  $\langle \psi | \psi \rangle = \sum_i |\psi_i|^2$  and use it to normalise the two-basis state vector  $|\psi\rangle = |0\rangle + i|1\rangle$ . The external function should accept an assumed-shape array.

**Solution:**

```
program main
    implicit none

    complex :: psi(2)
```

You can also use **assumed-length characters** as function or subroutine arguments. These are specified inside the procedure using the declarations

`character(len=*)`

or equivalently

`character(*)`

which allows the procedure to accept a string of arbitrary length characters.

► This generalises to the multi-dimensional arrays – for example,

`integer :: y(:,:,:,:)`

is an assumed-shape array argument of rank 3.

```

real      :: N

! program code
psi = [(1.,0.),(0.,1.)]
N = 1./sqrt(normEF(psi)) ! Normalisation constant
write(*,*)N*psi

contains
  function normEF(psi)
    complex, intent(in) :: psi(:)
    real                 :: normEF

    normEF = sum(abs(psi)**2)
  end function normEF
end program main

```

### Assumed Shape Arrays and External Procedures

In Example 2.21, you can see an example of an internal function that uses assumed-shape array arguments. What about external procedures? Let's modify Example 2.19 to use assumed-shape arrays – the external function becomes

```

function normEF(psi)
  implicit none
  complex, intent(in) :: psi(:)
  real                :: normEF
  normEF = sum(abs(psi)**2)
end function normEF

```

and now only accepts *one* argument, the state vector itself. However, if you try and compile it, you'll get the following error:

```
Error: Procedure 'normef' at (1) with assumed-shape
dummy argument 'psi' must have an explicit interface
```

- ▶ As external procedures cannot use modern Fortran 90 features such as assumed shape arrays and optional arguments without defining an interface to the main program, it is always better to use an **internal procedure**.

What's going on? With our programs so far, the compiler has been able to deduce all the details of the external procedures automatically, and create an **implicit interface** between the main program and the external procedure. Unfortunately, this is not possible with external procedures using assumed-shape arrays or optional arguments – instead, we need to write an **explicit interface**.

### Interfaces

In order to use assumed-shape array arguments and optional keywords with **external procedures**, we *need* to include an **explicit interface** in the main

program – this allows the compiler to easily identify the variable and return data types.

Interfaces are placed inside an **interface block** before variable declarations:

```
interface
    function myExternalFunc(x,y)
        real, intent(in) :: x,y
        real              :: myExternalFunc
    end function myExternalFunc
    subroutine myExternalSub(x,y,z)
        real, intent(in)  :: x,y
        real, intent(out) :: z
    end subroutine myExternalSub
end interface
```

- ▶ If you have multiple external functions, you can place them all in the same interface block.

Within the interface block, we describe the external procedure (whether it is a function or subroutine), its name, dummy variables, and data types, using a syntax similar to external functions. However, note that we don't replicate the procedure code – this still takes place in the external procedure after the main program code – we are just providing the main program with more information about data types and arguments. See Example 2.22 for an example of an explicit interface.

Note that maintaining explicit interfaces to external procedures adds complexity to the Fortran code – if you make changes to an external procedure, not only do you have to sync these changes across all function calls, but you also need to make sure you update the interface.

Thus, comparing external and internal procedures in Fortran, **internal procedures are preferred** – they allow us to utilize many of the advanced features of Fortran 90 (optional arguments, assumed-shape arrays) whilst avoiding the need to use interfaces. However, for more complicated programs where you would prefer to split your code amongst various files, or perhaps use the same function across multiple programs, you'll need to use modules, which will be discussed in the next section.

- ▶ Always use internal procedures or modules rather than external procedures!

**Example 2.22** External normalisation function with assumed-shape arrays

Write an external function that calculates the norm of a state vector  $\langle \psi | \psi \rangle = \sum_i |\psi_i|^2$  and use it to normalise the two-basis state vector  $|\psi\rangle = |0\rangle + i|1\rangle$ . The external function should accept an assumed-shape array.

**Solution:**

```
program main
    implicit none

    ! interface block as we have an assumed-shape array
    interface
        function normEF(psi)
            complex, intent(in) :: psi(:)
            real                 :: normEF
        end function normEF
    end interface

    complex :: psi(2)
    real    :: N

    ! program code
    psi = [(1.,0.),(0.,1.)]
    N = 1./sqrt(normEF(psi)) ! Normalisation constant
    write(*,*)N*psi
end program main

function normEF(psi)
    implicit none
    complex, intent(in) :: psi(:) ! assumed-shape array
    real                 :: normEF

    normEF = sum(abs(psi)**2)
end function normEF
```

## Overloading

Perhaps you have a function `sfunc` that accepts a `real` variable and returns a `real` result – what if you want the same function to also apply to and return double precision `real(8)` values? As Fortran requires each function and argument to have a single, explicit data type, a new function would have to be written; let's call it `ifunc`. However, we can use a process called **overloading** to define a **generic** name for these two functions using an `interface` block.

This takes several forms depending on the structure of the program and subprocedures. For **external procedures**, the overloading interface needs to be coupled with an explicit interface of the individual functions:

```
interface func
    function sfunc(x)
        real, intent(in) :: x
        real              :: sfunc
    end function sfunc
    function dfunc(x)
        real(8), intent(in) :: x
        real(8)             :: dfunc
    end function dfunc
end interface func
```

For an **internal procedures**, explicit interfaces are not necessary, and the process of overloading is greatly simplified:

```
interface func
    procedure sfunc, dfunc, ...
end interface func
```

Now, in our main program we simply use the function `func` – depending on whether the argument is single or double precision, compiler automatically works out which function to call!

**Example 2.23** Internal function overloading

Write a generic function that converts Fahrenheit into Celsius for both single and double precision arguments.

**Solution:**

```
program main
  implicit none

  interface FtoC
    procedure dFtoC, sFtoC
  end interface FtoC

  ! Pass a single precision argument
  write(*,*)"Returned data type: ",kind(FtoC(33.))
  ! Pass a double precision argument
  write(*,*)"Returned data type: ",kind(FtoC(33.d0))

  contains
    function sFtoC(f) result(c)
      real, intent(in) :: f
      real :: c
      c = 5.*(f-32.)/9.
    end function sFtoC
    function dFtoC(f) result(c)
      real(8), intent(in) :: f
      real(8) :: c
      c = 5.d0*(f-32.d0)/9.d0
    end function dFtoC
end program main
```

**Output:**

Returned data type:	4
Returned data type:	8

### Passing Functions to Procedures

Sometimes it is useful to have the ability for our procedures to accept functions as dummy arguments themselves. For example, a function that approximates the derivative  $f'(x)$  at a particular point could have 3 arguments; the function to differentiate  $f$ , the point  $x = x_0$  at which to differentiate, and the step size  $\Delta x$ :

```

1  program test
2    implicit none
3
4    write(*,*)finiteDiff(f,2.5,0.01)
5
6    contains
7      function f(x)
8        real, intent(in) :: x
9        real              :: f
10
11     f = x**2
12   end function f
13
14   function finiteDiff(func,x,dx)
15     real, intent(in)  :: x, dx
16     real              :: finiteDiff
17
18   interface
19     function func(x)
20       real, intent(in) :: x
21       real              :: func
22     end function func
23   end interface
24
25   finiteDiff = (func(x+dx)-func(x-dx))/(2*dx)
26   end function finiteDiff
27 end program test

```

The main difference you can see is that, rather than declare the ‘dummy’ function `func` as we would normally do for a dummy argument, we must instead define an `interface` within the procedure itself. This enables our invoked function `finiteDiff` to ‘see’ that `func` is a `function` that accepts one `real` argument and returns a `real` result.

Aside from this one modification, the rest of the code looks as expected. We have also defined the function  $f(x) = x^2$ , and the `finiteDiff` function is used to differentiate  $f(x)$  at  $x = 2.5$  and with  $\Delta x = 0.01$ .

Try implementing this code yourself. How accurate is the result of the centered finite difference function?

## Recursive Procedures

Suppose that we have a procedure that requires a *variable* number of **do** loops – that is, the number of **do** loops required to return/achieve the correct result depends on the input provided. In some cases, we could implement this using a **do while** construct, or a general **do** loop with **exit** and **cycle** statements.

Another alternative is **recursive procedures**. Recursive functions or subroutines are indicated to the compiler by using the **recursive function** or **recursive subroutine** statements, respectively.

### Direct Recursion

- ▶ A word of warning: there is a slight computational overhead every time a recursive function or subroutine calls itself, which can often cause recursive functions to be **slower** than the equivalent do loop implementation.

However, some problems are more naturally suited for a recursive approach, leading to clearer and more maintainable code.

#### Example 2.24

Write a recursive function and a recursive subroutine that calculates the factorial of an integer argument, and output the results of 4!.

**Solution:**

```
program main
    implicit none

    integer :: n, res

    n = 4

    write(*,*)"Recursive function: 4!=",fact1(n)

    call fact2(n,res)
    write(*,*)"Recursive subroutine: 4!=",res

contains

    ! for a recursive function, we must use the
    ! result statement, and not the functions name
    recursive function fact1(n) result(res)
        integer, intent(in) :: n
        integer             :: res

        if (n==1) then
            res = 1
        else
            res = n*fact1(n-1) ! invoke the function
        endif
    end function fact1
```

```
recursive subroutine fact2(n,res)
    integer, intent(in) :: n
    integer, intent(out) :: res

    if (n==1) then
        res = 1
    else
        call fact2(n-1,res) ! invoke the subroutine
        res = n*res
    endif
end subroutine fact2
end program main
```

**Output:**

```
Recursive function: 4!=24
Recursive subroutine: 4!=24
```

Note that in Example 2.24, we used the **result** statement rather than the functions name to return the result. This is *required* for directly recursive functions, as the function name instead is used to invoke the new instance of the function.

### Indirect Recursion

Indirect recursion occurs when a **function** or **subroutine** calls itself indirectly, via an intermediate procedure. For example, consider two functions *A* and *B*. If *A* calls *B*, which then calls *A*, which in turn calls *B*, and so on, we have an example of indirect (or *mutual*) recursion.

## 2.8 Modules

So far, we have considered two ways of structuring programs and procedures:

- **External procedures**

- Introduced in the FORTRAN 77 standard
- These occur *outside* the scope of the main program – the interface is *implicit* and is guessed by the compiler.
- To use modern Fortran 90 features (such as optional arguments, assumed-shape arrays, overloading), an *explicit interface* must be added manually to the main program.

- **Internal procedures**

- Introduced in Fortran 90
- These occur *within* the scope of the main program using the **contains** statement, and the explicit interface is added automatically by the compiler.
- Can use modern Fortran 90 features (such as optional arguments, assumed-shape arrays, overloading) without using an interface block.

The main issue in using external procedures with explicit interfaces is the duplication of code in two different places – this can make code maintenance difficult and unwieldy, as changes in external procedures will need to be repeated in their respective interfaces. To alleviate this issue, internal procedures were introduced in Fortran 90, and these should always be used in preference to external procedures.

However, even working with internal procedures can become unwieldy as the size of the program grows. Good programming practice dictates we split the main program and procedures amongst various files, and this has the added benefit of allowing us to use the same functions/subroutines across multiple programs.

The solution to this, provided by Fortran 90, are **modules**; these are procedures that are *separate* from the main program, can be placed in their own files, and can contain

- sub-procedures such as functions and subroutines
- interface blocks
- global variables and parameters

amongst other things. Most importantly, the compiler can *automatically create an explicit interface* between modules and Fortran programs. Thus, **always use modules!**

► It is good practice to get into the habit of **always** using modules for storing sub-procedures and common parameters!

### 2.8.1 Module Structure

Modules have the following structure:

```
module modulename
  ! [load other dependent modules]
  implicit none
  private

  ! SPECIFICATION PART
  ! [interfaces]

  ! [declare global variables and parameters]
  ! [declare private variables and parameters]

  ! [list public module procedures]

contains
  ! [module procedures (functions and subroutines)]
end module modulename
```

Note that, like a program, modules

- require the **implicit none** statement to ensure all variables used are declared
- utilize a similar form for defining interfaces, declaring variables, and placing executable statements
- use the **contains** statement (directly preceding the **end module** statement) to contain provided module procedures.

However, there are some deviations – most importantly, the distinction between public and private module entities.

### 2.8.2 Privacy and Accessibility

By default, all entities in a module are **public**; that is, they can be accessed by any other module or program that imports the module. This goes against good programming practice, as we are making visible even just locally required variables and procedures, which may lead to conflicts between modules and difficult to maintain code.

To avoid this, we use the **private** statement at the beginning of the module, after the **implicit none** statement. This instructs the compiler to keep all entities **private** to the module, *unless otherwise specified*. Treat this statement like you would **implicit none** – get into the habit of automatically including it at the beginning of modules without even thinking about it!

► An **entity** refers to variables, parameters, functions, or subroutines.

That is, any 'physical entity' that can be referenced by name.

To specify module procedures that we would like to be accessible to programs using the module, we use the **public** statement. This has a similar syntax to variable declaration:

```
public :: func1, func2, subroutine1, ...
```

### 2.8.3 Global Variables

Since we have declared the whole module **private**, variables declared in a regular fashion are *automatically* private. Note that they are still global with respect to the module itself (they are visible to every entity *within* the module, for example module procedures), but cannot be accessed by anything *outside* the module.

To make a variable **public**, we could add it to the list of public modules we discussed already. However, this makes it difficult to easily look over your code and distinguish between private and public variables. Instead, it is better practice to maintain two separate sections of variable declarations – one using the **public** attribute, and one not:

```
! public variables
real, public :: x, y
integer, public :: N

! private variables
real :: tmp1, tmp2
```

Have a look at Example 2.25 to see a working example of the differences between public, private, global and local variables.

#### Example 2.25 Global variables

```
1  module globaltest
2    implicit none
3    private
4
5    ! public variables
6    integer, public :: N = 1
7    ! private variables
8    integer :: M = 2
9    ! public procedures
10   public :: writeNM
11
12  contains
13    subroutine writeNM
14      integer :: L
15      write(*, '(a,i1,a,i1)')"N=",N," M=",M
```

```

16    end subroutine writeNM
17 end module globaltest
18
19 program main
20   use globaltest
21   implicit none
22   integer :: M = 6
23
24   call writeNM()
25   N = 5
26   call writeNM()
27 end program main

```

**Output:**

```
N=1 M=2
N=5 M=2
```

A couple of things to note from this example:

- N is a **public, global** variable of module **globaltest**.
  - **Visible** to all module procedures, as well as the main program.
  - When a program or procedure changes its value, it changes *everywhere*, i.e. for all programs, functions and subroutines. Compare lines 24–26.
  - If you attempt to define a separate variable **also called N**, inside of program **main**, you will run into a **conflict error**:

```
Error: Symbol 'N' at (1) conflicts with symbol
from module 'globaltest'
```

This is because the variable N is already associated with the public variable of the same name imported from the **globaltest** module.

- M is a **private, global** variable of module **globaltest**
  - **Visible** to all module procedures, but *not* the main program importing the module
  - Any attempt to access it from the main program will produce the compilation error
 

```
Error: Symbol 'M' at (1) has no IMPLICIT type
```

 since, as far as the compiler is concerned, no variable M exists within the scope of the program.
  - We can therefore declare M as a variable within program **main** without a conflict error. This new variable is accessible *only* to program **main**, and is a completely separate variable from the private variable M in module **globaltest**

- `L` is a **private**, **local** variable of module subroutine `writeNM`

#### 2.8.4 Parameters

Perhaps you would like to use a module to easily provide a set of often-used parameters; this would help reduce clutter and increase maintainability, by placing all constants in one place rather than being strewn around various programs and procedures. How might you do this?

Public variables! I hear you shout. How can you forget, you just told us about them! While this appears a decent solution, the issue with public variables is that it is possible to accidentally change their value in one small portion of the code, thereafter affecting its value *everywhere*.

To avoid this, we can declare public **constants**, using the **parameter** attribute inside our parameter module:

```
module parms
    implicit none
    private

    real, public, parameter :: PI = 3.14159
    integer, public, parameter :: N = 152
end module parms
```

If this module is now imported into another program or module, you will be able to access these variables, but attempting to change their value will result in a compilation error. Exactly what we wanted!

#### 2.8.5 Custom Kinds

##### • `kind()`

Returns the precision information of a literal or variable.

##### • `selected_real_kind(d,e)`

Returns the kind value of a real data type with decimal precision of `d` digits, and an optional exponent range of `e`.

##### • `selected_int_kind(R)`

Returns the kind value of the smallest integer type that can represent all values ranging from  $-10^R$  (exclusive) to  $10^R$  (exclusive).

Another important use of module parameters is the ability to define our own floating point data types (or **kinds**) with custom precision. This makes changing the precision of **real** variables across our code faster and more efficient. For instant, consider the module `mykinds`.

##### `mykinds.f90`

```
module mykinds
    implicit none
    integer, parameter :: SP=kind(1.0) !Single precision
    integer, parameter :: DP=kind(1.0d0) !Double Precision
    ! Quad Precision
    integer, parameter :: QP=selected_real_kind(33, 4931)
    public SP, DP, QP
end module mykinds
```

**Table 2.20** Intrinsic kind functions. The result of these functions are sometimes **processor dependent**, and thus these provide a safe way of guaranteeing a particular precision.

Rather than defining floating point variables using the old fashioned `real`, `real(4)`, and `real(8)`, we can now use our custom kinds, like so:

```
program main
  use mykinds
  implicit none

  real(DP) :: x
  real(QP) :: y

  x = 1.0_DP
  y = 1.0_QP
end program main
```

Note that, in addition to slightly modifying how we declare our variables, we also have a new way of assigning values to our variables. Before, we would use `1.` or `1.d0` to represent a single precision and double precision floating point value respectively; now we use `1.0_XP` where `XP` is our custom kind parameter. This has two benefits:

- We now have a method of writing literals to any custom precision, not just single or double precision
- Our code is easier to maintain if we need to change precision types – we can either change the definition of the custom kind parameter in the module, or use find and replace in any text editor (for example, to change all instances of `DP` to `SP`).

Thus, as you can probably guess, from now on it is Good Practice<sup>TM</sup> to use the above method of defining custom kinds.

### 2.8.6 Overloading

Procedure overloading works almost identically to what we saw back in Sect. 2.7.4 with internal functions; however, we now use the statement `module` `procedures` within the interface block:

```
interface [genericfuncname]
  module procedure sfunc, rfunc, ...
end interface [genericfuncname]
```

#### Example 2.26 Module function overloading

Write a generic module function that converts Fahrenheit into Celsius for both single and double precision arguments, using the `mykinds` module defined in the previous section.

Solution:

```
module temperature
  use mykinds
  implicit none
  private

  interface FtoC
    module procedure dFtoC, sFtoC
  end interface FtoC

  public :: FtoC

  contains
    function sFtoC(f) result(c)
      real(SP), intent(in) :: f
      real(SP)             :: c
      c = 5.0_SP*(f-32.0_SP)/9.0_SP
    end function sFtoC

    function dFtoC(f) result(c)
      real(DP), intent(in) :: f
      real(DP)             :: c
      c = 5.0_DP*(f-32.0_DP)/9.0_DP
    end function dFtoC
end module temperature
```

### 2.8.7 Importing Modules

To import modules into programs, use the `use` statement. This is *always* the first statement of the code, and goes directly between the initial `program` statement and the `implicit none` statement. You can have multiple `use` statements here; each one importing a separate module. For example,

```
program main
  use mykinds
  use module1
  implicit none
  ! [program code]
end program main
```

Further, we can import modules into *other* modules, again using the `use` statement before the `implicit none` statement:

```
module module2
  use mykinds
  use module1
  implicit none
  ! [module code]
end module module2
```

When we import a module, we import *all* public variables, parameters, and procedures. If we only want to import one entity, or a small selection of entities, we can use the following syntax:

```
use module1, only : var1, var2, param1, func1, ...
```

### Conflicts and Renaming

You can run into compilation errors, known as **conflict errors**, if you import an entity from a module which has the same name as an entity (variable, procedure, or parameter) within the program itself. For example, the module might have a public parameter `N` which conflicts with another variable also called `N` in the main program.

These errors can be avoided with the `use` statement to ‘rename’ the module entity; this has the syntax

```
use module1, name_in_this_program => name_in_the_module
```

For example, we could rename the module parameter `N` to `Nmod` within the program, like so:

```
use module1, Nmod=>N
```

Now the program can access the module variable `N` using the specifier `Nmod`, and can continue using `N` for its own, local variable.

Furthermore, we can combine the renaming property with the `only` statement, like this:

```
use module1, only : var1, var2Mod=>var2, func1mod=>func1
```

### 2.8.8 Compiling Programs and Modules

When it comes to compiling your program and modules, there are several approaches that can be taken.

#### A Single File

If you only have one or two relatively small modules, you *may* include them in the same file as your main program (e.g. `main.f90`), preceding your program code. In that case, compilation is simply

```
gfortran main.f90 -o main
```

However, this defeats one of the main purposes of using modules – the ability to better organise your code and use the same module in several programs. Thus it is Good Practice<sup>TM</sup> to always use a separate `.f90` file for each module.

### Multiple Files

Let's say you have multiple modules, with filenames `kinds.f90`, `constants.f90`, `functions.f90`, and a main program `main.f90`. The modules `kinds` and `constants` are *independent* – they do not depend on or use other modules – whereas `functions` uses both these modules. The main program then uses all of these modules; in order to compile and link the program, we use the following command:

```
gfortran kinds.f90 constants.f90 functions.f90 main.f90 -o main
```

Whether the order of the source files matters or not depends on your compiler, however some are more finicky than others, and it is safer to always try and list modules that have no dependencies first, followed by those modules which only use the preceding modules, followed by the main program itself.

### Object Files and Linking

Sometimes, you may want to pre-compile modules and not worry about compiling it repetitively along with the main program. This can be done by compiling your modules to **object files** (with file extension `*.o`), and then linking them together later.

To individually compile your modules to an object file, you use the `-c` flag:

```
gfortran -c kinds.f90
gfortran -c constants.f90
gfortran -c functions.f90
```

This creates the files `kinds.o`, `constants.o` and `functions.o` (note that these are *not* executables, they cannot be run from the command line!).

Once you are ready to compile your main program and create an executable, you have two choices:

- (a) Compile your main program to an object file,

```
gfortran -c main.f90
```

and then **link** all your object files together:

```
gfortran kinds.o constants.o functions.o main.o -o main
```

- (b) Alternatively, you could do the compiling and linking in one step:

```
gfortran kinds.o constants.o functions.o main.f90 -o main
```

## 2.9 Command-Line Arguments

A great time saver when running numerical simulations is the ability to run the same program numerous times, however executed with differing options in the form of **command line arguments**. This is a great way to easily change the behaviour/input of a program without needing to recompile it, and is supported (since Fortran 2003) by the two intrinsic functions `command_argument_count()` and `get_command_argument()`.

---

### `command_argument_count()`

Returns the **integer** number of arguments passed on the command line when executing the program

---

### `call get_command_argument(N,ARG,length,status)`

- N: **integer, intent(in)** where  $N > 0$
- ARG: **character, intent(out)**
- length: (optional) **integer, intent(out)**
- status: (optional) **integer, intent(out)**

Retrieves the  $N$ th command line argument, storing it in string ARG. If the length of the command line argument is larger than the length of ARG, it is truncated to fit in ARG.

The optional length argument contains the length of the  $N$ th command line argument. If the argument retrieval fails,  $status > 0$ ; if ARG contains a truncated argument,  $status == -1$ ; otherwise  $status == 0$ .

---

**Table 2.21** Command line argument intrinsic functions and subroutines (Fortran 2003)

### Example 2.27 Position-valued command line arguments

Let's say we have a Fortran executable `argtest`, and we want to be able to pass some simple command line arguments. The simplest way to do this is by using **position-valued** arguments – that is, we differentiate the arguments based on their *positions* when passed to the executable.

For a simple example, let's consider the case of two arguments:

- an input filename (denoted by the variable `input`)
- an integer (denoted by the variable `N`)

The aim is to be able to call the program like this,

```
argtest data.txt 10
```

and have several things happen:

- (a) check that all arguments were read without error

► If your compiler does not support the Fortran 2003 standard, you can use the following Fortran 90 compatible intrinsics instead:

`iargc()` instead of `command_argument_count()`

`call getarg(N,ARG)` instead of `get_command_argument`

Note that the `getarg()` subroutine does *not* support length and status arguments.

- (b) check that an input file has been provided
- (c) check that the input file *exists*
- (d) and finally, if all the above checks have passed, set `N=10` and `input='data.txt'`.

**Solution:**

To begin with, let's write a module containing all our 'checking' subroutines; `PrintHelp()`, `checkArgStatus(ierr)`, and `CheckFileExists(filename)`.

► Note that the subroutine `checkFileExists()` uses an **assumed-length** character argument, `filename`; hence the declaration

`character(len=*)`

See Sect. 2.7.4 for more information.

```
module argfuncs
  implicit none
  contains
    ! print a short help message to the console
    subroutine PrintHelp()
      write(*,*)"usage: main [OPTIONS]"
      write(*,*)""
      write(*,*)"describe command line arguments here"
    end subroutine PrintHelp

    ! check that the argument was read
    ! correctly by get_command_argument(i,arg,status=ierr)
    subroutine checkArgStatus(ierr)
      integer, intent(in) :: ierr

      if (ierr>0) then
        write(*,*)"Error parsing argument"
        call PrintHelp()
        stop
      elseif (ierr==1) then
        ! argument truncation occurred!
        write(*,*)"Error: argument truncation occured"
        stop
      endif
    end subroutine checkArgStatus

    subroutine checkFileExists(filename)
      character(len=*), intent(in) :: filename
      logical :: stat

      inquire(file=filename,exist=stat)

      ! if the input file doesn't exist, quit
      if (.not.stat) then
        write(*,*)"Error: file ' &
          & //trim(filename) &
          & //' not found"
        stop
      endif
    end subroutine checkFileExists
end module argfuncs
```

Now that we have a module containing our ‘checking’ subroutines, we can write the main program:

```

program testargs1
use argfuncs
implicit none
character(len=50) :: arg, input
integer           :: i, ierr, N, Nargs
logical            :: debug, findN, findInput

! Set the default command line options
N = 10
! count number of arguments
Nargs = command_argument_count()
if (Nargs==0) stop "Error: must provide input file"

! loop over command line arguments
do i=1, Nargs
    ! store the ith argument in arg & check for errors
    call get_command_argument(i,arg,status=ierr)
    call checkArgStatus(ierr)

    ! adjust the argument to the left
    arg = adjustl(arg)

    ! choose action depending on argument position
    select case(i)
        case(1)
            ! 1st argument is the input filename
            input = arg
            ! check if the input file exists
            call checkFileExists(input)
        case(2)
            ! 2nd argument is N; conver arg to an integer
            read(arg,*)N
        case default
            ! ignore further arguments
    end select
end do

write(*,'(a,i3,/,2a)')"N:",N,"Input file: ",input
! continue program execution here
end program testargs1

```

Try running the program, and passing several different combinations of arguments – see what happens!

- ▶ In the program, we first check how many arguments have been provided using `command_argument_count()`.

Since we *need* to have an input file specified, if no arguments are provided (`Nargs==0`) we then terminate the program.

Next, we loop over all arguments, making sure there is no error, and then using a case selection to determine what happens.

After reading the first argument, we set the input filename and check if the file exists.

If the second argument exists, we use it to set N; otherwise it has the default value 10 as given at the start of the program.

Finally, arguments 3+ go to the default case. At the moment they are ignored, but this could be altered.

### Example 2.28 Flag-valued command line arguments

Sometimes we want a little more flexibility when it comes to passing command line arguments. You may have noticed that many UNIX programs use **flags** in their arguments, providing shortcuts and allowing you to specify the *order* in which you place input information, or whether you provide the information at all.

- ▶ Unlike Example 2.27, we now decide how to act on the argument based on the **value of the flag**, not its position.

If the flag is identified as a **value flag** (for example, **--input** and **-N**), then we instruct the program that the **next** argument will contain the **value**, via the use of a Boolean variable (e.g. **findN** or **findInput**).

Once the value has been set, we clear the Boolean variables and continue searching for flags.

We have also introduced the **Boolean flags**:

**--help** or **-h**  
print a short help statement and quit  
**--version** or **-v**  
print the version number and quit  
**--debug** or **-d**  
turn on debugging

The debug flag is quite useful for debugging code. If provided, the variable **debug** is set to **.true.**, and various **write** statements outputting the state of the program will be conditionally allowed.

Flags fall into two broad categories:

- **Boolean flags**: These do not accept any options, the mere presence of the flag itself changes the behaviour of the program.
  - For example, you've probably come across the flag **--help** or **-h** – these are commonly used to instruct the program to output a brief help message and exit.
  - These can also be used to pass the value of a variable with only two possible options (i.e. **.true.** or **.false.**)
- **Value flags**: these require one or more additional *values* to be passed, separated from the flag by a space.
  - For example, **-N 5** or **--input data.txt**. This is commonly used to pass real, integer, complex or character data to the program.

#### Solution:

Let's modify Example 2.27 to use flag-valued command line arguments; for example,

```
argtest -N 5 --input data.txt
```

(note that the order of arguments does not matter when using flags).

```
program testargs2
  use argfuncs
  implicit none
  character(len=50) :: arg, input
  integer            :: i, ierr, N
  logical             :: debug, findN, findInput

  ! Set the default command line options
  N = 10
  input = ""
  debug = .false.

  ! loop over command line arguments
  do i=1, command_argument_count()
    ! store the ith argument in arg
    call get_command_argument(i,arg,status=ierr)
    ! check if an error occurred
    call checkArgStatus(ierr)

    ! adjust the argument to the left
```

```
arg = adjustl(arg)

! determine what action to take,
! depending on the value of the argument
select case(arg)
  case("-N")
    ! the next argument will hold the value of N
    findN = .true.
  case("--input","-i")
    ! the next argument will hold the value of input
    findInput = .true.
  case("--version","-v")
    ! print the version number and quit
    write(*,*)"version 3.14159"
    stop
  case("--help","-h")
    ! print the help and quit
    call PrintHelp()
    stop
  case("--debug","-d")
    ! set the debug logical to true,
    ! and move on to the next argument
    debug = .true.
  case default ! this will match all the flag VALUES.
    if (findN) then
      ! convert the argument to integer N
      read(arg,*)
      findN = .false.
    elseif (findInput) then
      ! convert the argument to string input
      input = arg
      ! check if the input file exists
      call checkFileExists(input)
      findInput = .false.
    else
      ! argument unrecognised; quit
      write(*,*)"Error: argument " &
                  & //trim(arg) &
                  & //" unknown"
      call PrintHelp()
      stop
    endif
  end select
end do

if (input=="") stop "Error: no input file specified"

if (debug) write(*,'(a,i3,/,2a)')"N:",N,"Input file: ",input
  ! continue program execution here
end program testargs2
```

## 2.10 Timing Your Code

In most general programming cases, code execution time is typically neglected – at least, it has been for all examples and problems posed so far in this text! But in the field of computational physics, where simulations can become increasing computationally-intensive (for example, solving Schrödinger’s equation in multiple dimensions), the difficulty in ensuring your code converges to a solution in a timely manner can sometimes approach (if not overtake!) the difficulty of numerically solving your problem in the first place.

In fact, knowing the total elapsed time it takes for your program to complete, from start to finish – referred to as the **wall-clock** or **wall-time** – can be quite useful (for example, in estimating the wall-time required when submitting jobs to a scheduler on a cluster). Additionally, determining the wall time for specific functions, subroutines, or control structures (e.g. do loops) can help with determining the most computationally-intensive portion of your program, and indicate which sections should be optimised if possible.

The Fortran standard provides two intrinsic subroutines for timing; the `cpu_time` and `system_clock`. Let’s have a look at these two functions.

### 2.10.1 CPU Time

CPU time differs from wall time in that it measures the time spent on the CPU by your process. So for instance, whilst wall time gives you the total elapsed time (including the time spent by your process *waiting* for resources such as the processor, network, and/or IO devices), the CPU time simply gives you the time spent executing your code on the CPU – thus, in general for non-parallel programs, CPU time < wall time.

Subroutine	Description
<code>cpu_time(time)</code>	returns the elapsed CPU time in seconds. • <code>time</code> – <b>real</b> , <b>intent(out)</b>

To measure CPU time in Fortran, we use the `cpu_time` function as below:

```
real :: start, finish, cputime
call cpu_time(start)
! code to time
call cpu_time(finish)
cputime = finish - start
```

Note that the absolute value of `cpu_time` does not mean anything; rather, it is the *relative* CPU timing (or *difference* between two CPU times) that give us the CPU time of the code section we are interested in.

- ▶ Note that some compilers provide their own, non-standard, intrinsic Fortran timing functions.

For example, gfortran provides a function `itime`.

However, to ensure portability of your code, it is safer to stick to the intrinsic subroutines defined in the standards – `cpu_time` and `system_clock`.

### 2.10.2 Wall Time

Whilst measuring CPU time is sufficient for simple sequential programs, in more advanced cases it is better to measure the wall time. For example, when working on a multi-tasking machine, the CPU and wall time could differ significantly as different processes compete for resources. Furthermore, the resulting CPU time for a parallel program will sum together the individual CPU times of each thread/process – leading to the result CPU time > wall time!

---

**call system\_clock(count,crate,cmax)**

returns the count as specified by a processor clock. For increased timing precision, it is recommended double precision variables be used.

- **count – integer, intent(out)**  
variable returning the current count of the processor clock
  - **crate – integer or real, intent(out)**  
(optional) variable returning the number of clock ticks per second
  - **cmax – integer, intent(out)**  
(optional) variable returning the max number of clock ticks before the count resets to 0
- 

To measure wall time in Fortran, we use the `system_clock` intrinsic subroutine as follows:

```
integer :: count_start, count_finish, count_rate
real    :: walltime
call system_clock(count_start, clk_rate)
! code to time
call system_clock(count_finish, clk_rate)
walltime = (count_finish - count_start)/real(count_rate)
```

Note that, as with `cpu_time`, it is the *difference* between the count rates that gives us the elapsed count. However, we need to do an additional calculation to convert the count difference into seconds, by dividing by the count rate.

#### Further reading

The internet has become an incredible resource to programmers, allowing access to countless pages of documentation tutorials withing seconds. A great (but high level) resource is the online documentation for `gfortran` (see <https://gcc.gnu.org/onlinedocs/gcc-8.1.0/gfortran/>), providing details on all intrinsic Fortran functions. In addition, [Stack Over-](#)

flow is an online Q&A forum for programmers; you can pose your own questions, or search the site to find similar posts.

For additional Fortran texts, focused on scientific computation, we recommend the following, ranging from intermediate to advanced:

- Chapman, S. J. (2018). Fortran for scientists and engineers (Fourth edition). New York, NY: McGraw-Hill Education.
- Metcalf, M., Reid, J. K., Cohen, M., & Metcalf, M. (2011). Modern Fortran explained. Oxford; New York: Oxford University Press.
- Chivers, I., & Sleighholme, J. (2018). Introduction to programming with Fortran. New York, NY: Springer Berlin Heidelberg.
- Hanson, R. J., & Hopkins, T. (2013). Numerical computing with modern Fortran. Philadelphia: Society for Industrial and Applied Mathematics.
- Hager, G., & Wellein, G. (2011). Introduction to high performance computing for scientists and engineers. Boca Raton, FL: CRC Press.

The last two books in the list include chapters on parallelisation using OpenMP and MPI, and important part of high performance computation.

---

## Exercises

**P2.1** Write a program which reads a value  $x$  from keyboard, and calculates and prints the corresponding value  $\sin(x)/(1 + x)$ . The case  $x = -1$  should be programmed to produce an error message, and be followed by an attempt to read a new value of  $x$ .

**P2.2** Write a program to calculate and print the factorial of 1, 2...12. Your code will probably not work for any value beyond 12.  
If that is the case, can you find out why? Can you get correct answers for the factorials of up to 20?

**P2.3** Consider the following Fortran program, which calculates the sum  $a+b+c$  in two different ways, where  $a = 12345678.$ ,  $b = -12345677.$ , and  $c = 0.5412382$  are single precision reals:

```
program AssociativeSum
    implicit none
    real(4) :: a,b,c

    a=12345678.
    b=-12345677.
    c=0.5412382

    write(*,*) '(a+b)+c = ', (a+b)+c
    write(*,*) '(a+c)+b = ', (a+c)+b
end program AssociativeSum
```

Try compiling and running the code, and compare the two outputs.

- (a) Why do these two sums produce different results, when summation is associative in Fortran? Which of these two associative sums is correct?
- (b) How can you fix the code so that they both produce the correct output (as would be expected)?

**P2.4** Write a program to evaluate the exponential function using a Taylor series:

$$e^x = 1 + x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \dots$$

Output the result for  $x = -5.5$ .

How quickly does your code converge (i.e. how many terms do you need in the sum in order to achieve an accuracy of 1% or better)?

**P2.5** The  $N \times N$  discrete Fourier transform (DFT) matrix is defined by

$$W = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

where  $\omega = e^{-2\pi i/N}$ .

Use a Fortran array to create a  $4 \times 4$  DFT matrix. Apply what you know about formatting to output a neatly formatted matrix on the screen.

**P2.6** Write a function or subroutine that accepts integer  $N$  as input and returns an  $N \times N$  DFT matrix

**P2.7** Fortran does not provide an intrinsic function or value for  $\pi$ . One method of calculating  $\pi$  is via the formula  $\pi = 4 \tan^{-1}(1)$ .

- (a) Write a module containing the functions `sPI()`, `dPI()`, and `qPI()`, which calculate  $\pi$  using single, double, and quad precision respectively. Compare the values of  $\pi$  calculated using these three functions.
- (b) Modify the three functions `sPI(x)`, `dPI(x)`, and `qPI(x)` so that each accepts a single real variable  $x$  of single, double, and quad precision respectively. Then, use module procedure overloading to define a generic function `PI(x)` that calculates  $\pi$  to the same precision as the provided real argument  $x$ .

**P2.8** The `real` intrinsic function `atan2(x,y)` returns the *argument* of the complex number  $z = x + iy$ , where  $x$  and  $y$  are both `real` variables. In other words, it returns the complex phase  $\phi = \arg(z)$ .

The absolute value function, `abs()`, can also accept complex variables, and can be used to calculate the *magnitude* of the complex number  $r = |z|$ .

- (a) Using the intrinsic functions `atan2(x,y)` and `abs(z)`, write a Fortran function that accepts a complex variable `z=(x,y)` and returns a 2 element vector containing the magnitude and phase.
- (b) Using format statements and the result of part (a), write the complex number  $z$  to the terminal in polar form ( $z = re^{i\phi}$ ). For example, you could display  $z = 1 + 2i = 2.23e^{1.11i}$  as `2.23 exp(1.11 i)`.
- (c) Modify the program so that the real and complex parts of the initial complex number  $z$  are passed via command line arguments.



## Chapter 3

# Python

In the previous chapter, we provided a crash course in Fortran for physicists, a programming language first developed in the fifties specifically for scientific computing. While well-established and underpinning decades of scientific computational results (and still an excellent candidate if speed is a major concern), Fortran is beginning to be supplemented by more modern programming languages. Chief among these is Python, which is becoming increasingly popular among scientists for several important reasons:

- **It's easy to learn.** Most scientists aren't programmers by trade; programming is instead an important tool. Python code is often more readable and concise compared to Fortran, especially for people new to programming.
- **Python is open source and cross-platform.** This make it easy to install on your choice of operating system, whether MacOS, Windows, Linux, or even Android!
- **There is a multitude of available scientific libraries for Python.** As Python increases in popularity, more and more software libraries for Python become available, ranging for example from micromagnetic simulators to quantum computation. This reduces the time required to write your code — instead of writing an algorithm yourself, simply make use of an already existing implementation.
- **It makes prototyping faster, and is interactive.** Because Python is an *interpretive* language, there is no need to recompile your code every time you make a change; the Python interpreter simply compiles your code as you run it. This means it is even possible to run Python interactively, line-by-line.

► This chapter complements Chap. 2, the introduction to Fortran. You're welcome to read and work through both chapters, but this is not required! From this chapter onwards, all code examples (denoted with the heading Example) will be provided in both Fortran (a grey background) and Python (an orange background).

This last point, however, comes with consequences. Because Python is an interpretive language, it will never be as fast as compiled Fortran or C code. So, be careful when choosing the programming language you use; if you

need speed, it might be better to use Python to prototype your code, before porting it to Fortran for the proper run.

Below, we illustrate one of the simplest Python programs. If you have just read Chap. 1, notice that we no longer need to compile the program — we simply save our code as a text file with file extension `.py`, and run the file using Python.

### Example 3.1

Write a simple Python program that outputs Hello World! to the terminal.

- ▶ The first line of a Python program is conventionally written as

```
#!/usr/bin/env python
```

This is known as the **shebang**, and on Unix based operating systems (such as Linux and MacOS), instructs your terminal that the Python interpreter should be used to run the script.

The character `#` is also used for comments in Python, so operating systems that don't use the shebang (such as Windows) will ignore this line.

#### Solution: `hello_world.py`

```
#!/usr/bin/env python
# write 'Hello World!' to the terminal
print("Hello World!")
```

and then run the code using

```
$ python hello_world.py
```

In this chapter, we will provide an introduction to Python, setting you on the path to solving simple and then more difficult problems in physics.

#### Important note: Python 2 versus Python 3

After debuting in 2000, Python version 2.0 saw a rapid rise in popularity, becoming the dominant form of Python used in scientific computing. The last version of Python 2, version 2.7, was released in June 2009.

In 2008, the next major version of Python was released, Python 3.0. Python 3 was specifically designed to simplify Python, and remove fundamental flaws in the language. Due to these major changes, Python 3 is no longer backwards compatible with Python 2; code written using Python 2 would need to be modified to run using Python 3.

For a while, most developers and scientists stayed with Python 2.7, due to the huge number of libraries and modules that they depended on also remaining on Python 2.7. However, as more and more libraries have been ported to Python 3, and with the end of support for Python 2.7 (scheduled for 2020) looming, Python 3 adoption finally gains momentum in the scientific community.

In this book, we will be teaching Python 3, currently at version 3.7. So, every time we mention ‘Python’ code, a ‘Python’ program, or a ‘Python’ interpreter, you should generally assume we mean Python 3.

### 3.1 Preliminaries: Tabs, Spaces, Indents, and Cases

Before we delve into details, we need to go over a few ground rules of Python.

- **Python is case-sensitive**, unlike Fortran.

Try changing Example 3.1 to read

```
Print("Hello World!")
```

What happens? You should see the following error message:

```
NameError: name 'Print' is not defined
```

The error message occurs because Python doesn't recognise the function `Print()` with a capital letter – in Python, the use of upper case and lower case matters. For example, the Python interpreter will treat `var1` and `Var1` as different variables. So when copying examples and functions from here, always make sure the case is right.

- **Python uses indents to structure code-blocks.**

What do we mean by this? Well, consider a program that checks if a variable  $x$  is less than 12; if it is, it sets a variable  $y$  to 'am'. Using Fortran, this program might look like this:

```
if (x<12) then
    y = "am"
end if
```

Here, the entire `if` statement is a **block**, with the Fortran compiler knowing it has reached the end of the block when it reads the statement `end if`. In C, we would indicate the start and end of blocks with braces {} instead:

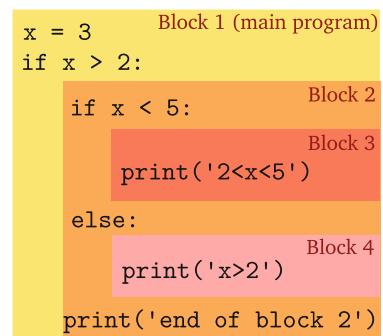
```
if (x<12) {
    y = "am";
}
```

In both cases, it doesn't matter how we format our code — as long as the `if`'s and `end if`'s, {'}s and `}`'s are in the right place, they're all equally valid. For instance, we could rewrite the C code on one line: `if (x<12) {y="am";}` So, how does Python use indents to separate code blocks? Well, like this:

```
if x<12:
    y = "am"
```

Instead of braces or 'statements', Python indicates the start of a code block with a **colon :**, with the remainder of the code block indented by using either tabs or spaces. The code block ends when the indentation reverts back to the previous indentation.

So, in Python, **whitespace at the beginning of lines is important and can affect the code**. As a result, when copying Python code from this textbook, always make sure to copy the whitespace as-is.



**Figure 3.1** An example of how Python distinguishes between code blocks using indentation

- A consequence of Python placing importance on whitespace is that everyone is forced to use the same formatting and style for their code blocks — formatting becomes consistent between different programmers, helping with readability.

### When can I press enter?

When writing a Python program, sometimes you need to make the distinction between a physical line and a logical line.

- **Physical line:** a single line of text that you see in your text editor
- **Logical line:** a single statement that the Python interpreter sees

This distinction is important; you can have multiple logical lines on one physical line, or vice versa, multiple physical lines describing one logical line.

### Compound statements

For multiple logical lines on a physical line, we can use a **semicolon**,

```
x = 5; y = 6; x + y
```

This line of code contains three logical lines. This is sometimes used for conciseness or for communicating short code snippets, however from convention it is generally discouraged — only use compound statements as a last resort.

### Line joining

If statements become too long and unwieldy, they can continue over multiple lines by explicitly using a backslash at the end of the line:

```
if x < 3 and y > 5 and \
x*y < 2 or x*y > 10:
    print('Yes')
```

We now have one logical line (the **if** statement) spread over two physical lines. Note that you cannot have a comment after the backslash — as far as the Python interpreter knows, you're still in the middle of one statement! You also can't use the backslash to create multiline comments.

If you need to, it is better convention to use **implicit** line joining. Any expression between round, curly, or square brackets can span multiple lines. For example,

```
print(5, 6.01, 8.2, # implicit line breaks
      -0.1, "hello") # can have comments
```

or

```
days_of_the_week = ["Mon", "Tues",
                    "Wed", "Thurs", "Fri",
                    "Sat", "Sun"]
```

---

**Tabs or spaces?**

The decision of whether to indent your Python code using spaces or tabs is up to you, and is a matter of style. However, it is important not to mix styles — make sure to keep to solely tabs *or* spaces, not both.

However, due to different operating systems (and even different text editors) displaying tabs differently, it is sometimes easier to use spaces in the long run. [PEP 8](#), the recommended Python style guide, recommends spaces over tabs, with 4 spaces per indentation level.

---

## 3.2 Variables, Numbers, and Precision

### 3.2.1 Assigning Variables

- If you try and use a variable in a Python statement without first assigning it a value, you will get a **NameError**

In Python, we assign variables using the equals sign, `=`:

```
a = 1
```

You can assign multiple variables at once, all with the same value:

```
a = b = c = 2
```

Or assign multiple variables at once, all with difference values:

```
a, b, c, d = 1, 2.5, True, "hi"
```

Python3 variable names can use **any** alphanumeric unicode character, but *must* start with either a letter or an underscore.

### Example 3.2

Write a program that converts temperature in Celsius to Fahrenheit.

**Solution:**

```
#!/usr/bin/env python

# the input statement reads input from the keyboard as a string
key_input = input('type in temp in C\n')

# we then convert this to a float, a real number,
# and store it in variable DegC
DegC = float(input('type in temp in C\n'))

# converting this to Farenheit
DegF = (9./5.)*DegC + 32.

# the print command lets us output text to the terminal
print('This equals to',DegF,'Farenheit')
```

In the above example, we have a simple Python program which accepts a real-valued temperature (in degrees Celsius) as input, converts it to Fahrenheit, and outputs the result. Let's walk through a couple of important constructs in this example.

- **Anything to the right of the symbol # are comments.** In a line, anything following `#` is ignored by the Python interpreter, and will be coloured in a dull blue in this book.

- **Arguments are passed to functions using parentheses ( and ).**

In the above example, `input()`, `float()`, and `print()` are all **intrinsic Python functions**, and accept arguments given inside the parentheses. In this example,

- `input()` will output its text argument to the terminal, and will return the resulting user input from the keyboard.
- `float()` converts a **string** (describing a collection of characters or text) to a real number, a **float**.
- `print()` will output its string argument to the terminal.
- The string '`\n`' is interpreted as a **new line** by Python, and does not actually appear in the program output. We'll cover this further when looking at string manipulation.
- **You assign variables in Python using the operator =.** In Python, you do not need to declare your variables and their type beforehand — this is all done under-the-hood and on the fly by the Python interpreter.

Python is **dynamically typed**, you do not declare your variables and their type before you use them. Programming languages like Fortran and C are **statically typed**, and you **do** need to declare your variables and their types beforehand.

In Python, several different types of scalar variables are supported; the type that your variable is assigned to will always match the type on the right hand side of the assignment. If the expression to be assigned is an integer, then Python will assign the variable as a `int`. If the expression instead is a numeric with decimal places, then Python will instead assign it as a `float`. In Table 3.1, a partial list of available Python data types is provided.

<code>int</code>	<b>Integer</b> , of arbitrary precision (the precision is only limited by the computational memory available)
<code>float</code>	<b>Double precision real number</b> containing 53 bits of precision
<code>complex</code>	<b>Double precision complex number</b> written using the suffix <code>j</code> after the imaginary part; for example, <code>5.234-0.9432j</code> , represents $5.234 - 0.9432i$
<code>str</code>	A <b>string</b> , for example ' <code>abcdefg123</code> ' or " <code>abcdefg123</code> "; these can be delimited by single quotes or double quotes
<code>bool</code>	<b>Boolean literal</b> which can only take two values; <code>True</code> or <code>False</code>

Table 3.1 Python scalar data types

If you're ever unsure of the data type of a variable in Python, simply apply the intrinsic function `type()` to the unknown variable to find out.

### 3.2.2 Type Conversion

As Python is dynamically typed (you don't declare variable types in advance), you generally don't need to worry too much about variable types — Python will try and anticipate what you're trying to do, and perform **implicit conversions** when appropriate. At other times, however, you might need to convert variables between data types **explicitly** yourself. Here, we'll go through a couple of times when you'll have to consider both cases.

#### Implicit Type Conversion

- ▶ Note that the presence of a decimal point indicates a float rather than an integer, even if no decimal places are present. So `a = 5.` will be stored as a float, and `b = 5` will be stored as an int.

Say you have two integer variables, `int1` and `int2`, and you perform a division on them as in the following:

```
>>> int1 = 5
>>> int2 = 2
>>> int3 = int1/int2
```

What do you suppose the data type of `int3` is, integer or float? For example, you can add the following line to the end of this program:

```
>>> print('int3 = ',int3)
int3 = 2.5
>>> print(type(int3))
<class 'float'>
```

This indicates that an implicit type conversion is occurring, with the division of `int1` and `int2`, two integers, producing a `float`.

In general, for any expression containing a combination of numbers, the following implicit type conversions take place:

1. If any argument is a complex number, the result will be complex;
2. If there are no complex numbers, but there are floats, then the result will be a float.

There are exceptions, of course; operations (such as division as seen above), or functions (such as the square root of a negative number) might result in an implicit type conversion taking place.

#### Explicit Type Conversion

While you need to be aware of any implicit type conversions that might affect the outcome of your code, in Python you will mostly end up applying type conversions explicitly. Table 3.2 lists a couple of the most useful type

conversion functions. Note that the type conversion functions for converting to a particular type have the same name as the type itself! (How convenient.)

### Example 3.3

Convert the string "125.6" into a float, add 21 to it, and then convert it back into a string.

**Solution:**

```
a = "125.6"
# converting the string into a float and adding 21
b = float(a) + 21
#converting back into a string
c = str(b)
# printing variables a, b, and c:
print(a,b,c)
```

*Output:* 125.6 146.6 146.6

• <b>int(x [,b])</b>
Converts a float or string x to an integer, in base b (optional)
• <b>float(x)</b>
Converts a integer or string x to an float
• <b>complex(x [,y])</b>
Creates the complex number $x + yi$
• <b>str(x)</b>
Converts x to a string

**Table 3.2** Intrinsic Python functions to convert between types.  
Arguments in square brackets are *optional*.

### 3.2.3 Precision and Round-Off Errors

Let's consider a quick, almost trivial, example.

### Example 3.4

Add together the numbers 0.1 and 0.05. Print out the result.  
Is it what you expect?

**Solution:**

```
print(0.1 + 0.05)
```

*Output:* 0.1500000000000002

Before you shriek and run for cover, Python is not failing you, and the laws of math are not collapsing in on themselves! In fact, Python is doing exactly as it is supposed to. Recall from Chap. 1, that computers store real numbers using floating-point representation, which may result in **round-off** or **truncation** error. We mentioned before that Python 3, by default, represents all **floats** using IEEE754 double precision, allowing 53 bits of precision (approximately equal to 15 or 16 significant digits in base-10). The error seen here is due to unavoidable truncation of the numbers 0.1 and 0.5 in this floating-point representation.

While it is hard to avoid round-off error, it is important to be aware of when it might impact parts of your program — in later chapters, we will see first hand what happens if you don't consider round-off errors.

### Exploring floating-point precision in Python

There are two Python modules that let you explore how Python stores floating-point numbers: `decimal` and `fractions`.

The `decimal` module provides a type called `Decimal`, that allows for **exact decimal representation** of numbers in Python — providing a schoolbook approach to mathematics in Python! For example,

```
>>> from decimal import Decimal  
>>> x = Decimal('0.1') + Decimal('0.05')  
>>> print(x)
```

provides an output of exactly `0.15`, unlike when using the inbuilt `float` type. `Decimal` also allows you an insight into how Python stores its floating-point numbers, by passing a `float` directly to the `Decimal` function instead of a string:

```
>>> print(Decimal(0.1))
```

which will output

`0.100000000000000055511151231257827021181583404541015625`

The `fractions` module provides a type called `Fraction`, and works similarly to `decimal`, however instead of providing exact decimals, it provides an exact fractional representation of each `float`. This is possible because floating-point numbers, due to their fixed number of significant digits, can only encode rational numbers. For example,

```
>>> from fractions import Fraction  
>>> print(Fraction(0.1))
```

provides

`3602879701896397/36028797018963968`

which is equal to the output of `Decimal(0.1)` shown above.

---

### 3.3 Operators and Conditionals

Operators are programming constructs that act on various data types, allowing us to perform pre-defined ‘operations’ on different data types. Python has seven main categories of operators: arithmetic, assignment, bitwise, relational, membership, identity, and logical. Here, we’ll go through several of these categories, and provide examples along the way.

#### 3.3.1 Arithmetic Operators

In order of highest to lowest precedence, the arithmetic operators are:

- **Unary operators:** minus (`-`) and plus (`+`)

These operators, unlike the ones detailed below, only act on a *single* numeric value — hence the term ‘unary operators’. The plus operator makes no change to a numeric value (`+5 = 5`) while the minus operator negates it (`-5 = -5`).

- **Exponentiation (`**`)**

Note that, somewhat unintuitively, exponentiation in Python is right to left associative; that is,

$$2 ** 3 ** 2 = 2^{(3^2)} = 2^9 = 512$$

and **not**

$$2 * * 3 * * 2 \neq (2^3)^2 = 8^2 = 64$$

The exponentiation operator takes precedence over unary operators on the right,

$$2 ** -5 = 2^{-5}$$

but not on the left

$$-2 ** 5 = -2^5$$

- **Multiplication (`*`), division (`/`), integer/floor division (`//`), and remainder/modulo (`%`)**

Unlike exponentiation, these are left to right associative:

$$a * b * c = (a \times b) \times c$$

Integer or floor division is distinct from regular division, as it performs the floor operation, rounding down to the nearest integer:

$$5 // 2 = \left\lfloor \frac{5}{2} \right\rfloor = 2$$

The remainder or modulo operator `%` returns the remainder after division as an integer:

$$5 \% 2 = 5 \bmod 2 = 1$$

► To make your code easier to read, to avoid ambiguities, and to make sure the order of operations performed is the order of operations required, always use brackets `()` to group arithmetic operations!

► Both the floor division and modulo operator perform an implicit type conversion, always giving their output as an integer.

- **Addition (+) and subtraction (-)**

Like multiplication and subtraction, there are also left to right associative.

**Note:** the addition operator `+` and the multiplication operator `*` can also be used on non-numeric data types, including strings, lists, and sets.

### 3.3.2 Assignment Operators

We've already seen come across one example from the assignment operators back when discussing variable declaration; the assignment operator `=`, which assigns values on the right hand side to variables on the left hand side. Python also allows us to combine the assignment operator with the arithmetic operators introduced above; providing a convenient shorthand when an assignment *and* an arithmetic operation **on the variable to be assigned** need to be combined. Below, we list the available assignment operators in Python.

<code>=</code>	assignment	<code>x=z</code> assigns <code>x</code> the same value as <code>z</code>
<code>+=</code>	addition assignment	<code>x+=1</code> is equivalent to <code>x=x+1</code>
<code>-=</code>	subtraction assignment	<code>x-=1</code> is equivalent to <code>x=x-1</code>
<code>*=</code>	multiplication assignment	<code>x*=2</code> is equivalent to <code>x=x*2</code>
<code>/=</code>	division assignment	<code>x/=2</code> is equivalent to <code>x=x/2</code>
<code>**=</code>	exponentiation assignment	<code>x**=2</code> is equivalent to <code>x=x**2</code>
<code>//=</code>	floor division assignment	<code>x//=2</code> is equivalent to <code>x=x//2</code>
<code>%=</code>	modulo assignment	<code>x%=4</code> is equivalent to <code>x=x%4</code>

Table 3.3 Python assignment operators

### 3.3.3 Bitwise Operators

- The intrinsic Python function `bin()` allows you to view the binary representation of an integer:

`bin(51) = 0b110011`

Note that the preceding `0b` indicates a binary number; the significant digits start from *after* the `0b`.

In fact, you can enter binary integers *directly* in Python, by prefacing your binary digits with `0b`

Of all the operators presented here, the bitwise operators are perhaps the hardest to interpret. The bitwise operators apply only to **integers**, and act on the binary representation of the integer; they are listed in Table 3.4 in order of precedence from highest to lowest (with one caveat — the bitwise invert operator, being unary, has the same precedence as the previously discussed arithmetic unary operators).

To see how this works, consider the two integer variables `x = 2` and `y = 11`. Using a 4 bit binary representation,  $2 = 0010_2$  and  $11 = 1011_2$ . Thus, working through the list of bitwise operators:

- Bitwise inverse:  $\sim x = -2 - 1 = -3$

<code>~</code>	bitwise inverse	A unary operator, $\sim x$ inverts the binary representation of $x$ , and returns the corresponding integer, $-x - 1$
<code>&lt;&lt;</code>	bit-shift left	$x \ll b$ shifts the binary representation of $x$ left by $b$ digits, and returns the corresponding integer
<code>&gt;&gt;</code>	bit-shift right	$x \gg b$ shifts the binary representation $x$ right by $b$ digits, and returns the corresponding integer
<code>&amp;</code>	bitwise AND	$x \& y$ returns an integer with binary representation such that it only has value 1 in the positions where both $x$ and $y$ have 1s
<code>^</code>	bitwise XOR	$x ^ y$ returns an integer with binary representation such that it only has value 1 in the positions where either $x$ or $y$ have 1s, but not both
<code> </code>	bitwise OR	$x   y$ returns an integer with binary representation such that it only has value 1 in the positions where $x$ and/or $y$ have 1s

**Table 3.4** Python bitwise operators

- Bit-shift left:  $x \ll 2 = 1000_2 = 8$
- Bit-shift right:  $y \gg 1 = 101_2 = 5$
- Bitwise AND:  $x \& y = 0010_2 = 2$
- Bitwise XOR:  $x ^ y = 1001_2 = 9$
- Bitwise OR:  $x | y = 1011_2 = 11$

We will not delve much into the bitwise operators for the duration of this book, and only include them for completeness. For more details, please see the further reading section at the end of this chapter.

### 3.3.4 Relational Operators

Also known as comparison operators, these operators allow you to compare the values of different expressions or variables. Note that the relational operators have a lower precedence than the arithmetic operators.

The relational operators take two operands, and return either `True` (if the statement is true) or `False` (if the statement is false). For example,

```
>>> 5.4 <= 6.
True
>>> "hello" == "Hello"
False
```

<code>==</code>	equal to
<code>!=</code>	not equal to
<code>&lt;</code>	less than
<code>&gt;</code>	greater than
<code>&lt;=</code>	less than or equal to
<code>&gt;=</code>	greater than or equal to

**Table 3.5** Python relational/comparison operators

In the latter case, we get a result of **False** due to the case difference in the string.

Relational operators can also be chained, accepting 3 operands:

```
>>> x=2
>>> 3 <= x < 7
False
>>> 2 >= x < 5
True
```

### 3.3.5 Membership and Identity Operators

The membership operators test for membership of certain elements in a sequence, and can be applied to **sequences**, such as strings. The two available membership operators are **in** and **not in**, and work as follows.

- **in**: returns **True** if the element provided on the left hand side is contained within the operator on the right hand side, and **False** otherwise:

```
>>> "o" in "Hello World!"
True
>>> "Hello" in "Hello World!"
True
>>> "j" in "Hello World!"
False
```

- **not in**: **True** if the element provided on the left hand side is *not* contained within the operator on the right hand side, and **False** otherwise. It is the negation of the **in** operator.

The identity operators **is** and **not is** work similarly to the relational operator **equals** (**==**). However, rather than checking whether two variables have the same *value*, the identity operator checks whether two variables point to the same *memory location*. **The identity operator is thus a stronger test of equality than the equals to operator.**

Consider the following example:

```
x is y
implies
x == y
However, note that the
reverse is not guaranteed.
```

```
>>> x = 5
>>> y = x
>>> x == y
True
>>> x is y
True
```

The final two statements are **True**, as **x** and **y** both refer to the *same* memory location on the computer – rather than creating a new memory address to store **y=5**, Python optimises the process by simply pointing to the same memory location that stores **x=5**. To see that this is the case, run the following

code, and check the output of both `id(x)` and `id(y)` — they should give the same answer.

To see how this differs from using `==`, see what happens if we tweak the above slightly:

```
>>> x = 5
>>> y = 5.
>>> x == y
True
>>> x is y
False
```

The last statement is now false. As `x` and `y` are now different data types (`x` is an integer, and `y` is a float), they are stored in different memory locations. On the other hand, the equality operator performs an implicit type conversion and returns true since both have the same *value*. Double check this yourself by running `id(x)` and `id(y)`.

### 3.3.6 Logical Operators

Also known as boolean operators, these have the lowest precedence of all Python operators. Python has three logical operators; `not`, `and`, and `or`. Table 3.6 describes them in order of precedence, from highest to lowest.

<code>not</code>	Logical negation; applying it to a <code>True</code> statement returns <code>False</code> , and applying it to a <code>False</code> statement returns <code>True</code>
<code>and</code>	<code>True</code> if and only if both operands are <code>True</code>
<code>or</code>	<code>True</code> if at least one operand is <code>True</code>

Table 3.6 Python logical operators

- ▶ Logical operators are left to right associative
- ▶ The `not` operator accepts one operand on the right, while `and` and `or` accept two operands, one to the left and one to the right:

`not x`

`x and y`

`x or y`

#### Example 3.5

In the Python program below, what do you expect the value of `y` and `z` to be?

```
#!/usr/bin/env python
x = True
y = not x and 3.5 <= 5. or 2 < -1
z = False or True and False and True
```

**Solution:** Following the order of precedence, the expression for `y` will be evaluated as `((not x) and 3.5 <= 5.) or 2 < -1`. As `x = True`, 3.5 is obviously less than 5, but 2 is *not* larger than -1, this becomes

$$\begin{aligned} ((\text{not } \text{True}) \text{ and } \text{True}) \text{ or } \text{False} &\Rightarrow (\text{False and True}) \text{ or } \text{False} \\ &\Rightarrow \text{False or False} \\ &\Rightarrow \text{False} \end{aligned}$$

To determine the value of  $z$ , let's rewrite the expression using brackets as per the order of precedence and associativity of logical operators:

$$\begin{aligned} \text{False or } ((\text{True and False}) \text{ and True}) &\Rightarrow \text{False or } (\text{False and True}) \\ &\Rightarrow \text{False or False} \\ &\Rightarrow \text{False} \end{aligned}$$

**Extension:** Extend the Python program above to print out the values of  $y$  and  $z$  to verify our working out above.

### Important

As you can see from the above example, always use brackets when writing out logical expressions (and any other expressions!), to make sure that (a) they work as expected, (b) there is no ambiguity, and (c) other people can easily read and understand your code!

#### 3.3.7 Conditional Statements

An important feature of a programming language is the ability to alter the code to be executed depending on whether a particular condition evaluates to true or false — in Python, this method of ‘code selection’ can be done using if statements.

##### Selection: if Statements

The **if** statement enables a choice of code to execute, depending on various logical conditions. It is structured as follows:

```
if condition1:
    # Python code block
elif condition2:
    # Python code block
else:
    # Python code block
```

In this case, **condition1** and **condition2** correspond to Python expressions that return either **True** or **False**. When the above **if** statement is encountered, the conditions are evaluated sequentially (i.e. from the top down). When the first condition that evaluates to **True** is found, the corresponding Python code block will be executed. Otherwise, if all of the conditions evaluate to **False**, the **else** code block is executed.

##### Example 3.6 : if statement

Find the smallest of three real numbers  $a$ ,  $b$ , and  $c$ .

**Solution:**

```
if a < b and a < c:  
    result = a  
elif b < a and b < c:  
    result = b  
else:  
    result = c
```

Note that the **elif** (a portmanteau of else-if) and **else** statements are completely optional; the only part required is the initial **if** statement. In this case, no code is executed if the condition is not met, and you may write the single **if** statement on one line if you wish:

```
if condition1: #code to be executed if true
```

**if** statements can also be multiply nested inside other **if** statements when needed, with the whitespace or indentation level distinguishing between the nested **if** statements. For example,

```
if condition1:  
    if nestedcondition:  
        # code block  
    elif:  
        # code block  
else:  
    # code block
```

## 3.4 String Manipulation

Python provides several intrinsic function and operators for manipulating strings. We'll begin by considering some string operators, and an important intrinsic function.

### The `len()` function

A string can be thought of as a sequence of characters; as such, we can use intrinsic Python function `len()` to determine the length of a string:

```
len("Everything's coming up Milhouse!")
```

In the above, we can see that `len()` acts on a string, returning an integer representing the number of characters or length of the string. In this case, we would get 32.

The `len()` function is very useful, and we will return to it again when considering other sequence data structures.

### 3.4.1 String Operators

- ▶ These string operators use the same symbols we use for addition and multiplication of numeric values. How does Python know which one to use? Easy: if the left operand is a string, then they will be interpreted as string operators.

#### • Concatenation (+)

The concatenation operator combines strings on either side into one single string; for example,

```
>>> s1 = "Hello"
>>> s2 = "world"
>>> s3 = "!"
>>> print(s1 + " " + s2 + s3)
'Hello world!'
```

#### • Repetition (\*)

The repetition operator repeats a string an integer number of times, and concatenates them together. For example,

```
>>> s1 = "shake, "
>>> s2 = "shake it off!"
>>> print(s1*4 + s2)
'shake, shake, shake, shake, shake it off!'
```

#### • Membership (`in` and `not in`)

We saw previously that the membership operators can be used to check to see whether a string contains specific substrings.

### 3.4.2 Substrings and Slicing

Another neat string manipulation is the ability to extract consecutive parts of a string, otherwise known as a substring. To extract a single character

The diagram shows a string "A\_string" enclosed in quotes. Below the string, the characters are indexed from 0 to 7. The characters are colored red: 'A' is at index 0, ' ' is at index 1, 's' is at index 2, 't' is at index 3, 'r' is at index 4, 'i' is at index 5, 'n' is at index 6, and 'g' is at index 7. The indices are black numbers from 0 to 7 positioned below each character.

**Figure 3.2** How Python indexes the characters in string. The characters are in red, and the respective indices in black

substring from a string, we can use the following syntax:

```
string[i]
```

where `i` is an integer that represents the  $(i-1)$ th character. Why the  $(i-1)$ th?  
Because **Python starts indexing sequences from 0**.

To extract a *range* of consecutive characters, we can use **slicing**

```
string[start:end:step]
```

This will extract the substring starting with the element indexed by `start`, and ending with the character indexed by `end - 1`, using an optional index step of `step`. Note that *any* of these parameters can be left blank; if this is the case, the Python interpreter will assume that `start = 0`, `end` coincides with the end of the string (i.e. for a string of length  $n$ , `iend = n`), and `istep = 1`. For example, consider the string `s = "arbitrary string"`:

```
s[0]      == "a"
s[4:]    == "trary string"
s[2:8]   == "bitrar"
s[:6]    == "arbitr"
s[2:10:2] == "btay"
```

### 3.4.3 String Methods

In Python, string manipulation functions are available in the form of **methods**; these are functions that are members of the `str` type. A method is similar to a function, but runs directly ‘on’ an object. To get an idea of how this works, consider the string variable `s1`, defined below:

```
>>> s1 = "The aurora borealis? At this time of year?"
```

Since `s1` has type string (`type(s1) == str`), it **inherits** the string methods that are a part of `str`. These can be accessed by appending a period to the end of the variable name, and then calling the method. For example, the method `str.upper()` converts all characters in a string to uppercase:

```
>>> s1.upper()
'THE AURORA BOREALIS? AT THIS TIME OF YEAR?'
```

Note that this applies to all strings, even those without assigned variables:

```
>>> 'At this time of day?'.upper()
'AT THIS TIME OF THE DAY?'
```

So unlike intrinsic functions, methods act directly on the object they are called from, with the potential to accept additional optional arguments.

Some of the more common Python string methods are listed in Table 3.7. Note that this is nowhere near a complete list; for the full list of available methods, refer to the list of further reading at the end of this chapter, or to the Python 3 documentation.

- ▶ The coding paradigm where objects ‘inherit’ methods and attributes from their initialising ‘class’ is called Object Oriented Programming.

<code>str.lower()</code>	Returns the string with all characters converted to lowercase
<code>str.upper()</code>	Returns the string with all characters converted to uppercase
<code>str.strip([chars])</code>	Returns the string with leading and trailing whitespace removed from a string. You can optionally provide any trailing/leading characters to be removed using the <code>chars</code> argument
<code>str.center(width [, char])</code>	Add leading and trailing <code>char</code> (if not specified, by default <code>char=' '</code> ), centering the original string, and returning a string of specified width
<code>str.find(substring [, istart, iend])</code>	Returns the index locations of a substring within the original string (within range <code>[istart, iend]</code> if required). If no cases are found, it returns <code>-1</code>
<code>str.count(substring [, istart, iend])</code>	Counts the number of times the substring appears in the original string (with no overlapping, and within range <code>[istart, iend]</code> if required)
<code>str.replace(old, new [, count])</code>	Replaces all occurrences of substring <code>old</code> with substring <code>new</code> . If optional argument <code>count</code> is specified, only the first <code>count</code> occurrences are replaced
<code>str.isalnum()</code> <code>str.isalpha()</code> <code>str.isdecimal()</code> <code>str.isspace()</code>	Returns <code>True</code> if all characters in the string are alphanumeric/alphabetic/decimal numbers/whitespace
<code>str.join(iter)</code>	Concatenate an iterable/sequence of strings into a single string, delimited by the original string. For example, <code>'-'.join(['some', 'strings', '2'])</code> would return <code>"some-strings-2"</code>
<code>str.split([char])</code>	Splits the string at every occurrence of a delimiting substring <code>char</code> to form a list of substrings. If not provided, <code>char=' '</code> by default. For example, <code>'comma,separated,values'.split(',')</code> would return <code>['comma', 'separated', 'values']</code>

Table 3.7 Python string methods

### 3.4.4 String Formatting

One string method we have not yet mentioned, but arguably one of the most important<sup>1</sup> is the `format` method, `str.format()`. String formatting allows for much finer control than our previous method of just ‘hoping for the best’ with the `print()` function, by using replacement fields to pass numeric values (and how they should be formatted) directly to strings.

The `format` method passes values to the string as *arguments*, and they are inserted into the replacement fields, denoted by curly braces `{}`:

```
>>> x = 3
>>> print("I have {} kids and no money".format(x))
```

In this example, we are passing the variable `x`, which holds an integer value of 3, to the `format` method acting on the specified string — the value of `x` is inserted directly into string at the location of `{}`:

```
"I have 3 kids and no money"
```

If we have multiple variables, we just use multiple replacement fields; the variables are placed successively in the replacement fields in the order they are supplied to the `format()`:

```
>>> x = 3
>>> y = 0
>>> print("Why can't I have {} money and {} kids".format(x,y))
```

We can also pass arguments to the replacement fields themselves, that determine the **ordering** of the variable replacement, as well as the **formatting** of each field. We use the following syntax:

```
{field_name:[[fill][align][sign][width][.precision][type]]}
```

where `[]` indicates optional arguments, and

- `field_name` is a integer that numbers the replacement field, with the numbering starting from 0. This gives the order for the variable replacement.
- `fill` and `align`: `align` indicates whether the variable should be centered (`^`), right-aligned (`>`, default for numbers), or left-aligned (`<`, default for everything else), within the field. If indicating alignment, you may also indicate a `fill` character to fill the remaining space.
- `sign`: valid only for number variables, this tells `format()` how to deal with signs for positive and negative numbers.

<sup>1</sup>So important that it receives its own section!

#### f strings

Python 3.6 introduced a new method for formatting strings, **f-strings**.

These work similarly to `format()`, but allow you to use named variables *directly*:

```
>>> name = 'Zach'
>>> f'My name is {name}'
'My name is Zach'
```

This is done by prepending a single `f` right before the string, and results in cleaner code compared to `format()`.

However, the `format()` method used in Python versions 3.5 and earlier continue to be supported in 3.6.

- **–** indicates that a sign should only be used for negative numbers (default)
- **+** indicates that a sign should be used for *all* numbers
- a space indicates that a **–** sign is used for negative numbers, and a leading space for positive numbers
- **width:** an integer that defines the character width of the field
- **.precision:** valid only for `float`, this argument indicates how many decimal digits to use. For example, `.8` would result in 8 decimal places.
- **type:** the *presentation type* of the field — see Table 3.8 for details.

#### Presentation types

For strings

`s str`, string format

For integers

`d int`, integer format

For floats

`e` scientific notation using `e`

`E` scientific notation using `E`

`f` fixed point (6 decimal places by default)

`g` chooses `e` or `f` depending on magnitude of number (default for `int` and `float`)

`G` chooses `E` or `f` depending on magnitude of number

`%` multiplies by 100 and displays with the `%` sign

#### Example 3.7 Scientific notation

Format the float 18947.81274593, its square, and its cube, using scientific notation with 2, 3, and 4 decimal places respectively.

#### Solution:

```
x = 18947.81274593
print("x = {:.2e}, x^2 = {:.3e}, x^3 = {:.4e}".format(x,x**2,x**3))
```

Output: `x = 1.89e+04, x^2 = 3.590e+08, x^3 = 6.8026e+12`

Note that in this example, we did not worry about providing a field name, since we know we are filling the replacement fields in the order they are given.

#### Example 3.8 Sign control

Use string formatting and sign control to align three columns of numbers. The first column should include 10 numbers starting from  $-10$  in steps of  $2.2$ , the second column should include all odd numbers between  $50$  and  $70$ , and the third column should include all powers of  $4$  of the integers  $0$ – $9$ . Use any delimiter you wish to visually separate the three columns.

#### Solution:

```
for i in range(0,10):
    output = "{0: 5.1f} {1:3d} {2:2d}\u2074 = {3:<4d}"
    print(output.format(-10+2.2*i, 51+2*i, i, i**4, delim="|"))
```

Output:

$-10.0$	$ $	$51$	$ $	$0^4 = 0$
$-7.8$	$ $	$53$	$ $	$1^4 = 1$
$-5.6$	$ $	$55$	$ $	$2^4 = 16$
$-3.4$	$ $	$57$	$ $	$3^4 = 81$
$-1.2$	$ $	$59$	$ $	$4^4 = 256$

Table 3.8 Format presentation types

#### String escape characters

`\n` newline

`\r` carriage return

`\s` space

`\t` tab

Table 3.9 String escape characters

```
1.0 | 61 | 54 = 625
3.2 | 63 | 64 = 1296
5.4 | 65 | 74 = 2401
7.6 | 67 | 84 = 4096
9.8 | 69 | 94 = 6561
```

- In Python 3, all strings are encoded in UTF-8, meaning they can include all the available Unicode characters.

Several things to note in this example:

- We are using `delimiter="|"` to name the string "`|`" using the keyword `delimiter`; this is known as a **keyword argument**, and we'll see more on this later when talking about functions. Keyword arguments must always come *after* the non-keyword arguments. In this case, they are useful due to the number of times it repeats in the string.
- In the first column, we are using sign control to align the decimal place, a field width of 5, allowing 1 decimal point of precision, and using presentation type `float`.
- For the third column, we are including the preface  $i^4 =$ ; the integer `i` has width two, and we are using the Unicode character U+2074, representing the superscript <sup>4</sup>. On the right hand side of the equals sign, we are using a left-aligned integer with field width 4.

### Multi-line string syntax

In cases where you need to include a large string that spans over multiple lines, Python provides **multi-line string syntax**, using triple quotes:

```
bigstring = """
Working out another system to replace Newton's laws took a
long time because phenomena at the atomic level were quite
strange. One had to lose one's common sense in order to
perceive what was happening at the atomic level.
--- Richard Feynman
"""
```

Be careful though — indentation and whitespace will become part of your multi-line string when using triple quotes!

## 3.5 Data Structures

So far, we have introduced you to numeric types (`int`, `float`, `complex`), strings (`str`), and logical types (`bool`). However, you've probably noticed hints throughout the previous chapters that this is not all — there are other data types available, known as **data structures**. These include **sequences** (tuples, lists, and ranges), and **collections** (sets, dictionaries).

### 3.5.1 Sequences

Sequences are data structures containing a collection of elements with a specific order. In fact, we've already come across one example of a sequence — a Python string is simply a sequence of Unicode characters! In addition, there are three other types of sequences we haven't yet covered; **tuples**, **lists**, and **ranges**.

#### Tuples

The first data structure we'll look at is the **tuple**, a sequence of values enclosed within parentheses () and separated by commas. For example, the following are all tuples:

```
>>> tp1 = (1, 2, 5, 1, -5, 0, -3, 8, 10)
>>> tp2 = ("In", "this", "part", "of", "the", "country?")
>>> tp3 = (1.653, 7.5-9.1j, True, "s")
>>> tp4 = (7,)
```

A couple of things to note about tuples:

- ▶ It is actually the presence of comma's that indicate to Python that you are defining a tuple, not the brackets!  
For instance,
- ```
x = 1, 4, 2
```
- creates a perfectly valid 3 element tuple.

- They can contain any combination of data types; they don't need to be restricted to just one data type per tuple.
- For a tuple of size one, you still need to include the trailing comma to indicate that this is a tuple, and not brackets as used in arithmetic!
- Tuples can be nested. For example, the following is perfectly valid:

```
>>> (tp1, tp4, 8)
((1, 2, 5, 1, -5), (7,), 8)
```

#### Lists

Lists, at first glance, are very similar to tuples. Both are data structures that can contain ordered sequences of elements, with tuples denoted by brackets, and lists denoted by square brackets:

```
>>> a = [1, 2, 5, 6, 1, -1, 0, 10]
```

Like tuples, they can contain any combination of data types, and they can be nested to form nested lists.

We can convert a list to a tuple using the `tuple()` function, and we can convert tuples to lists using the `list()` function:

```
>>> tuple(a)
(1, 2, 5, 6, 1, -1, 0, 10)
>>> list(tp2)
['In', 'this', 'part', 'of', 'the', 'country?']
```

## Ranges

Ranges are slightly different from tuples and lists — they represent a sequence of integers, and are created by using the `range` function. It has the following syntax:

```
range(start, end, step)
```

- `start` is the starting index of the slice, this will be the first element included.
- `end` is the ending index of the slice; the returned range of elements will *not* include the element at the `end` index.
- `step` is the integer increment between each index. The step size is optional, and if omitted defaults to 1.

We can convert range objects into lists or tuples to see the integer elements in the sequence:

```
>>> tuple(range(5,10))
(5, 6, 7, 8, 9)
>>> list(range(-2,5,1))
[-2, -1, 0, 1, 2, 3, 4]
```

Furthermore, we can use a **negative** step size to create a range of integers in descending order:

```
>>> list(range(3,-2,-1))
[3, 2, 1, 0, -1]
```

### 3.5.2 Sequence Operations and Intrinsic Functions

Sequences can be acted on with the **concatenation operator** `+`, which acts on two sequences joining them together:

```
>>> [1,2,6] + ['hi', True, 8]
[1, 2, 6, 'hi', True, 8]
```

► Look familiar?

Strings are a **special case** of sequences, so everything in this section applies equally to strings. We even covered how these apply specifically to strings in the previous section, when we considered substrings and string concatenation!

We also have the **repetition operator** `*`, which repeats a sequence a specified number of times:

```
>>> (1, 2, 3)*5
(1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> ["yes"]*5 + ["no"]
['yes', 'yes', 'yes', 'yes', 'no']
```

The **membership operators** we came across earlier, `in` and `not in`, also work on sequences:

```
>>> -1 in [5,6,-1,2]
True
>>> True not in (True, False, 0, 0.543)
False
```

We also have several intrinsic functions in Python that can be applied to sequences (Table 3.10), as well as some methods common to all sequences.

|                       |                                                                                                       |
|-----------------------|-------------------------------------------------------------------------------------------------------|
| <code>len()</code>    | Returns the length of the data structure; i.e. the number of elements contained in the data structure |
| <code>max()</code>    | Returns the element of the data structure with the largest numeric value                              |
| <code>min()</code>    | Returns the element of the data structure with the smallest numeric value                             |
| <code>sorted()</code> | Sorts the data structure numerically, from lowest to highest numeric value                            |

Table 3.10 Python intrinsic data structure functions

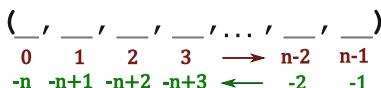


Figure 3.3 How Python indexes the elements in a data structure. The indices from left to right are in red, and the indices from right to left are in green

|                                            |                                                                                                                                                                                                    |
|--------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>t.index(val, [start, [stop]])</code> | Returns the index of the first element found with the given value. If <code>start</code> and <code>stop</code> are provided, the search will only be in the index range <code>[start, stop)</code> |
| <code>t.count(val)</code>                  | Returns the number of occurrences of given value                                                                                                                                                   |

Table 3.11 Python intrinsic methods for sequences

### 3.5.3 Indexing and Slicing

When working with sequences such as tuples, there are several ways we can access the stored data:

1. We can access individual elements through **indexing**, by using square brackets to indicate the element index.

In Python, indexing of data structures **always starts at 0**; thus, a tuple with  $n$  elements will be indexed by the indices  $0, 1, 2, \dots, n - 1$ . So, an index of  $i$  will correspond to the  $(i - 1)$ th element:

```
>>> c = [12, 13, 14, 15, 16, 17, 20, 5, -1]
>>> print(c[0], c[2], c[3])
12 14 15
```

Furthermore, we can use **negative indices** to access elements in reverse, from right to left, without having to know how long the data structure is. Negative indices index from right to left, with  $-1$  indexing the last element, and  $-n$  indexing the first element:

```
>>> c[-1]
-1
>>> c[-4]
17
```

What if we want to extract multiple elements at once?

2. We can access a range of elements through a process known as **slicing**, returning another tuple. This can be done by using the notation `[start:end:step]`, again enclosed in square brackets.

- `start` is the starting index of the slice, this will be the first element included.
- `end` is the ending index of the slice; the returned range of elements will *not* include the element at the `end` index.
- `step` is the integer increment for the list index.

For example,

```
c[2:4]      # (c[2], c[3]) = [14, 15]
c[1:8:2]    # (c[1], c[3], c[5], c[7]) = [13, 15, 17, 5]
c[-1:-5:-1] # (c[-1], c[-2], c[-3], c[-4]) = [-1, 5, 20, 17]
```

Note that the slice lower bound `start`, upper bound `end`, and step size `step` are all *optional* — if not specified, the following defaults are used:

- Slice lower bound default: `start = 0`
- Slice upper bound default: `end = n - 1`
- Slice step size default: `step = 1`

For example, the following are *equivalent*:

```
c[3:] ≡ c[1:8] = [15, 16, 17, 20, 5, 1]
c[::2] ≡ c[0:8:2] = [12, 14, 16, 20, -1]
```

► You can think of a negative index  $-i$  as taking the  $i$ th element from the end of the list.

► Slicing in reverse (from right to left) always requires a negative stride `istep < 0`.

For a tuple of length  $n$ ,

`tuple[-i:-j:-d]`

will extract elements with index  $n - i$  to  $n - j + 1$ , in increments of  $-d$ .

► `seq[i:j:d]`

will extract elements with index  $i$  to  $j - 1$ , in steps of  $d$ .

### Nested lists

When a list contains elements which are also lists, this is known as a nested list. For example, consider the list

```
a = [[0, 1, 2],
     [3, 4, 5],
     [6, 7, 8]]
```

We can access the elements of the nested lists using multiple indices:

```
a[1] = [3, 4, 5]
a[1][2] = 5
```

### 3.5.4 Sequence Unpacking

When working with functions that return tuples or lists, it is common to use **sequence unpacking** in order to initialise a collection of variables with values given by the tuple elements. For example, consider a function `spacetime_coordinates()` that returns a tuple of 4 variables;  $(x, y, z)$  coordinates and a time value. In order to set the variables `x, y, z, t` according to this function, we use sequence unpacking:

```
x, y, z, t = spacetime_coordinates()
```

For sequence unpacking to be valid, there must be the same number of variables on the left hand side as there are elements on the right hand side.

- ▶ Strings are another type of immutable sequence.
- ▶ A useful phrase to remember the distinction between tuples and lists, from Python developer [Raymond Hettinger](#):

*"Loopy lists and structy tuples"*

#### Lists versus tuples

'Well' you're probably saying. 'Now that you've shown me lists, no need to keep storing all that information about tuples in my head!' Oh no, wait! Tuples and lists *both* have their places in modern Python programming.

Let's try changing one of the elements of a tuple:

```
>>> a = (1, 2, 5)
>>> a[0] = 7
TypeError: 'tuple' object does not support item assignment
```

What happens if we try modifying a list instead?

```
>>> a = [1, 2, 5]
>>> a[0] = 7
>>> print(a)
[7, 2, 5]
```

As it happens, lists are what are known as **mutable** sequences, meaning we can modify them after they have been created and stored in memory. Tuples, on the other hand, are **immutable**; they cannot be modified after being created.

Tuples, being immutable, are usually used to store a **known, unchanging** number of **heterogeneous elements** — that is, elements which are not all necessarily the same type. For example, say we want to store the population, area (in km<sup>2</sup>), and state of the Australian city of Perth:

```
>>> perth = (1943858, "6417.9", "Western Australia")
```

Lists, being mutable, are instead conventionally used to store an **unknown or changing** of **homogeneous elements** — elements which

all share the same type. For example, a list of strings could be used to store the lines of text in a text file.

### 3.5.5 List Manipulation

As we've seen, lists are remarkably similar to tuples. Both are data structures that can contain ordered sequences of elements, can be acted on by the same operators and intrinsic functions, and indexing and slicing works exactly the same. Lists, however, unlike tuples, are **mutable** — the elements of a list can be altered in-place after the list has been created. This one small difference results in lists being incredibly versatile, and one of the most commonly used data structures in Python.

#### Modifying Lists

Using indexing and slicing, we can modify the elements of a list in Python. For example, consider the list variable `a` defined above:

```
>>> a[0] = -10
>>> a[4:] = [11, 12, 13, 14]
>>> print(a)
[-10, 2, 6, 1, 11, 12, 13, 14]
```

We can use a step size to change every second element, and even delete elements using the `del` statement:

```
>>> a[::2] = [0, 0, 0, 0]
>>> del a[-1]
>>> print(a)
[0, 2, 0, 1, 0, 12, 0]
```

► The `del` statement can also be used to delete whole variables from memory; for example,

```
a = 6.5
del a
```

#### List Methods

In addition to the methods available to all data structures (Table 3.11), there are some additional methods available to `list`, due to it being mutable.

- `list.append(x)`

This will append the element `x` to the end of the given list.

For a list `a`, this is equivalent to `a[len(a):] = [x]`.

- `list.extend(iter)`

This extends the list with an iterable `iter` — the elements of `iter` are appended to the list. The iterable can be a `range`, `tuple`, or `list`.

For a list `a` and iterable `b`, this is equivalent to `a[len(a):len(a)] = b`.

► `list.extend()` can also be written using a concatenation assignment operator, `a += b`

- `list.insert(i, x)`

This inserts the element `x` into the list at position given by the index `i`.

For a list `a`, this is equivalent to `a[i:i] = x`.

- `list.remove(x)`

This removes the first occurrence of element `x` from a list

- `list.clear()`

This deletes all elements of a list, leaving just an empty list.

For a list `a`, this is equivalent to `del a[:]`

- `list.copy()`

This copies a list, creating the same list in a **new memory location**.

For example, to create a copy of list `a`, this is equivalent to `b = a[:]`

- `list.reverse()`

This reverses all items from a list **in-place**. For example,

```
>>> a = [0, 1, 2, 3]; a.reverse(); print(a)
[3, 2, 1, 0]
```

- `list.pop([i])`

This will output the element at index position `i` from the list, while also deleting it from the list. Note that the index argument `i` is optional; if not provided, the default will be to pop the last element in the list

- `list.sort()`

Sorts the list **in-place**, and accepts the same arguments as `sorted()`

### Functions versus methods

We briefly discussed methods in the previous section, when introducing string methods. But what's the difference between a function and a method?

A function is something that you call, and apply to an object along with any other arguments. For example, we already came across the `sorted()` function, which sorts data structures:

```
>>> l1 = [4, 56, 2]
>>> sorted(l1)
[2, 4, 56]
```

Methods are also functions, but they are instead *attached* to the object they act on. We call them by writing the variable we have used to store an object, followed by a period, and then the name of the method:

```
>>> l1 = [4, 56, 2]
>>> l1.sort()
```

```
>>> print(l1)
[2, 4, 56]
```

Different data types in Python support different methods. To make it clear which objects a method is available for, they will always be written in the format `object.method([arguments])`.

### List copying

You might be wondering why the `list.copy()` method is required — can't we just do

```
>>> a = [1, 2, 6, 5]
>>> b = a
```

Not quite, as this produces a **reference** instead of a copy; `a` and `b` both point to the same memory location:

```
>>> print(id(a) == id(b))
True
```

This then suffers from the side effect that changing elements in `a` will change the corresponding element in `b`! For example,

```
>>> a[2] = 9
>>> print(b)
[1, 2, 9, 5]
```

In order to create a copy at a new memory location, we can use `b = a.copy()` or `b = a[:]` to create a **shallow copy**. Give this a go, and compare the output of `id(a)` and `id(b)`. What happens now if you change an element of `a`?

The reason this is called a shallow copy is that if any of the elements of `a` are data structures themselves, for example the last element in `a = [1, 2, [0.6, 1.5]]`, these *nested* elements will *not* be copied by a shallow copy, and instead will be stored as a reference. So, to make a copy of a **nested list**, a **deep copy** must be made. This can be done by importing the `copy` module:

```
>>> import copy
>>> b = copy.deepcopy(a)
```

### 3.5.6 Sets

- ▶ Try printing out these example sets. What do you notice?

Since sets are unordered and contain only unique elements, Python will print the set *sorted* (from lowest to highest number or character), and will only display each unique element once.

Sets, constructed via the intrinsic Python function `set()`, are a data structure designed to work in the same manner as mathematical sets — they contain a series of **unordered and unique immutable elements**. Because they are unordered, they are not sequences like lists and tuples — they are collections. Furthermore, since the elements of a set must be immutable, they cannot contain lists. However, sets *themselves* are a mutable type.

To construct sets, we use curly brackets:

```
>>> s1 = {1, 5, 7, 2, 11}
```

We can also use the `set()` function to convert a list or tuple to a set:

```
>>> s1 = set(["a", "c", "g", "b", "a"])
>>> print(s1)
{'a', 'b', 'c', 'g'}
```

### Set Methods and Operators

The membership operators (`in` and `not in`), as well as the intrinsic functions `len()`, `min()`, and `max()` we saw previously, apply to sets. Sets also have the following inbuilt methods, designed to replicate the set operations we see in mathematics.

- `set.add(x)`

This adds the element `x` to the set. If `x` already exists in the list, no change is made.

- `set.update(iter)`

This adds multiple elements contained within an iterable `iter` — the iterable can be a `range`, `tuple`, or `list`.

- `set.remove(x)`

This removes the element `x` from the set. If this element does not exist, an error is thrown.

- `set.discard(x)`

Similar to `set.remove(x)`, but no error is thrown if the element does not exist.

- `set1.union(set2)`

`set1 | set2`

Returns the union of `set1` and `set2`; that is, it outputs a set containing all elements of the sets.

- `set1.intersection(set2)`

`set1 & set2`

Returns the intersection of `set1` and `set2`, outputting a set containing only elements common to the two sets.

- ▶ Slicing and indexing is not supported by sets, since the elements have no intrinsic ordering.

- ▶ The concatenation and repetition operators **cannot** be used on sets!

- ▶ The set operators can also be used as **assignment operators**, in order to perform the operation and update the original set. For example,

`s1 = s1 | s2`

can be written

`s1 |= s2`

- `set1.difference(set2)`

`set1 - set2`

Returns the set difference between the sets, outputting a set containing the elements of `set1` except those also contained in `set2`.

- `set1.symmetric_difference(set2)`

`set1 ^ set2`

Returns the symmetric difference between the sets, outputting a set containing the elements of the sets which are *not* in their intersection. This is equivalent to `(set1-set2)|(set2-set1)`.

- `set1.issubset(set2)`

`set1 <= set2`

Returns `True` if `set1` is a subset of `set2`.

- `set1.issuperset(set2)`

`set1 >= set2`

Returns `True` if `set2` is a subset of `set1`.

### 3.5.7 Dictionaries

So far, we've covered both tuples and lists, with elements accessed via indexing, as well as sets, which cannot be indexed. Dictionaries provide a mutable data structure that can be accessed through a third method: **mapping**.

Like sets, dictionaries are constructed using curly brackets with each element consisting of a **key** and a **value**, separated by a colon:

```
>>> d1 = {key1:val1, key2:val2:, key3:val3}
```

When constructing dictionaries, values can be of any type, while keys are restricted to only being immutable types (meaning no lists, sets, or dictionaries can be used as keys!). So, for example, the following is a valid dictionary:

```
>>> d1 = {"City": ["Perth", "Toronto"], "ID": 43, 5: 9.1}
```

Values corresponding to a particular key are then accessed using square brackets:

```
>>> d1["ID"]
43
>>> d1["City"]
["Perth", "Toronto"]
>>> d1[5]
9.1
```

► As both sets and dictionaries use curly brackets, Python uses the existence of key-value pairs to distinguish between the two.

However, an empty dictionary and an empty set looks the same! By default, `{}` initialises an empty dictionary. To initialise an empty set, use `set()`.

- ▶ Like other data structures, we can use the Python intrinsic function `len()` to determine the number of key-value pairs in a dictionary.

The functions `min()` and `max()` can be used to determine the minimum and maximum **key** in the dictionary, but will only provide useful results if all keys are of the same type.

#### Making sure keys exist

If you try and access a dictionary key that does not exist, you will get a Python `KeyError`, and your Python program will terminate.

To avoid this, you can use the `in` and `not in` operators to check if a key exists before accessing a dictionary:

```
if key1 is in dict1:
    print(dict1[key1])
```

- ▶ The `copy` method produces a **shallow copy** of the dictionary. See **List Copying** on page 113 for more details.

Dictionaries can also be created by using the `dict()` function, either by using keywords to denote the key-value pairs,

```
>>> d2 = dict(key1=val1, key2=val2, key3=val3)
```

or by passing as an argument a list or tuple containing nested `(key, value)` elements:

```
>>> l2 = [[key1, val1], [key2, val2], [key3, val3]]
>>> d2 = dict(l2)
```

## Modifying Dictionaries

ince dictionaries are mutable, we can modify dictionary key-value pairs, by indicating the key we want to modify:

```
>>> d1["ID"] = 12
>>> print(d1)
{"City": ["Perth", "Toronto"], "ID": 12, 5: 9.1}
```

If the key on the left hand side does not already exist, then the key and its assigned value are both added to the dictionary. We can also delete keys using the `del` statement:

```
>>> del d1["City"]
>>> print(d1)
{"ID": 12, 5: 9.1}
```

## Dictionary Methods

Python also provides dictionary method for more advanced dictionary manipulations. We will list a few of the most common ones here, to find out more, see one of the texts in the ‘further reading’ section at the end of the chapter.

- `dict.clear()`

Deletes all key-value pairs in the dictionary, resulting in an empty dictionary.

- `dict.copy()`

This copies all key-value pairs from a dictionary, creating an identical dictionary in a **new memory location** — modifying key-value pairs of the original dictionary will not affect the copy, and vice versa.

- `dict.items()`

Returns a **dynamic view** of the dictionary key-value items, provided as `(key, value)` tuples. It is ‘dynamic’ because if the dictionary changes, the dynamic view assigned to a variable earlier will also change. The dynamic view can be converted to other data structures: `list(dict.items())` or `tuple(dict.items())`.

- `dict.keys()` and `dict.values()`

Similar to `dict.items()`, however provides *only* the keys or values respectively.

- `dict.update()`

Updates a dictionary with a new value for a given key; if the key does not exist, the key-value pair is added to the dictionary.

You might be wondering what the point of the `update` method is — haven't we already covered how to add and modify dictionary items? `dict.update()` is actually much more powerful, as it allows the modification of dictionaries using lists of key/value pairs or even *other dictionaries* — allowing you to add or update all the keys from one dictionary to another.

## 3.6 Loops and Flow Control

We have already come across conditionals and if statements, which allows us to ‘branch’ our Python program depending on whether a condition is met. Loops and flow control are another useful construct, which allows us to repeat certain sections of code a predetermined number of times, or until a specific condition is satisfied.

### 3.6.1 The `while` loop

- ▶ Like `if` statements, loops can also be infinitely nested.

```
while condition:
    # Python code block
```

If the condition evaluates to `True`, the indented code-block will be repeatedly executed. As soon as the condition is no longer true, the while loop exits.

#### Example 3.9 : `while` loop

Print out all square numbers less than 60.

#### Solution:

```
#!/usr/bin/env python
i = 1

while i**2 < 60:
    print(i**2)
    i += 1
```

### 3.6.2 `for` loops

The `for` loop allows you to run a particular section of code, iterating over an **iterable**:

```
for i in iter:
    # Python code for each value of i
```

- ▶ `range()` is a particularly useful iterable when writing for loops.

What exactly is an iterable? An iterable is any object capable of returning its constituent elements one at a time. Sound familiar? All sequences, such as lists, tuples, strings, and ranges, are iterables, as are sets and dictionaries.

For example, try running the following Python code, which iterates over a string:

```
s = "Hello world!"
for c in s:
    print(c)
```

**Example 3.10 :** `for` loop

Calculate the squares of all numbers between 5 and 10.

**Solution:**

```
#!/usr/bin/env python
for i in range(5,11):
    print(i**2)
```

Recall that `range(5,11)` will iterate over 5, 6, 7, 8, 9, 10; *one less* than the second argument in `range`.

**3.6.3 Flow Control Statements**

Flow control statements allow for additional control in `while` and `for` loops; in Python, available flow control statements are `continue`, `break`, and `pass`. Check out Table 3.12 for detailed descriptions of these control statements.

|                       |                                                                                                            |
|-----------------------|------------------------------------------------------------------------------------------------------------|
| <code>continue</code> | All remaining lines in the current loop iteration are to be skipped, and the next loop iteration begins    |
| <code>break</code>    | The loop is terminated at the current iteration, the <code>else</code> statement is skipped                |
| <code>pass</code>     | Do nothing! This is a useful placeholder when you are prototyping code blocks that require indented syntax |

**Table 3.12** Python flow control statements

Wait, hang on. What do you mean ‘the `else` statement is skipped’? `else` statements are used in `if` statements, not loops! Do I get a finders fee for finding this flagrant error?”

Well, actually, you can use `else` statements in loops! We didn’t introduce it earlier, because it usually is not needed unless you use `break` within your loop. The code inside the `else` statement runs after the loop only if the loop ends normally. If it ends prematurely via a `break` statement, the `else` statement will be skipped.

```
for i in range(0,11):
    pass
    if condition: break
else:
    print("loop completed!")
```

For example, in the above code snippet, we have an `else` statement as part of the `for` loop; if the condition in the nested if statement is false, then the code within the else statement will execute upon the last loop iteration. However, if the condition is true, then the loop will exit straight away, and the else statement will be ignored.

### Example 3.11

Complete the following code to print all square numbers between 10 and 60.

```
#!/usr/bin/env python
i = 0

while True:
```

**Solution:** A `while` loop with condition simply ‘`True`’ will loop infinitely — the condition by definition will never be `False`! In order to work around this, we must use flow control statements.

```
#!/usr/bin/env python
i = 0

while True:
    i += 1
    isq = i**2

    if isq < 10:
        continue
    elif isq > 60:
        break

    print(isq)
```

### 3.6.4 Iterator Functions

As noted before, an `iterable` in Python is any object that returns its elements one-by-one, allowing us to iterate over it during a `for` loop. Let’s look into this with a bit more detail; how does each data structure iterate?

- ▶ Beware! Since sets and dictionaries are **unordered** data structures, they might iterate in a different order to the order in which they were defined.

- **Strings:** iterates character-by-character
- **Tuples:** iterates through each element
- **Lists:** iterates through each element
- **Sets:** iterates through each element
- **Dictionaries:** iterates through keys (using the iterable `dict.keys()`), values (using the iterable `dict.values()`), or `(key,value)` tuples (using `dict.items()`)

**Example 3.12**

```
revenue = {"Jan":11000, "Feb":16000, "Mar":10000, "Apr":23000}
```

is a dictionary representing business revenue raised each month from January to April, while

```
costs = {"Feb":15000, "Mar":7000, "Apr":5000, "May":1520}
```

represents the business costs each month from February to May.<sup>2</sup>

Create a new dictionary called `profits`, and use a `for` loop to store the profit for the overlap months February to April.

- ▶ If you attempt to iterate directly over a dictionary instead of specifying `dict.items()`, `dict.values()`, or `dict.keys()`, the iteration will by default be over **keys only**.

**Solution:**

```
profit = dict()
for key, val in revenue.items():
    if key in costs:
        profit[key] = val - costs[key]
```

Python also provides some nifty intrinsic functions for modifying iterables, see Table 3.13.

|                          |                                                                                  |
|--------------------------|----------------------------------------------------------------------------------|
| <code>zip()</code>       | Combines and loops through multiple iterables simultaneously, element by element |
| <code>enumerate()</code> | Prepends the current index to the current element                                |
| <code>reversed()</code>  | Iterates through an iterable in reverse order                                    |

**Table 3.13** Python intrinsic iterable functions

How do these work in practice? Let's look at `zip()` and `enumerate()` in more detail, as these are incredibly useful functions that you might find yourself using again and again.

- `zip()`: combines multiple iterables  $(a_1, a_2, \dots), (b_1, b_2, \dots), \dots$ , to return the iterable  $((a_1, b_1, \dots), (a_2, b_2, \dots), \dots)$ .

```
>>> a = [2, 4, 1]
>>> b = ['a', 'b', 'c']
>>> for i,j in zip(a,b):
...     print(i,j)
2 a
4 b
1 c
```

<sup>2</sup>This business has particularly shoddy bookkeeping.

- `enumerate()`: takes an iterable  $(x_1, x_2, \dots)$  and prepends the index to each element, returning  $(0, x_1), (1, x_2), \dots$ . This is a convenient way of iterating over both indices and values of lists simultaneously.

```
>>> b = ['a', 'b', 'c']
>>> for idx, val in enumerate(b):
...     print(idx, val)
0 a
1 b
2 c
```

### 3.6.5 List Comprehension

Say we have been given a systematic method to populate the elements of a list; for example, we would like to define a list `squares` that contains all the squares between 0 and 100. One way we could do this is by starting with an empty list, and using a `for` loop to append each element:

```
>>> squares = []
>>> for i in range(0, 101):
...     squares.append(i**2)
>>> print(squares)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

► For those who have come from the Fortran chapter, Python's list comprehension is the same idea as constructing a Fortran array using an implied do loop.

However, Python provides a more concise way to populate lists. This is known as **list comprehension**, and looks like this:

```
[function(i) for i in iter]
```

Unlike a regular for loop, we start by giving the expression for each element, followed by the `for` statement. So, rewriting the above square number example using list comprehension,

```
>>> [i**2 for i in range(0, 11)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

You can also use **filters** in list comprehensions, in order to select only elements that satisfy a condition.

```
>>> [function(i) for i in iter if condition]
```

For example, if we are interested in creating a list of all *odd* squares of integers in the range 0–100:

```
>>> [i**2 for i in range(0, 11) if i**2 % 2 != 0]
[1, 9, 25, 49, 81]
```

The `if` statement comes *after* the `for` loop, filtering out the list elements that do not satisfy the condition.

**Example 3.13** (List comprehension filters)

Create a list containing the squares of all integers from 0 to 100 that are divisible by 3 and 7.

**Solution:**

```
[i**2 for i in range(0,101) if (i**2 % 3 == 0 and i**2 % 7 ==0)]
```

*Output:* [0, 441, 1764, 3969, 7056]

If we need to include a second element expression if the condition is *not* met, then we use a slightly different syntax. Instead, we use a **conditional statement** *before* the **for** loop:

```
>>> [f(i) if cond1 elif cond2 g(i) else h(i) for i in iter]
```

This is equivalent to writing

```
>>> a = []
>>> for i in iter:
...     if cond:
...         a.append(f(i))
...     elif cond2:
...         a.append(g(i))
...     else:
...         a.append(h(i))
```

which, you might agree, is *much* more convoluted than simply using the list comprehension solution!

- ▶ Like standard **if-else** statements, the **elif** and **else** are *optional* in conditional list comprehension.

**Example 3.14** (List comprehension conditionals)

Use list comprehension to construct a list containing the tuples  $(x, H(x))$  for integers over the range  $-10 \leq x \leq 10$ , where  $H(x)$  is the Heaviside step function ( $H(x) = 0$  for  $x < 0$ ,  $H(x) = 1$  for  $x \geq 0$ ).

**Solution:**

```
[(x,0) if x<0 else (x,1) for x in range(-10,11)]
```

Python also allows for **nested loops** in list comprehension, similar to nested **for** loops. For example, constructing a list using the following nested list,

```
>>> a = []
>>> for x in iter1:
...     for y in iter2:
...         a.append(f(x,y))
```

can be done using list comprehension:

```
>>> [f(x,y) for x in iter1 for y in iter2]
```

### Example 3.15 (List comprehension nested loops)

Use list comprehension to construct a list containing the tuples  $(x, y)$  for integers over the range  $0 \leq x \leq 10$  and  $0 \leq y < x$ .

**Solution:**

```
[(x,y) for x in range(0,11) for y in range(0,x)]
```

Similarly, we can also perform **nested list comprehensions**.

### Example 3.16 (Nested list comprehension)

Use list comprehension to return the transpose of the array

```
a = [[1, 2, 3],  
     [4, 5, 6],  
     [7, 8, 9]]
```

**Solution:**

```
[[row[i] for row in a] for i in range(len(a))]
```

- `range(len(a))` will provide the iterator  $i = 0, 1, 2$ .

## General Comprehensions

While almost universally referred to and used with lists, Python's comprehension syntax actually generalises to the other data structures we've covered as well, such as tuples, sets, and dictionaries.

- **Tuple comprehension:** indicated by the use of round brackets/parenthesis:

```
>>> (i**3 for i in range(10))
```

- **Set comprehension:** indicated by the use of curly brackets/braces.

```
>>> {x**2 for x in range(0,100) if x%3==0}
```

- **Dictionary comprehension:** indicated by the use of curly brackets/braces, and colons to distinguish the dictionary keys and values.

```
>>> names = ["John", "Jamie", "James", "Jim", "Jonathan"]  
>>> {x:len(x) for x in names}
```

As with list comprehensions, filters, conditions, and nested loops can all be used.

## 3.7 Input and Output

At some point, hard-coding data into a program becomes too lengthy, takes focus away from the algorithm itself, and makes it difficult to change input data on the go. This is where considering various methods of input and output becomes important — for example, by reading the input data from a file, a program can be run multiple times on different data sets without having to modify the code. Additionally, writing the output data to a file is significantly more convenient and less prone to error when large amount of data is involved (while also providing the flexibility to use external data analysis software).

### 3.7.1 Opening and Closing Files

In order to make a file available to Python, it first needs to be ‘opened’, using the aptly named `open` statement,

```
>>> f = open("filename.txt", "r+")
```

Here, the first argument is a string which passes the location and filename of the file to be written/read, and the second argument is a string indicating the **mode** (see Table 3.14).

|                   |                                                                                                                                                                                                                            |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>"r"</code>  | <b>Reading mode:</b> used for reading files, <code>open()</code> will start at the beginning of the file. <i>Default</i>                                                                                                   |
| <code>"w"</code>  | <b>Writing mode:</b> if it already exists, the file is erased before writing                                                                                                                                               |
| <code>"a"</code>  | <b>Appending mode:</b> used for writing data only; unlike <code>"w"</code> , new data is written to the <i>end</i> of an already existing file                                                                             |
| <code>"r+"</code> | <b>Read/write mode:</b> used for reading and writing to the same file. If the file does not exist, it is <i>not</i> created                                                                                                |
| <code>"a+"</code> | <b>Read/append mode:</b> used for reading and writing to the same file; writing is automatically appended to the end of current contents. If the file does not exist, it is created                                        |
| <code>"t"</code>  | <b>Text mode:</b> the file will be opened in text mode, and the contents converted into a string. This can be appended to any of the read/write modes above, e.g. <code>"wt"</code> or <code>"a+t"</code> . <i>Default</i> |
| <code>"b"</code>  | <b>Binary mode:</b> the file will be opened in binary mode, and the contents provided as-is. This can be appended to any of the read/write modes above, e.g. <code>"rb"</code> or <code>"a+b"</code>                       |

Table 3.14 Python `open()` mode arguments. The default is `"rt"`

Once the file is no longer needed, it can be closed using the `close()` statement:

```
>>> f.close()
```

While it is not strictly necessary to close files (this is done automatically when the program terminates), closing files when they are no longer needed allows other programs to access them, and may avoid potential memory issues if working with large quantities of data. So, it is **always** good programming practice to close your files when they are no longer needed!

The best method for opening and closing files is by using Python's `with` statement. The `with` statement will properly close your files *automatically* once the code block inside `with` is complete, even if there is an error or exception inside your code. So, try and always use the `with` statement to read and write to files. It works as follows:

```
>>> with open("filename.txt", "a+") as f:
...     # file input and output operations on file f
```

### 3.7.2 Reading from a Text File

Once we have opened the text file, we can read each line simply by iterating over the file object:

```
>>> with open("filename.txt", "r") as f:
...     for line in f:
...         print(line)
```

Alternatively, we can access the contents of the file through several available **file methods**, listed in Table 3.15. Of these methods, `readlines()` provides a clean, clear, and easy to use approach to extracting the lines of a file:

```
>>> with open("filename.txt", "r") as f:
...     contents = f.readlines()
```

#### Example 3.17

Write a Python program that reads in a one-column list of numbers, and calculates the sum.

**Solution:**

```
# extract the data
with open("data.txt", "r") as f:
    data = f.readlines()

# use a for loop to calculate the sum
```

#### File methods

|                              |                                                                |
|------------------------------|----------------------------------------------------------------|
| <code>f.read()</code>        | Returns <code>str</code> containing contents of <code>f</code> |
| <code>f.readline()</code>    | Returns successive lines from <code>f</code> as strings        |
| <code>f.readlines()</code>   | Returns all lines in <code>f</code> as a list of strings       |
| <code>f.write(s)</code>      | Writes string <code>s</code> to <code>f</code>                 |
| <code>f.writelines(l)</code> | Write list of lines <code>l</code> to <code>f</code>           |

Table 3.15 Python file methods

```
sum = 0
for i in data:
    sum += float(i)

print(sum)
```

### 3.7.3 Writing to a Text File

- ▶ Remember, the `open()` mode arguments you used when opening the file determine whether the data you are writing to the file is appended to the end, or overwritten from the beginning!

Similarly, we can also use file methods listed in Table 3.15 to write text to a file; namely, `write(s)`, which writes a string `s` to the file, or `writelines(l)`, which writes a list of strings (for example, `["Hello", "world!"]`) as individual lines of the file.

#### Example 3.18

Write a Python program that calculates the 5 times table, and outputs it to a file `5times.txt`

#### Solution:

```
# generate the 5 times table using list comprehension
l = ["5 x {} = {}".format(i, 5*i) for i in range(1,11)]

# create the file for writing
with open("5times.txt", "w") as f:
    # write to the file
    f.writelines(l)
```

### Serialisation: JSON and Pickle

Python's file methods have one disadvantage, however; by only reading and writing strings, they aren't very suitable for more complex data types, such as lists and dictionaries (which may be arbitrarily nested, and even contain other lists and dictionaries).

One solution is to use the `json` module; this module automates the process of converting complex data structures into strings using the JSON data storage format (a process known as **serialising**), and converting the string representation back into Python data structures (**deserialising**).

```
import json

var1 = {'name': 'josh', 'route': ['PER', 'HKG', 'YYZ']}

with open('var1_data.txt', 'w') as f:
    # serialise var1 to the file
    json.dump(var1, f)

# read from the file
with open('var1_data.txt', 'r') as f:
    # deserialise and save to a variable
    var2 = json.load(f)
```

Try running the above code snippet — you should find that `var1` and `var2` are identical, as expected. Open the JSON file, and compare the format to the standard Python dictionary.

There are still some cases where JSON serialisation might fail, as it can only serialise a subset of all available Python data structures and types. In this case, the `pickle` module can be used — it uses a similar syntax to above, but unlike `json`, this serialises any arbitrary Python object to a *binary* file, not a text file. As a result, it is Python specific, and cannot be used by other applications. For more information on `pickle`, see the [online documentation](#).

► JSON is short for  
JavaScript Object Notation

## 3.8 Functions

Sometimes, a calculation must be made numerous times in the same program. In order to increase code readability and maintainability (i.e. avoid having to change the code that does the same thing in numerous locations) wouldn't it be nice if we could create our own functions?

Well, guess what! We can. Defining functions is incredibly easy in Python, and simply requires three things:

- ▶ Functions don't have to return a value.  
For example, a function that writes an argument to a file has nothing to return, and may omit the `return` statement.  
Functions that omit the `return` statement will simply return `None`.
- ▶ You may also set a variable to a function's `return` value directly:  
`x = my_func(5, 6)`  
If the function returns multiple values in the form of a tuple, for example  
`return 0.84, True, "Yes"`  
then we can use sequence unpacking:  
`x, y, z = my_func2(1)`
- ▶ Python doesn't have a built in square root function, so we need to import one from the `math` module. We will cover additional mathematical functions in the next section

- The `def` statement; this is how Python knows you are creating a function.
- The function name, and the arguments it will receive in brackets. For example, `my_func(a,b)`.
- After the function name, we have a colon :, indicating that an indented codeblock containing the function follows. Arbitrary lines of code can follow.
- Finally, the `return` statement is used to return the result of the function.

Let's have a look at a quick example:

```
>>> def my_func(a,b):
...     return a+b
```

This simple function accepts two arguments, adds them together, and returns the result. Once it has been defined, you can call it like any other Python function:

```
>>> print(my_func(5,6))
11
```

### Example 3.19

Write an external function that normalises a state vector such that  $\langle \psi | \psi \rangle = \sum_i |\psi_i|^2 = 1$ , and use it to normalise the two-basis state vector  $\psi = |0\rangle + i|1\rangle$ .

**Solution:**

```
from math import sqrt

def norm(psi):
    # calculate the vector norm
    N = sqrt(sum([abs(i)**2 for i in psi]))
    return psi/N

psi = [1, 1j]
print(norm(psi))
```

### 3.8.1 Local and Global Scope

When you define a new variable inside a function, it is local to that function - i.e. it can be seen by that function only. This is referred to as **scope** — the visibility of a function or variable to other functions or variables.

- **Global scope:** variables and functions defined in the main program have global scope, they can be accessed anywhere in the program.
- **Local scope:** variables and functions defined within an indented function block can only be accessed from inside that function

For example, consider the following:

```
>>> def sum_list(x):
...     sumTmp = 0
...     for j in x:
...         sumTmp += j
...     return sumTmp

>>> sum_list(range(0,101))
5050
>>> sumTmp
NameError: name 'sumTmp' is not defined
```

The above function accepts a list argument `x` and sums its elements, by using the local variable `sumTmp` and a `for` loop. Since `sumTmp` is a local variable in the `sum_list()` function, we cannot access it in the global scope.

However, global variables, defined outside of functions, can be used and accessed by functions without being passed as arguments.

```
>>> pi = 3.141592
>>> def deg_to_radian(x):
...     rad = x*pi/180
...     return rad

>>> deg_to_radian(30)
0.5235986666666667
>>> pi
3.141592
>>> rad
NameError: name 'rad' is not defined
```

In the above, `pi` is a global variable that can be accessed inside all user defined functions, while `rad` is a local variable, accessible only inside the function it is defined in.

### 3.8.2 Function Arguments

We saw above that we must pass the number of arguments when defining a function — these are given **parameter names**, which are used to refer to the argument within the function definition. Python also allows us several more advanced features when working with function arguments.

#### Default Arguments

You can set the default value of an argument, so that if it is not supplied to the function at run time, the default value is used. This is done with the syntax `arg=default_value`. For example, consider a function that raises any number to  $n$ th power, but defaults to the 2nd power if no value of  $n$  is given:

- ▶ You can use the default argument syntax to create **optional arguments** if you wish. For example, we could define

```
def my_func(x,n=None):
```

and use an if statement to determine when `n` is `None`.

```
>>> def raise_to_power(x,n=2):
...     return x**n

>>> raise_to_power(2,3)
8
>>> raise_to_power(2))
4
```

Default arguments must *always* be listed in the function definition after non-default arguments; otherwise, Python will return a **SyntaxError**.

#### Keyword Arguments

We can also *call* a function with **keyword arguments**; that is, instead of having to remember the order we need to give certain arguments, we simply use the parameter name to tell Python which argument we are passing. With keyword arguments, the order of arguments no longer matter!

```
>>> raise_to_power(x=5,n=2)
25
>>> raise_to_power(n=3,x=5)
125
```

#### Variable Number of Arguments

Sometimes, it is also useful to be able to define a function which accepts any number of arguments. ‘Well, we can all dream’ we imagine you are saying. No! Wait! Python actually let’s us do this. We can specify that a parameter refers to a variable number of arguments by prefixing the parameter name with an asterisk, `*`. The function will then bind all additional variables to a **tuple**.

**Example 3.20**

Define a function that accepts an arbitrary number of arguments, and multiplies them together.

**Solution:**

```
def multiply_numbers(*args):
    result = 1
    for i in args:
        result *= i
    return result
```

Calling the function written in the above example, we find:

```
>>> multiply_numbers(2, 6, 3, 1)
36
```

But what happens in the above example if we call the function with only one argument, or even no arguments?

```
>>> multiply_numbers()
1
>>> multiply_numbers(2)
2
```

That seems a bit useless! Let's redefine the function, so that it must be given a minimum of two arguments, but still allowing a variable number of arguments:

```
>>> def multiply_numbers(x,y,*args):
...     result = x*y
...     for i in args:
...         result *= i
...     return result
>>> multiply_numbers(2,6,100,10)
12000
```

Now see what happens if we try calling it with zero arguments:

```
>>> multiply_numbers()
TypeError: multiply_numbers() missing 2 required
positional arguments: 'x' and 'y'
```

Perfect!

In addition, Python also provides a mechanism for binding *variable number keyword arguments* to a **dictionary**, using two asterisks, **\*\***

► When defining functions, the parameter order goes:

1. Named arguments
2. Variable arguments
3. Default arguments
4. Variable keyword arguments

### Example 3.21

Define a function that accepts an arbitrary number of arguments, and an arbitrary number of keyword arguments, and prints each keyword and value on a separate line.

**Solution:**

```
def print_args(*args, **kwargs):
    for val in args:
        print('Argument: {}'.format(val))
    for key, val in kwargs.items():
        print('Keyword: {}, Argument: {}'.format(key, val))
```

- ▶ By convention, variable arguments and variable keyword arguments are almost always denoted `*args` and `**kwargs`.

### Argument Unpacking

You can also use the syntax `*args` and `**kwargs` as arguments in function *calls*; when used in this context, they perform a complementary but different procedure to that described above. For instance, consider the following, which makes use of the `print_args()` function defined in the previous example:

```
>>> a = [1.12, 2.54]
>>> b = {'name': 'Josh', 'age': 27}
>>> print_args(*a, **b)
Argument: 1.12
Argument: 2.54
Keyword: name, Argument: Josh.
Keyword: age, Argument: 27.
```

What is happening here?

- `*args` performs **argument unpacking** — it acts on the sequence `args` (i.e. a list, tuple, string, set, or range), passing each element of the sequence as an individual argument.
- `**kwargs` performs **keyword argument unpacking** — it acts on the dictionary `kwargs`, passing each key-value pair as the keyword argument `key=value`.

### 3.8.3 Lambda Functions

Also known as **anonymous functions**, these differ from the standard function definition we've been using above in that they ditch the `def` statement, replace it with a `lambda` statement (hence the name!), and are limited to a single expression. They have the following syntax:

```
>>> my_func = lambda arg1, [arg2, [arg3, ...]]: expression
```

The parameter names are listed to the left of the colon, and the expression to be returned on the right. Since only a single expression is allowed, lambda functions do not have a local variable scope — they can only access the arguments listed.

To see how this works in practice, consider a lambda function that squares its argument:

```
>>> square = lambda x: x**2
>>> square(5)
25
```

Or, a lambda function that adds together the squares of two numbers:

```
>>> add = lambda x,y: x**2 + y**2
>>> add(3, 4)
25
```

Lambda functions are especially useful when short, temporary, nameless functions are needed, for example as arguments to other functions. One such case is when we want to use the `sorted()` function to sort a list of tuples by the *last element* of each tuple:

```
>>> a = [(6,1,3), (1,0), (8,-6), (-3,)]
>>> sorted(a, key=lambda val: val[-1])
[(8, -6), (-3,), (1, 0), (6, 1, 3)]
```

► The `key` keyword argument for `sorted` determines the function to be evaluated on each element to determine the sorting.

## 3.9 NumPy and Arrays

We briefly mentioned in the previous chapter that Python doesn't have an intrinsic function for the square root, but brushed it away as a future problem. Well, the future is now! Let's have a closer look at how to access common mathematical functions in Python.

In Python, very few intrinsic mathematical functions are provided by default; those that are included are `abs(x)` (returns the absolute value or magnitude of an integer, float, or complex number); `round(x,n)` (rounds a number to a given number of digits); and `sum(iter)` (sums the elements of an iterable). In order to access more, we need to import a math module.

One option is the standard Python module `math` — this is a convenient choice, since it is installed alongside Python. However, with the rise of scientific computing with Python, it has become common to use another mathematics module, one that comes with significantly more mathematical functions, that can accept more types of arguments, and a whole new data structure of arrays. This module is NumPy.

### 3.9.1 Mathematical Functions

Before using NumPy, it must first be imported:

```
>>> import numpy as np
```

Here, we import it using the common abbreviation ‘`np`’, allowing us to access NumPy methods and functions with slightly less typing each time. Once imported, we can now access NumPy attributes, functions, and methods; for example, the mathematical constants  $\pi$  and  $e$ :

```
>>> np.pi
3.141592653589793
>>> np.e
2.718281828459045
```

NumPy provides numerous available mathematical functions, including trigonometric functions, exponential functions, logarithms, and functions for working with complex numbers, amongst others. In this section, we will list some of the more important mathematical functions you will need in this course.

The complex analysis functions are especially useful in the context of quantum mechanics, as they allow us to easily convert between polar and Cartesian representations of complex numbers. For example, converting  $1+2i$  into polar form,

```
>>> x = 1+2j
>>> phi = np.angle(x)
>>> r = np.abs(x)
```

- ▶ In fact, the rise of [NumPy](#) and [SciPy](#) have directly contributed to the success of Python in the world of scientific computing.

- ▶ For a full list of all NumPy mathematical functions, the best source is the excellent online [NumPy documentation](#); click on each function in the list to see how it is used.

| Function                                                                               | Description                                                                 |
|----------------------------------------------------------------------------------------|-----------------------------------------------------------------------------|
| <code>np.sqrt(x)</code>                                                                | The square root of $x$                                                      |
| <code>np.cbrt(x)</code>                                                                | The cube root of $x$                                                        |
| <code>np.exp(x)</code>                                                                 | The exponential of $x$                                                      |
| <code>np.log(x)</code><br><code>np.log2(x)</code><br><code>np.log10(x)</code>          | The natural logarithm, base-2 logarithm, and base-10 logarithm respectively |
| <code>np.sin(x)</code><br><code>np.cos(x)</code><br><code>np.tan(x)</code>             | The trigonometric functions                                                 |
| <code>np.arcsin(x)</code><br><code>np.arccos(x)</code><br><code>np.arctan(x)</code>    | The inverse trigonometric functions                                         |
| <code>np.sinh(x)</code><br><code>np.cosh(x)</code><br><code>np.tanh(x)</code>          | The hyperbolic trigonometric functions                                      |
| <code>np.arcsinh(x)</code><br><code>np.arccosh(x)</code><br><code>np.arctanh(x)</code> | The inverse hyperbolic trigonometric functions                              |
| <code>np.deg2rad(x)</code><br><code>np.rad2deg(x)</code>                               | Converting between radians and degrees                                      |

**Table 3.16** NumPy math functions

| Function                                | Description                                                                                                                                                                                                                                                                                           |
|-----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>np.angle(x)</code>                | Returns the phase of complex number $x$                                                                                                                                                                                                                                                               |
| <code>np.real(x)</code>                 | The real part of $x$                                                                                                                                                                                                                                                                                  |
| <code>np.imag(x)</code>                 | The imaginary part of $x$                                                                                                                                                                                                                                                                             |
| <code>np.conj(x)</code>                 | The conjugate of $x$                                                                                                                                                                                                                                                                                  |
| <code>np.abs(x)</code>                  | The absolute value of $x$                                                                                                                                                                                                                                                                             |
| <code>np.real_if_close(x[, tol])</code> | Converts <code>complex</code> $x$ to a <code>float</code> if $\text{Im}(x) < \text{tol} \times \epsilon$ (default <code>tol=100</code> ). $\epsilon$ is the IEEE754 double precision machine epsilon, and on most computers, $\epsilon \approx 2.2 \times 10^{-6}$ . To determine $\epsilon$ exactly: |
|                                         | <code>&gt;&gt;&gt; np.finfo(np.float).eps</code><br><code>2.2204460492503131e-16</code>                                                                                                                                                                                                               |

**Table 3.17** NumPy complex number functions

| Function                         | Description                                                                                                                              |
|----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| <code>np.around(x[, n])</code>   | Rounds the <code>float</code> $x$ to $n$ (default 0) decimal places. If equal or more than halfway, it rounds to the closest even number |
| <code>np.rint(x)</code>          | Rounds to the nearest integer                                                                                                            |
| <code>np.floor(x)</code>         | $\lfloor x \rfloor$ , the floor of $x$                                                                                                   |
| <code>np.ceil(x)</code>          | $\lceil x \rceil$ , the ceiling of $x$                                                                                                   |
| <code>np.trunc(x)</code>         | Discards the fractional part of $x$ , and returns an integer                                                                             |
| <code>np.copysign(x1, x2)</code> | Copies the sign of number $x1$ to $x2$                                                                                                   |

**Table 3.18** NumPy numeric functions

```
>>> print('{:4}exp({:4}i)'.format(phi,r))
1.107exp(2.236i)
```

and then back into Cartesian form:

```
>>> r*np.exp(phi*1j)
(1.0000000000000002+2j)
```

- ▶ We are using **string formatting** to only print 4 significant figures.

### Problem

In the above example, we converted the complex literal  $1 + 2i$  to polar form and back to Cartesian form, and got the result  $1.0000000000000002 + 2i$ . What accounts for this slight difference?

*Hint: think back to the very first chapter of this book!*

## 3.9.2 Arrays

NumPy's rapid rise in popularity in scientific computing isn't just due to these mathematical functions, it has one other important feature up its sleeve — the ability to make use of a new data structure known as **arrays**. While similar in function to built-in Python lists, they differ in several important areas:

- **NumPy arrays can only store one type of data per array.** An array must be either all integers, or all floats, or all strings, etc.
  - As a result of this restriction, NumPy arrays use a lot **less memory** compared to their equivalent Python list.
- **NumPy operations on NumPy arrays are much faster than using lists and loops in Python.** This is because NumPy is written using Fortran, a compiled programming language. Therefore, NumPy allows you to take advantage of the speed of Fortran, from the more modern and expandable environment of Python!
- **NumPy arrays are more akin to mathematical matrices and vectors, as opposed to Python lists.** In fact, they're more similar to arrays in *Mathematica* or *Matlab*. NumPy provides many matrix and vector operations and functions that you can apply to NumPy arrays.

- ▶ This is why NumPy provides their own mathematical functions — all of the functions we introduced above apply equally as well to NumPy arrays!

## Creating Arrays

The `np.array()` function can be used to convert any Python **ordered sequence** (such as a list, tuple, or range) into a NumPy array:

```
>>> x = np.array([4, 5, 7, 1])
>>> print(x)
[4 5 7 1]
>>> print(type(x))
<class 'numpy.ndarray'>
```

- If the ordered sequence contains multiple datatypes as elements, NumPy will automatically cast them to the same datatype. For example,

```
>>> np.array([1, 2, '3'])
['1', '2', '3']
```

- When printing out a NumPy array, it may look identical to a Python list, but using the type function will highlight that this is not the case.

- We can also *return* the dimension/shape of an existing array by using the shape attribute:

```
>>> x = np.zeros((3,2))
>>> x.shape
(3, 2)
```

Note that since shape is an **attribute**, not a method or a function, we don't need to provide round brackets!

NumPy also provides methods unique to arrays. For example, to convert an array back into a Python list, you can use the `np.ndarray.tolist()` method:

```
>>> y = x.tolist()
>>> print(y)
[4 5 7 1]
>>> print(type(y))
<class 'list'>
```

NumPy also has built in functions for defining arbitrary lists of specific dimension:

- `np.zeros(shape, dtype=np.float64)`: create an array of zeros
- `np.ones(shape, dtype=np.float64)`: create an array of ones
- `np.identity(n, dtype=np.float64)`: create an  $n \times n$  identity matrix
- `np.full(c, dtype=None)`: creates an array of constant values  $c$
- `np.arange([start,]end,[step,]dtype=None)`: creates an array in a similar fashion to Python's `range`, but (a) returns a NumPy array, and (b) allows you to construct the range using floats instead of just integers;

```
>>> np.arange(0, 1, 0.2)
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

In all of the above cases, shape refers to a tuple or list of integers that indicates the **shape** or **dimension** of the array to create. For example, to create a  $2 \times 3$  array of ones,

```
>>> np.ones([2, 3])
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

while the keyword argument `dtype` refers to the datatype of elements in the created array; common options include `np.int64` for integers and `np.float64` for double precision floats. Note that when the default value is `dtype=None`, NumPy will automatically determine the datatype based on the other arguments you provide.

### 3.9.3 Array Indexing and Assignment

NumPy indexing is similar to indexing/slicing of Python lists, with one slight difference; we must specify the slice or index *all* array dimensions simultaneously, separated by commas. If we don't wish to slice a particular dimension, we indicate this with `:`, a colon. For those of you familiar with how `List` slicing occurs in the software package *Mathematica*, this is a similar approach but with Python notation.

For example, consider the following NumPy array:

```
>>> a = np.array([[1, 2, 3, 4],
   [5, 6, 7, 8],
   [9, 10, 11, 12],
   [13, 14, 15, 16]])

>>> a[0,:] # returns the first row
array([1, 2, 3, 4])

>>> a[:,1] # returns the second column
array([ 2,  6, 10, 14])

>>> a[:, -2] # returns the second last column
array([ 3,  7, 11, 15])

>>> a[1,2] # return the element in the second row, third column
7
```

- ▶ Indexing does not return a new array, but instead a reference of `view` of the original array — modifying the view will modify the original array as well.
- ▶ Instead of indexing all dimensions at once, you can also apply `successive` indexing. For example, `a[1,2]` is the same as `a[1][2]`.

#### Example 3.22

For the array labelled `a` above, use array slicing to extract, starting from the first row, every second row, with elements from the first to third column.

**Solution:**

```
>>> a[::-2, 1:4]
array([[ 2,  3,  4],
       [10, 11, 12]])
```

Like Python lists, NumPy arrays are mutable and can be changed after creation:

```
>>> a[0,0] = 0 # Set the first row, first column element to zero
>>> a[:, 2] = [0, -1, -2, -3] # Set the third col to [0,-1,-2,-3]
>>> a[-1, :2] = -a[-1, :2] # negate the first two elements
                           of the fourth row
```

However, there are two additional indexing features that NumPy supports, that cannot be used with standard Python lists and tuples.

- **Boolean indexing:** this allows us to index a dimension based on `boolean conditions`, only returning elements that satisfy the condition.

For example, consider the array `a` we assigned above — let's set a boolean condition for all elements of value larger than 3:

```
>>> a > 3
array([[False, False, False, True],
       [True, True, False, True],
       [True, True, False, True],
       [False, False, False, True]], dtype=bool)
```

This creates a NumPy array of the same shape, with `bool` elements indicating if the condition is met. We can then use this as a `mask`, to directly extract the elements that satisfy this condition:

```
>>> a[a > 3] # returns all elements larger than 3
array([ 4,  5,  6,  8,  9, 10, 12, 16])
```

- **Integer array indexing:** this allows the extraction of particular, non-contiguous elements, by providing lists or tuples as the index.

For example, to extract the elements at location (1, 2) and (3, 0) from the above array:

```
>>> a[(1,3), (2,0)]
array([ 7, 13])
```

Note the re-ordering of the indices — indices for the  $n$ th dimension must always go as the  $n$ th argument in the slice, so first we pass the successive rows we are interested in (1 and 3), followed by the successive columns we are interested in (2 and 0).

Additionally, the integer array indices don't need to be one-dimensional! The shape of the index arrays determines the output size. For example, consider the case where we want to extract the corner elements of the array, but keep the two-dimensional shape of the array:

```
>>> rows = [[0, 0], [-1, -1]] # rows of top and bottom corners
>>> cols = [[0, -1], [0, -1]] # cols of top and bottom corners
>>> a[rows, cols]
array([[ 1,  4],
       [13, 16]])
```

### 3.9.4 Array Operations

We mentioned earlier that NumPy arrays are much better suited for representing vectors and matrices than Python lists. Why is that? Recall that, with lists, the `+` operator acts as the concatenation operator, joining lists together:

```
>>> [1, 2, 3, 4] + [5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

Using NumPy arrays, however, mathematical operations act as you would expect from a mathematical viewpoint;

- `+` performs element-wise addition
- `-` performs element-wise subtraction
- `*` performs element-wise multiplication
- `/` performs element-wise division
- `**` performs element-wise exponentiation

and that's not all, in fact, *all* the Python intrinsic mathematical operators (including the relational operators `==`, `!=`, `>`, `<`, `>=`, `<=`, logical operators `and`, `not`, `or`, and even modulo `%` and floor division `//`) act in an element-wise manner with NumPy arrays.

Let's re-consider the example above, but using NumPy arrays rather than lists:

```
>>> np.array([1, 2, 3, 4]) + np.array([5, 6, 7, 8])
array([ 6,  8, 10, 12])
```

We now get behaviour that is more suited for working with mathematical vector and matrices! In this case, though, both of our arrays were the same size. What happens if we perform an operation between a NumPy array and a single numeric scalar? Consider the following:

```
>>> mat = np.array([[1,2,3],
                   [4,5,6],
                   [7,8,9]])

>>> mat+1
array([[ 2,  3,  4],
       [ 5,  6,  7],
       [ 8,  9, 10]])

>>> mat/10
array([[ 0.1,  0.2,  0.3],
       [ 0.4,  0.5,  0.6],
       [ 0.7,  0.8,  0.9]])
```

The answer: when adding, subtracting, multiplying, or dividing a NumPy array by an integer or a float, NumPy **broadcasts** that operation to all elements in the array. This also generalises to every other NumPy mathematical function we introduced. For instance, if we wish to calculate the exponential of each element of a matrix, we can simply use the `np.exp()` function:

► **Broadcasting** is a feature of NumPy in which, during an operation on two arrays with different sizes/dimensions, the smaller array is implicitly made larger to match the size of the larger array, by appending copies of itself to itself.

```
>>> np.exp(mat)
array([[ 2.71828183e+00,  7.38905610e+00,  2.00855369e+01],
       [ 5.45981500e+01,  1.48413159e+02,  4.03428793e+02],
       [ 1.09663316e+03,  2.98095799e+03,  8.10308393e+03]])
```

Note that this is just element-by-element exponentiation, and **not** the matrix exponential!

$$e^A = \sum_{n=0}^{\infty} \frac{1}{n!} A^n \neq \begin{bmatrix} e^{a_{11}} & e^{a_{12}} & \dots \\ e^{a_{21}} & e^{a_{22}} & \dots \\ \vdots & \vdots & \ddots \end{bmatrix} \quad (3.1)$$

### Example 3.23

Verify that `np.exp()` performs element-wise exponentiation.

**Solution:** To show that this is the case, we can compare the above to the result of using Python nested list comprehension to manually exponentiate each element:

```
>>> np.exp(mat) == [[np.exp(j) for j in row] for row in mat]
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]], dtype=bool)
```

### 3.9.5 NumPy Array Functions

To actually implement the Taylor series inspired definition of the matrix exponential in Eq. 3.1, we would need a way to perform matrix multiplication or the dot product. NumPy makes this easy; any variable which is an instance of a NumPy array has the method `np.dot()`, which accepts as an argument the other matrix/vector to dot product with. For example, to multiply two matrices together:

```
>>> A = np.array([[1,2],[4,-2]])
>>> B = np.array([[0.5,7],[0,-1]])
>>> A.dot(B)
array([[ 0.5,   5. ],
       [ 2. ,  30. ]])
```

Alternatively, if we don't want to store A and B as variables, we can call the `np.dot()` method directly:

```
>>> np.dot(np.array([[1,2],[4,-2]]), np.array([[0.5,7],[0,-1]]))
array([[ 0.5,   5. ],
       [ 2. ,  30. ]])
```

#### NumPy matrix multiplication

NumPy uses the same method for matrix-matrix multiplication, matrix-vector dot products,

```
>>> A.dot(np.array([2,1]))
array([4, 6])
```

and vector-vector dot products

```
>>> v1 = np.array([3,4,7,1])
>>> v2 = np.array([2,0,0,1])
>>> v1.dot(v2)
```

7

### The matrix multiplication operator

Starting in Python version 3.5 and above, there is a new way of writing matrix multiplication; using the symbol `@`. If you are running Python 3.5 or a higher version, create two NumPy arrays with variable names `A` and `B`, and try running `A @ B`. You should see the same result as when you call `A.dot(B)`!

If instead you get a syntax error, don't worry - this means you are using an older version of Python 3.

Other useful NumPy array methods, functions, and attributes are provided below (namely, methods Table 3.22, array inquiry functions Table 3.20, array reduction Table 3.21, and array manipulation functions Table 3.22). This is a *selection* of some of the most useful functions; for a full list of all functions, plus additional function arguments not listed here, please see the online NumPy documentation.

| Method/Attribute      | Description                                                                     |
|-----------------------|---------------------------------------------------------------------------------|
| <code>np.T</code>     | Returns the transpose of the array                                              |
| <code>np.dot()</code> | Performs matrix multiplication, matrix-vector products, and vector dot products |

**Table 3.19** NumPy array methods

| Function                              | Description                                                                                                                                                                       |
|---------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>np.argmax(x[,axis=None])</code> | Returns the indices of the maximum values along the given axis. If axis is not specified, the array is flattened first                                                            |
| <code>np.argmin(x[,axis=None])</code> | Returns the indices of the minimum values along the given axis. If axis is not specified, the array is flattened first                                                            |
| <code>np.all(x[,axis=None])</code>    | Returns <code>True</code> if all elements along a given axis are <code>True</code> , otherwise <code>False</code> . If <code>axis</code> is not set, the array is flattened first |
| <code>np.any(x[,axis=None])</code>    | Returns <code>True</code> if any element along a given axis is <code>True</code> , otherwise <code>False</code> . If <code>axis</code> is not set, the array is flattened first   |

**Table 3.20** NumPy array inquiry functions

| Function                                               | Description                                                                                                                                                                                                                                                       |
|--------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>np.sum(x[, axis=None])</code>                    | Sums the elements of array <code>x</code> along a given axis. If <code>axis</code> is not specified, all elements are summed                                                                                                                                      |
| <code>np.prod(x1, x2)</code>                           | Multiplies the elements of array <code>x</code> along a given axis. If <code>axis</code> is not specified, all elements are multiplied                                                                                                                            |
| <code>np.max(x[, axis=None])</code>                    | Returns the maximum element of array <code>x</code> along a given axis. If <code>axis</code> is not specified, the maximum of all elements is returned                                                                                                            |
| <code>np.min(x1, x2)</code>                            | Returns the minimum element of array <code>x</code> along a given axis. If <code>axis</code> is not specified, the minimum of all elements is returned                                                                                                            |
| <code>np.diagonal(x[, offset=0])</code>                | Returns a 1-dimensional array containing the diagonal of square 2-dimensional array <code>x</code> . <code>offset</code> can be used to extract immediate off-diagonals                                                                                           |
| <code>np.trace(a[, offset=0, axis1=0, axis2=1])</code> | Returns the matrix trace along the diagonal of a 2D array. If <code>offset</code> is non-zero, this is calculated along the requested off-diagonal. <code>axis1</code> and <code>axis2</code> are used to return the partial traces for higher dimensional arrays |
| <code>np.outer(x1, x2)</code>                          | Returns the outer product of two arrays <code>x1</code> and <code>x2</code>                                                                                                                                                                                       |

**Table 3.21** NumPy array reduction functions

| Function                                                    | Description                                                                                                                                                                                                                                                                                                                      |
|-------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>np.append(x,<br/>           vals[, axis=None])</code> | Appends the sequence of values <code>val</code> to the array <code>x</code> , along a given axis. If <code>axis</code> is not specified, both <code>x</code> and <code>val</code> are flattened before appending                                                                                                                 |
| <code>np.reshape(x, dim)</code>                             | Reshapes an array <code>x</code> according to the sequence of integers <code>dim</code> , which lists the size of each dimension. Multiplying the integers listed in <code>dim</code> should give the same number of elements as the original array. If <code>dim</code> is an integer, the returned array will be 1-dimensional |
| <code>np.concatenate([x1,x2,...][, axis=0])</code>          | Concatenates together a sequence of arrays along a given axis                                                                                                                                                                                                                                                                    |
| <code>np.vstack([x1,x2,...])</code>                         | Stacks consecutive rows from each array ‘vertically’ as rows in a larger array                                                                                                                                                                                                                                                   |
| <code>np.hstack([x1,x2,...])</code>                         | Concatenates the <i>i</i> th row of each array, to form the <i>i</i> th row of a larger array                                                                                                                                                                                                                                    |
| <code>np.roll(x,<br/>         shift[, axis=None])</code>    | Shift all elements in an array by integer <code>shift</code> along a given axis. If <code>axis</code> is none, the array is flattened, shifted, and re-shaped. If <code>shift</code> is a tuple of integers, each integer corresponds to the shift of each axis                                                                  |
| <code>np.sort(x[, axis=-1])</code>                          | Returns a copy of the array, sorted along the given axis. The default <code>axis=-1</code> sorts along the last axis                                                                                                                                                                                                             |

**Table 3.22** NumPy array manipulation functions

### Array methods versus functions

While we have attempted to split up the tables between methods (those applied directly to array instances, for example `x.T` to calculate the transpose of array `x`) and functions (those called directly from NumPy, which take the arrays it acts upon as arguments, e.g. `np.concatenate([x1, x2])`), some (not all) NumPy routines exist as both methods *and* functions.

For example, to calculate the diagonal of a matrix, you can use either `np.diagonal(x)` or `x.diagonal()`.

What exactly do we mean when we talk about the `axis` argument in the above lists of NumPy array routines? This is a very useful and important argument, so let's delve a little deeper to understand why.

The `axis` argument determines which dimension of the array the function or method applies to. If not provided, the array is generally flattened, and the routine is applied to all the elements. For example, consider the following array, a  $4 \times 4$  array of consecutive integers:

```
>>> x = np.reshape(range(16),(4,4))
>>> print(x)
[[ 0,  1,  2,  3],
 [ 4,  5,  6,  7],
 [ 8,  9, 10, 11],
 [12, 13, 14, 15]]
```

Applying `np.sum()` without specifying an axis will sum *all* elements, regardless of dimension.

```
>>> np.sum(x)
120
```

- ▶ Like with slicing, we can also specify `axis=-n` for an integer `n`, to indicate the `n`th last dimension.

However, if we instead specify `axis=0`, NumPy will take the summation of the first dimension only — this is equivalent to adding each row together, providing the total of each *column*:

```
>>> np.sum(x, axis=0)
array([24, 28, 32, 36])
>>> x[0] + x[1] + x[2] + x[3]
array([24, 28, 32, 36])
```

On the other hand, if `axis=1`, NumPy will take the summation of the second dimension only — this is equivalent to summing each row individually, providing the total of each *row*:

```
>>> np.sum(x, axis=1)
array([ 6, 22, 38, 54])
>>> np.sum(x[0]), np.sum(x[1]), np.sum(x[2]), np.sum(x[3])
(6, 22, 38, 54)
```

This generalises to every routine that accepts `axis` as a keyword argument.

### Example 3.24 Normalisation

Normalise the quantum mechanic state vector  $|\psi\rangle = |0\rangle + (1+2i)|1\rangle - 0.2|2\rangle$  using the relationship  $\langle\psi|\psi\rangle = \sum_i |\psi_i|^2 = 1$ , and then return the probability vector  $|\psi|^2$ .

**Solution:** No, you're not going mad, you have seen this exact problem before! We first attempted to normalise a state vector when we studied data structures such as lists. Let's give this another attempt, this time making use of NumPy.

```
import numpy as np

# create the array
psi = np.array([1, 1+2j, -0.2])

# Note: NumPy will automatically cast all elements
# to type numpy.complex128, as NumPy arrays must
# have elements of the same type.

# normalisation constant
N = 1./np.sqrt(np.sum(np.abs(psi)**2))

# normalise psi
psi = N*psi

# probability vector
P = np.abs(psi)**2
print(P)
```

*Output:* [ 0.16556291 0.82781457 0.00662252]

- ▶ You may sometimes also see the following shortcut to reshape NumPy arrays:

```
np.arange(5)[:, None]
```

This uses **slicing** to instruct NumPy to turn the 1D array into a 2-dimensional array, with all values in the first dimension, and nothing in the second dimension.

It is equivalent to

```
np.arange(5).
```

```
reshape(-1, 1)
```

### Broadcasting: Keeping NumPy fast

If it is possible, *always* try and perform intensive arrays calculations using *only* NumPy functions.

Using **for** loops — or even loop comprehension — on NumPy arrays results in Python converting the array to a list, doing the calculation much more slowly, and then converting it back into a NumPy array. You lose the massive speed advantage you get from using NumPy!

The trick to getting the most out of NumPy is through broadcasting, like we covered above. And the secret to getting the most out of broadcasting? The **np.reshape()** function/method.

**np.reshape()** was briefly introduced in Table 3.22 as a means of reshaping a NumPy array from one set of dimensions to another. What makes it so useful is that you can also indicate a reshaped dimension size of -1; indicating that NumPy should place all left over elements in this dimension. Don't worry if this doesn't make sense straight away, the best way to get your head around broadcasting is to try it yourself.

Let's have a look at a quick example. Consider a  $5 \times 5$  matrix defined by  $W_{mn} = e^{2\pi i nm/N}$ ,  $0 \leq n \leq 4$  and  $0 \leq m \leq 4$ . We can think of this as a 2-dimensional array, indexed by  $m$  in the first dimension, and  $n$  in the second. Using **np.reshape**, we can create our two indices  $n$  and  $m$ , with their values in the correct dimension:

```
>>> # m is rank-2 with all values placed in the 1st dimension
>>> m = np.arange(0, 5).reshape(-1, 1)
>>> # n is rank-2 with all values placed in the 2nd dimension
>>> n = np.arange(0, 5).reshape(1, -1)
```

Note that we are using the method form of **reshape** here. Have a go printing out **m** and **n** — for **m**, the indices 0–4 have been placed in the column of the  $5 \times 1$  array, whereas for **n**, they are in the row of a  $1 \times 5$  array.

Now, let's use these two arrays to construct  $W_{mn}$ :

```
>>> W = np.exp(2j*np.pi*m*n/5)
```

And that's it! No Python loops or list comprehensions needed — Python broadcasting automatically takes care of it. Let's double check the shape of the new array, check that the  $m = 2, n = 3$  element is correct:

```
>>> print(W.shape)
[5, 5]
>>> W[2,3] == np.exp(2j*np.pi*2*3/5)
True
```

As long as we restrict ourselves to NumPy functions and methods that understand how to automatically broadcast dimensions (such as `np.exp` above), broadcasting and the `reshape` function are very powerful. Combined with the ability to specify the axis/dimension on which NumPy operations act, almost any standard list comprehension/Python loop can be replaced by NumPy broadcasting.

## Input and Output

We mentioned on the previous page that NumPy arrays and objects should *always* be kept in memory as NumPy arrays, in order to benefit from the huge speedup and memory boost NumPy provides. Any time you use a Python intrinsic function, or a Pythonic method of iteration (such as a for loop), it will be converted away from its Fortran-based representation into a Python list. This is a Very Important Concept™, hence why we repeat again!

'Well', you might be saying, 'does that cover using the Python read and write commands we covered earlier? What if I need to read and write a NumPy array to a file?!" Don't panic just yet! NumPy has you covered. NumPy provides the following functions for reading and writing arrays to files:

- In addition, NumPy also provides the functions `np.load()` and `np.save()`, for saving and loading arrays as binaries. While these files can only be opened and read by NumPy itself, they are much more storage efficient.

- `numpy.savetxt(fname, x, fmt='%.18e', delimiter=' ')`  
Saves the array `x` to a text file with name `fname` (str). Optional arguments include:
  - `fmt`: determines the formatting of numbers in the text file, and has the format '`%[sign]width[.precision]type`', and uses the exact same arguments as in Sect. 3.4.4. It can be passed as a list of format strings, one for each column, or as a single string applied to all columns
  - `delimiter`: the string or character that separates columns
- `numpy.loadtxt(fname, dtype=np.float64, skiprows=0, delimiter=None, comments='#', usecols=None, unpack=False)`  
Loads an array from text file with name `fname` (str). Optional arguments include:
  - `dtype`: the data type of elements in the loaded array
  - `skiprows`: the number of rows to be skipped from the top of the file
  - `delimiter`: the string or character that separates columns. By default, any type of whitespace acts as the column delimiter
  - `comments`: any row beginning with the specified comment character will be skipped
  - `usecols`: a tuple indicating the column numbers to extract. For example, `usecols=(0,1,5)` will extract the first, second, and sixth columns.
  - `unpack`: if set to `False`, a single 2-dimensional array of rows and columns is returned. If set to `True`, a tuple containing each column as a one-dimensional array is returned, allowing for tuple unpacking:

```
>>> x, y, z = np.loadtxt('data.txt', unpack=True)
```

### SciPy, Linear Algebra, and Other Features

Hold up. Aren't we missing some common matrix operations? What if we want to calculate the inverse of a matrix, the norm of a matrix, the eigenvalues, or even the determinant?

It turns out that computationally, these linear algebra operations are quite a bit harder to do than the operations we've already seen. The sister package to NumPy, called **SciPy**, provides all of these routines (and more!) in a submodule called `scipy.linalg`. In addition, SciPy also currently provides support for special functions, integration, and ordinary differential equation (ODE) solvers — and like NumPy, most of these routines are written in Fortran for the speed boost this provides.

For now, we won't worry about linear algebra numerical techniques; we'll cover them in a later chapter.

### Single versus double precision in NumPy

At the very beginning of this chapter, we noted that Python 3 defaults to **IEEE754 double precision** representation for floating-point real numbers — this equates to allowing 53 bits of precision, or approximately 15 or 16 significant digits in base-10, for those of you frantically flicking back!

NumPy comes with a huge variety of built-in data types, including:

- **Boolean literals:** `np.bool_`
- **Integers:** `np.int8`, `np.int16`, `np.int32`, `np.int64`, referring to the *number of bits* used to store the (signed) integer; for instance `np.int8` uses 8 bits to represent integers  $-128$  to  $127$ .
- **Floating point numbers:** `np.float16` (half precision), `np.float32` (single precision), `np.float64` (double precision).
- **Complex numbers:** `np.complex64` (represented by two single precision floats), `np.complex128` (two double precision floats).

When generating an array with NumPy, it will attempt to intelligently cast to the correct data type; for instance, since Python 3 represents by default floating point numbers in double precision, the following array will be stored using type `np.float64`:

```
>>> x = np.array([1.5, 2.0, 1])
```

This can be checked using the array `np.array.dtype` attribute:

```
>>> x.dtype  
np.float64
```

You can cast a NumPy array to a specific data type by using the respective NumPy type function; for example,

```
>>> np.float32(x)
```

converts `x` to an array of single-precision floats. Alternatively, you can set the data type directly when creating the array, via the `dtype` keyword argument:

---

```
>>> x = np.array([1.5, 2.0, 1], dtype=np.float32)
```

---

## 3.10 Command-Line Arguments

A great time saver when running numerical simulations is the ability to run the same program numerous times, however executed with differing options in the form of **command line arguments**. This is a great way to easily change the behaviour/input of a program without needing to edit it each time, and is supported by a module provided alongside Python called `argparse`. `argparse` allows you to define the types of arguments and flags your program requires, checking whether the correct types of arguments are provided, while also automatically generating help messages.

Before we begin, just what is a flag? Flags are commonly used on Unix systems to pass command line arguments, without having to worry about the *order* of the arguments. Flags fall into two rough categories:

- **Boolean flags:** These do not accept any options, the mere presence of the flag itself changes the behaviour of the program.
  - For example, you have probably come across the flag `--help` or `-h` – these are commonly used to instruct the program to output a brief help message and exit.
  - These can also be used to pass the value of a variable with only two possible options (i.e. `True` or `False`)
- **Value flags:** these require one or more additional *values* to be passed, separated from the flag by a space or an equals sign.
  - For example, `-i data.txt`, `--input data.txt`, or `--input=data.txt`. These are commonly used to pass data such as numbers, strings, or lists to the program.
  - The value passed is referred to as a *flag argument*.

While `argparse` is a large library with a huge number of options and functions, we will focus on just three needed to implement a useful argument parser in Python; namely `ArgumentParser`, `add_argument`, and `parse_args`.

- `parser = ArgumentParser(prog=None, usage=None, description=None)`  
Initialises the argument parser. Here, `prog`, `usage`, and `description` are all optional keyword arguments accepting strings describing the program name, usage, and description.

```
parser.add_argument(flags[], action, default, nargs,
• type, choices, required, help])
```
- `parser.add_argument(flags[], action, default, nargs,`  
`• type, choices, required, help])`  
Adds a command line flag to the initialised parser, with specified flag names. Note that you can pass multiple flag names, both long (`-`) and short (`-`); e.g. `add_argument('--input', '-i', ...)`.  
Optional arguments include:

- `action='store'`: If `action='store'`, the flag is a value flag, and Python will store subsequent flag arguments. If `action='store_true'` or `action='store_false'`, the flag is a boolean flag, and no flag arguments are expected; if the flag is present, `True/False` is stored respectively.
  - `default=None`: The default value of the flag if no flag arguments are provided
  - `nargs='?'`: Determines how many flag arguments should be parsed with this flag. `nargs=2`, for example, will store the next two arguments as a list of length 2. The special cases `nargs='?'` stores a single, optional, argument, while `nargs='*'` will gather all subsequent arguments into a list.
  - `type=str`: The Python type to convert the flag argument into
  - `choices=None`: A list of allowed flag arguments
  - `required=False`: If set to `True`, the flag is required
  - `help=None`: A description of the flag
- `args = parser.parse_args()`  
Parses the command line flags and flag arguments. The flag arguments are stored as attributes `args.flag`, where `flag` is the name of the long flag (without the preceding `--`), or, if only a short flag is provided, the name of the short flag (dropping the `-`).  
In the above example, this would be `args.input`.

### Example 3.25 Command line arguments

Let us consider the case of three arguments:

- an input filename (denoted by the flag `--input` or `-i`)
- an integer (denoted by the flag `-N`)
- a boolean (denoted by the flag `-b`)

Implement a Python program that accepts these two flags, i.e.

```
python3 argtest.py --input data.txt -N 10 -b
```

with the following properties:

- (a) an input file **must** be provided
- (b) the integer flag is optional; if not provided, it defaults to a value of 1
- (c) the boolean flag is optional; if not provided, it defaults to `False`
- (d) set the variables `filename`, `N`, and `b`

Solution:

```
#!/usr/bin/env python3
import argparse

# first, we initialise the parser
name = 'ArgParse Example'
desc = 'An example Python program with command line arguments'
parser = argparse.ArgumentParser(prog=name, description=desc)

# add the input flag
parser.add_argument('--input', '-i', required=True)
# add the integer flag
parser.add_argument('-N', type=int, default=1)
# add the input flag
parser.add_argument('-b', action='store_true')

# parse the command line arguments
args = parser.parse_args()

# store the arguments as variables
filename = args.input
N = args.N
b = args.b

print(filename, N, b)
```

Try running the above program with a variety of differing command line arguments. What happens if `--input` or `-i` is omitted as a flag? What do you see if you supply the `-h` or `--help` flag?

## 3.11 Timing Your Code

In most general programming cases, code execution time is typically neglected – at least, it has been for all examples and problems posed so far in this text! But in the field of computational physics, where simulations can become increasing computationally-intensive (for example, solving Schrödinger’s equation in multiple dimensions), the difficulty in ensuring your code converges to a solution in a timely manner can sometimes approach (if not overtake!) the difficulty of numerically solving your problem in the first place.

In fact, knowing the total elapsed time it takes for your program to complete, from start to finish – referred to as the **wall-clock** or **wall** time – can be quite useful (for example, in estimating the wall-time required when submitting jobs to a scheduler on a cluster). Additionally, determining the wall time for specific functions or control structures (e.g. for loops) can help identify the most computationally-intensive portion of your program, and indicate which sections should be optimised if possible.

Python provides two ways to time code execution; either through the built-in **time** module, or through the built-in **timeit** module. We will discuss both methods.

### 3.11.1 CPU Time

CPU time differs from wall time in that it measures the time spent by your process on the CPU. So for instance, whilst wall time gives you the total elapsed time (including the time spent by your process *waiting* for resources such as the processor, network, and/or IO devices), the CPU time simply gives you the time spent executing your code on the CPU – thus, in general for non-parallel programs, CPU time < wall time.

To measure CPU time in Python, we use the **time.process\_time()** function like so:

```
import time
start = time.process_time()
# code to time
end = time.process_time()
cpu_time = end - start
```

Note that the absolute value of **time.process\_time()** does not mean anything; rather, it is the *relative* CPU timing (or *difference* between two CPU times) that give us the CPU time of code section we are interested in.

### 3.11.2 Wall Time

Whilst measuring CPU time is sufficient for simple sequential programs, in more advanced cases it is better to measure the wall time. For example, when working on a multi-tasking machine, the CPU and wall time could differ

significantly as different processes compete for resources. Furthermore, the resulting CPU time for a parallel program will sum together the individual CPU times of each thread/process – leading to the result CPU time > wall time!

To measure wall time in Fortran, we use the `time.perf_counter()` function as follows:

```
import time
start = time.perf_counter()
# code to time
end = time.perf_counter()
wall_time = end - start
```

Note that, like with `time.process_time()`, it is the *difference* between the count rates that gives us the elapsed count. However, we need to do an additional calculation to convert the count difference into seconds, by dividing by the count rate.

### 3.11.3 Timeit

However, one of the best ways to measure time in Python is through the (also built-in) `timeit` module. This module takes into consideration common deadfalls when measuring program execution time, determines the best `time` function to use to calculate execution time, and automatically times over several loops to determine the absolute minimum execution time. Compared to the `time` module, though, `timeit` is designed to be run from the command line, and times **your program as a whole**.

For example, say we have a Python program `example.py`, and we wish to use `timeit` to test the execution time. To do so, we simply execute the following on the command line:

```
$ python3 -m timeit example.py
10000 loops, best of 3: 67.1 usec per loop
```

What is happening here? Essentially, `timeit` has automatically decided to loop the program 10000 times; the total time of execution is then given by  $\text{time}/1000$ . This is repeated three times, and the best (lowest!) result of the three is returned.

`timeit` determines the number of loops and the number of times to repeat the timer automatically, but these can also be specified via command line arguments:

- `-n N, --number=N`: the number of times to loop the execution while timing
- `-r N, --repeat=N`: the number of times to repeat the timer (default 3)

► You might see some resources instructing you to use the functions `time.clock()` for CPU time, and `time.time()` for wall time.

Due to inconsistencies with how they worked on different operating systems, they have been **deprecated** since Python version 3.3.

`time.process_time()` and `time.perf_counter()` are their replacements respectively.

- `-p, --process`: measure the CPU time using `time.process_time()`, instead of measuring the wall time using `time.perf_counter()` (the default)

### Further reading

The internet has become an incredible resource to programmers, allowing access to countless pages of documentation tutorials within seconds. As Python is a relatively modern programming language, available online resources are quite high in quality, and almost rival the traditional textbooks. Here, I'll list some useful online resources for Python and scientific computation.

- As always, the online Python (<https://docs.python.org/3/index.html>) and NumPy/SciPy documentation (<https://docs.scipy.org/doc/>) are a good high level resource for looking up available functions, methods, arguments, and syntax.
- An invaluable resource is [Stack Overflow](#), an online Q&A forum for programmers; you can pose your own questions, or search the site to find similar posts.
- The website of Hans Petter Langtangen (<http://hplgit.github.io/>) provides various online books that cover, in huge detail, numerical techniques using Python. In addition, he also covers methods of optimising and speeding up computations by combining Python with compiled languages such as Cython and Fortran.

For additional Python texts, focused on general scientific computation with some additional discussion on other aspects of Python development, we recommend the following:

- Scopatz, A., & Huff, K. D. (2015). Effective computation in physics (First Edition). Sebastopol, CA: O'Reilly Media.
- Hill, C. (2015). Learning scientific programming with Python. Cambridge, United Kingdom: Cambridge University Press.

While Fortran use is still heavily focused on high performance computation, Python is currently carving out a niche as the language of choice for machine learning, with popular libraries including TensorFlow, Scikit-learn, and PyTorch (the reasons for this are manifold, but arguably a side-effect of the strong presence of numerical libraries in Python such as NumPy and SciPy). Machine learning is already becoming a useful tool in scientific computing, from simple data analysis to exploring intractable problems, adding another useful arsenal to the physicist's toolbox.

Simultaneously, as technology advances, we are beginning to explore methods of calculating classically intractable problems using quantum mechanics — a field known as **quantum computing**. Various quantum algorithms already exist; the most well known being Shor's algorithm, which allows the factoring of arbitrary integers efficiently on a quantum computer. The next goal is to build a functional quantum computer, and implement these quantum algorithms experimentally. Various companies and research institutes are currently working towards this goal, with Python generally chosen as the language of choice for interfacing with the prototype quantum computer. Examples include QISKit by IBM, PyQuil by Rigetti, and Strawberry Fields by Xanadu.

## Exercises

**P3.1** Write a program which reads a value  $x$  from keyboard, and calculates and prints the corresponding value  $\sin(x)/(1+x)$ . The case  $x = -1$  should produce an error message, and be followed by an attempt to read a new value of  $x$ .

**P3.2** One important statistic used in measuring the reliability of a circuit component is the mean time to failure. An engineer has tested many samples, and recorded the time at which each sample failed in the file `failuretime.dat`. Write a program to process the data and determine the mean time to failure.

**P3.3** Write a program to evaluate the exponential function using a Taylor series:

$$e^x = 1 + x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \dots$$

Output the result for  $x = -5.5$ .

How quickly does your code converge (i.e. how many terms do you need in the sum in order to achieve an accuracy of 1% or better)?

Can you revise your code to reduce the number of terms required using the above equation to achieve the same accuracy? If so, explain why your change has the effect it does on convergence.

**P3.4** The  $N \times N$  discrete Fourier transform (DFT) matrix is defined by

$$W = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

where  $\omega = e^{-2\pi i/N}$ .

Use a NumPy array to create a  $4 \times 4$  DFT matrix.

1. Apply what you know about formatting to output a neatly formatted matrix on screen.
2. Use `numpy.savetxt()` to save the resulting array to a text file, with commas separating each value.

**P3.5** Write a function or subroutine that accepts integer  $N$  as input and returns an  $N \times N$  DFT matrix

Part II:

Numerical Methods for Quantum  
Physics



# Chapter 4

## Finding Roots

Root-finding or equation-solving algorithms are an essential part of computational physics – they are used every time we need to solve for an unknown quantity appearing in an explicit or implicit equation.

In this chapter, we'll investigate some useful root finding algorithms. Before getting started, however, let's go over a few terms and conventions of numerical methods that we will encounter repeatedly in this chapter.

### 4.1 Big-O Notation

Also known as Landau notation, this is a very convenient method for denoting how particular approximations or algorithms behave in the neighbourhood of some asymptotic limit. For example, let's consider a Taylor series expansion of the exponential function  $f(x) = e^x$ ; we can write this as

$$e^{x-a} = \sum_{n=0}^{\infty} \frac{1}{n!} f^n(a)(x-a)^n = \sum_{n=0}^{\infty} \frac{1}{n!} e^a (x-a)^n. \quad (4.1)$$

Expanding around the point  $a = 0$ ,

$$e^x = \sum_{n=0}^{\infty} \frac{1}{n!} x^n = 1 + x + \frac{1}{2} x^2 + \frac{1}{6} x^3 + \frac{1}{24} x^4 + \dots \quad (4.2)$$

If we wish to truncate the infinite series to a particular order in  $x$ , for example  $x^3$ , then we expect the first truncated term to dominate the error as  $x \rightarrow a$ . We can use Big-O notation to denote the dominating error term:

$$e^x = 1 + x + \frac{1}{2} x^2 + \mathcal{O}(x^3). \quad (4.3)$$

What this tells us is that as  $x$  tends towards zero, the absolute error in our Taylor series approximation is at most  $c|x^3|$ , where  $c$  is some constant value. If we were to write this more generally in the form

$$f(x) = g(x) + \mathcal{O}(h(x)), \quad (4.4)$$

► When writing Big-O notation, we drop all constants, and keep only the term that dominates and provides an upper bound on the absolute error — in this case,  $x^3$ .

If truncating *directly*, without Big-O notation, we can use Taylor's theorem to determine the full error term directly:

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} (x-a)^{n+1}$$

where  $\xi \in [a, x]$ .

where  $g(x)$  is an *approximation* of  $f(x)$  in the limit  $x \rightarrow a$ , then the Big-O notation indicates that

$$\exists k > 0 \text{ and } c > 0 \text{ such that } \forall |x - a| < k, |f(x) - g(x)| \leq c|h(x)|. \quad (4.5)$$

Here,  $\epsilon = |f(x) - g(x)|$  is the **absolute error** of the approximation, and if  $h(x) = x^n$  for some  $n$ , then  $n - 1$  is often said to be the **order** of the approximation.

### Relative error

Aside from absolute error, another convention used in analysing the error of numerical approximations is relative error, defined as

$$\left| \frac{f(x) - g(x)}{f(x)} \right|, \quad (4.6)$$

where  $f(x)$  is the exact result of some computation for input  $x$ , and  $g(x)$  is an approximate result using a specified numerical technique.

Note that a slightly different definition of Big-O notation exists, that is primarily used in computer science to denote how efficient a particular algorithm is. In this case, if  $f(N)$  is an algorithm that accepts an input of size  $N$ , then we can say

$$f(N) \sim \mathcal{O}(h(N)), \quad (4.7)$$

that is,  $f(N)$  grows or scales to the order of  $h(N)$  as  $n \rightarrow \infty$ . More formally,

$$\exists n > 0 \text{ and } c > 0 \text{ such that } \forall N > n, |f(N)| \leq c|h(N)|, \quad (4.8)$$

i.e.  $|h(N)|$  provides an upper bound on the behaviour of  $f(N)$  as  $N \rightarrow \infty$ . For example, if we have the following nested loop pseudo-code,

```
for i in 1,2,...,N
    for j in 1,2,...,N
        f(i,j)
```

then we say that the computational complexity is  $\mathcal{O}(N^2)$  — we have a total of  $N^2$  loops executing.

We'll mostly be seeing the first definition in future sections (the **error scaling**), however in some cases we'll also come across the second definition in terms of how particular algorithms scale (**computational complexity**). In either case, it will be obvious from the context which Big-O definition we are using.

- ▶ For example,  $N$  can be the size of a matrix input required by the algorithm, or the number of loops required by the algorithm.

- ▶ Since  $\mathcal{O}(h(N))$  is an **upper bound**, sometimes it may be possible to find a  $g(N)$  such that

$$|f(N)| \leq c_1|g(N)| \leq c_0|h(N)|$$

at which case we now simply say that  $f(N) \sim g(N)$ .

## 4.2 Convergence

Another important concept that must be considered when applying numerical methods is **convergence**; whereas error analysis is concerned with the deviation from the exact result for different initial values, convergence instead considers whether the numerical method will even approach a fixed value in the neighbourhood of the correct solution. And, if it does, can we find a numerical method that converges faster?

In general, we will be presenting methods and techniques here that are known to converge; that is, if the numerical method produces a value  $x_n$  for each iteration  $n = 0, 1, 2, \dots$ , then

$$\lim_{n \rightarrow \infty} x_n = L, \quad (4.9)$$

and we say that the method has converged to the **fixed-point**  $L$ . To determine how *quickly* this convergence takes place, we can consider two successive ‘differences’ from the converged value in the iterative process, as the iteration step tends towards infinity:

$$|x_{n+1} - L| = C|x_n - L|^p, \quad C > 0, \quad n \rightarrow \infty. \quad (4.10)$$

Here,  $C$  is some positive constant. The value of  $p$  determines how quickly the convergence occurs, and is referred to as the **order of convergence**; for example,  $p = 1$  denotes linear convergence,  $p = 2$  quadratic convergence, and so on.

We can get some additional insight into this process if we consider that  $x_{n+1} = f(x_n)$  is the function providing the iterative process, and simply depends on the current value  $x_n$ . Letting  $\epsilon_n = x_n - L$  denote the error at step  $n$  of the iteration, then we can write

$$x_{n+1} = f(x_n) = f(L + \epsilon_n) = f(L) + \epsilon_n f'(L) + \frac{1}{2} \epsilon_n^2 f''(L) + \dots \quad (4.11)$$

Since, by definition,  $x_{n+1} = L + \epsilon_{n+1}$  and  $f(L) = L$  ( $L$  is a fixed-point of the iterative method), we have

$$\epsilon_{n+1} = \epsilon_n f'(L) + \frac{1}{2} \epsilon_n^2 f''(L) + \dots \quad (4.12)$$

Taking the absolute value and the limit  $n \rightarrow \infty$ , we can neglect all higher order terms in the Taylor series beyond the linear term, which will dominate, leading to

$$|x_{n+1} - L| = |f'(L)| |x_n - L|, \quad n \rightarrow \infty. \quad (4.13)$$

It can be seen that, in the case of linear convergence, the constant value  $|f'(L)|$  determines the exact nature of the convergence;  $0 < |f'(L)| < 1$  leads

to linear convergence with rate of convergence  $|f'(L)|$ , and  $|f'(L)| = 1$  gives **sublinear convergence**.

More interestingly, what happens if  $f'(L) = 0$ ? In this case, the convergence is superlinear, and we must move onto the next term in the Taylor series for the dominant error;

$$|x_{n+1} - L| = \frac{1}{2}|f''(L)||x_n - L|^2, \quad n \rightarrow \infty, \quad (4.14)$$

- However, we do know that for the interval of interest  $I$ ,

$$C = \sup_{x \in I} \frac{1}{p!} |f^{(p)}(x)|$$

that is,  $C$  is the **least upper bound or supremum** of values  $|f^{(p)}(x)|/p!$  within that interval.

i.e. we get quadratic convergence. In general, if  $f^{(m)}(L) = 0$  for all  $m < p$ , then we have  $p$ -order convergence. However, there is one slight subtlety here that we have been avoiding — in practical computation, if we don't know  $L$  beforehand, then we *can't* run an iterative procedure an infinite number of times to determine  $L$ ! The best we can say is that

$$\exists C > 0, \quad p > 0, \quad \text{such that } |x_{n+1} - L| \leq C|x_n - L|^p. \quad (4.15)$$

### 4.3 Bisection Method

The bisection method is one of the simplest root-finding algorithms, and relies on ‘zeroing’ in on a root by repeatedly bisecting an interval and using the intermediate value theorem.

#### Intermediate Value Theorem

Consider an interval in the real numbers  $D = [a, b]$ , and a continuous function  $f : D \rightarrow \mathbb{R}$ . If there exists some  $m \in \mathbb{R}$  such that  $f(a) < m < f(b)$ , then there exists a value  $c \in [a, b]$  such that  $f(c) = m$ .

**Corollary:** if  $f(a)$  and  $f(b)$  are of opposite signs, there exists at least one root  $c \in [a, b]$  such that  $f(c) = 0$

The bisection algorithm can therefore be summarised as follows:

#### Bisection Method Algorithm

Choose an appropriate interval  $x \in [a_0, b_0]$ , such that  $f(a_0)f(b_0) < 0$  (that is they have opposite signs).

Then, for  $n = 0, 1, 2, \dots$

1. Calculate the midpoint of the interval  $c_n = \frac{1}{2}(a_n + b_n)$
2. Check if it has converged to a root within acceptable precision; if so, **exit** the algorithm here
3. If there is no convergence, *bisect* the interval:
  - i. if  $f(a_n)f(c_n) > 0$  (they have the same sign), the new interval is

$$[a_{n+1}, b_{n+1}] = [c_n, b_n].$$

ii. if  $f(a_n)f(c_n) < 0$  (they have opposite signs), the new interval is

$$[a_{n+1}, b_{n+1}] = [a_n, c_n].$$

4. Return to step 1.

### Convergence

How do we know when to stop the bisection method? In other words, what criteria should we choose that indicates convergence has been achieved? There are two possible convergence criteria; either of which, when satisfied, indicate that a root has been found to within suitable tolerance:

1. If the value of the function calculated at the midpoint is within a certain specified **tolerance**, denoted  $\epsilon$ , then a root has been found. That is,

$$\text{if } |f(c_n)| \leq \epsilon \text{ for some } \epsilon \ll 1 \Rightarrow \text{the root is } x \approx c_n.$$

2. If the interval length has reduced to within a certain specified tolerance  $\epsilon$ , then a root has been found; i.e.

$$\text{if } b_n - a_n \leq \epsilon \text{ for some } \epsilon \ll 1 \Rightarrow \text{the root is } x \approx c_n = \frac{1}{2}(a_n + b_n).$$

However, in practice it is better to restrict ourselves to convergence criterion number 2, as it is possible to encounter functions where  $|f(c_n)| \leq \epsilon$  even if  **$c_n$  is not close to the root**.

Note that with every bisection, the interval size *halves*, as we are replacing one of the boundaries with the midpoint. So, after  $n$  iterations of the algorithm, the interval size is

$$b_n - a_n = \frac{1}{2^n}(b_0 - a_0). \quad (4.16)$$

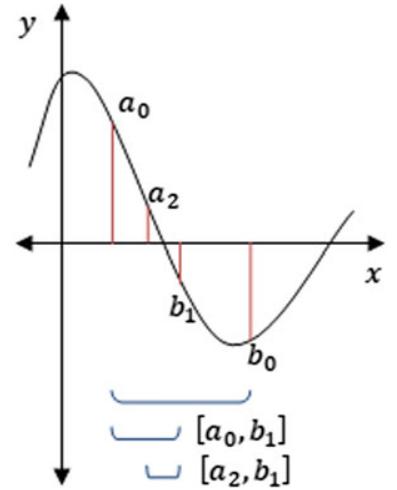
Thus it is intuitive to see that the bisection method *always* converges to the root. However, this convergence is comparatively slow – with every iteration, the **absolute error** between the actual root  $c$  and estimated root  $c_n$  halves, as the absolute error is bound by the size of the current interval:

$$|c_n - c| \leq \frac{1}{2^n}(b_0 - a_0). \quad (4.17)$$

As the rate of convergence is constant, the bisection method has a linear convergence.

Alternatively, we can use the convergence criterion  $b_n - a_n < \epsilon$  to solve for the number of iterations needed for convergence:

$$\begin{aligned} b_n - a_n \leq \epsilon &\Rightarrow \frac{1}{2^n}(b_0 - a_0) < \epsilon \\ &\Rightarrow 2^n \geq \epsilon^{-1}(b_0 - a_0). \end{aligned}$$



**Figure 4.1** Diagram illustrating the bisection method. Note that the interval size decreases by half with each iteration.

Taking the log of both sides,

$$n \geq \frac{\log(b_0 - a_0) - \log(\epsilon)}{\log 2}. \quad (4.18)$$

Therefore the number of iterations required depends only on the initial size of the interval  $b_0 - a_0$  and the chosen tolerance  $\epsilon$ .

## 4.4 Newton-Raphson Method

The Newton-Raphson method improves on the bisection method by using the gradient of the curve to ‘zero’ in on the root, rather than bisecting an interval. For example, consider the equation  $y = f(x)$ . The tangent line at a point  $f(x_n)$  is given by

$$y = f(x_n) + f'(x_n)(x - x_n). \quad (4.19)$$

We can now use the *root* of this tangent line (a first order approximation to the function  $f(x)$  at point  $x_0$ ) to calculate our next guess  $x_{n+1}$  for the root of the function:

$$\Rightarrow 0 = f(x_n) + f'(x_n)(x_{n+1} - x_n) \quad (4.20)$$

$$\Rightarrow x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \quad (4.21)$$

### Convergence

Similar to the bisection method, the most effective convergence criterion is

$$\frac{|x_{n+1} - x_n|}{|x_n|} \leq \epsilon.$$

Whilst Newton’s method has a faster rate of convergence compared to the bisection method (in fact, it has a quadratic rate of convergence), it is not always guaranteed to converge; this could be due to the following reasons:

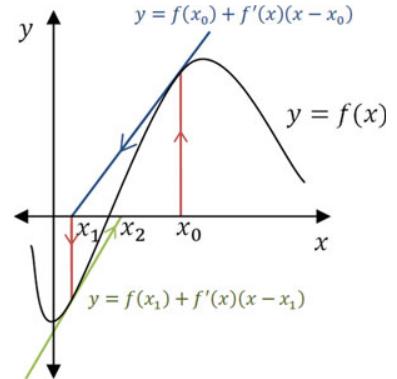
- The initial point  $x_0$  is not within a neighbourhood of the root where convergence occurs – in this case, another method (such as the bisection method) can be used to initially narrow down a region near the root, before applying Newton-Raphson.
- If any iteration point is a stationary point, then  $f'(x_n) = 0$  and the iterative formula is not defined – the method *terminates* before finding the root.
- Alternatively, a  $k$ -cycle may be encountered, where the iteration cycles between the same  $k$  points, never converging to the root.

To see why this is the case, let’s look into the convergence of the Newton-Raphson method slightly more rigorously.

### Newton-Raphson Convergence

Let  $L$  be an actual root of  $f(x)$ ; expanding the function  $f(L)$  around the point  $x = x_n$ , we have

$$f(L) = f(x_n) + f'(x_n)(L - x_n) + \frac{1}{2}f''(\xi)(L - x_n)^2 = 0 \quad (4.22)$$



**Figure 4.2** Diagram illustrating two iterations of the Newton-Raphson method.

► For an example of using the Newton-Raphson method, see Example 5.4.

assuming that  $x_n$  lies in the neighborhood of  $x = L$ ; that is,  $|L - x_n| \ll 1$ . Here,  $\frac{1}{2}f''(\xi)(L - x_n)^2$  is the error term, with  $\xi \in [x_n, L]$ . Rearranging, and dividing by  $f'(x_n)$ ,

$$L - \left( x_n + \frac{f(x_n)}{f'(x_n)} \right) = -\frac{f''(\xi)}{2f'(x_n)}(L - x_n)^2. \quad (4.23)$$

The bracketed term is simply the Newton-Raphson method, producing the next iteration  $x_{n+1}$ . Substituting in  $x_{n+1}$  for the bracketed term, and taking the absolute value of both sides,

$$|L - x_{n+1}| = \frac{|f''(\xi)|}{2|f'(x_n)|} |L - x_n|^2. \quad (4.24)$$

Thus, we can see that the Newton-Raphson method has a quadratic rate of convergence, when  $|L - x_n| \ll 1$ ,  $f'(x) \neq 0$ , and  $f''(x)$  is continuous for all  $x \in [L - x_0, L + x_0]$ .

Therefore an **additional ‘stopping’ criterion** should be  $n > N$  for some specified  $N$ , to ensure our code does not run forever, where  $N$  is the number of iterations performed.

Note that each iteration of the Newton-Raphson method requires knowledge of  $f'(x)$ . If you know  $f(x)$ , the derivative can be hard-coded straight into your algorithm. However, if  $f(x)$  is unknown prior to running the code, the derivative will need to be calculated numerically – this leads to the **secant method**.

## 4.5 Secant Method

Let's apply the backwards finite-difference formula,

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}, \quad (4.25)$$

to the Newton-Raphson iterative equation:

$$x_{n+1} = x_n - f(x_n) \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}. \quad (4.26)$$

### Convergence

The most effective convergence criterion remains the same as for the Newton-Raphson case; namely

$$\frac{|x_{n+1} - x_n|}{|x_n|} \leq \epsilon \quad \text{and} \quad n \geq N.$$

However now *two* initial conditions are required, i.e.  $x_0$  and  $x_1$ . As with the Newton-Raphson method, these should be chosen in the neighborhood of the root to increase the likelihood and rapidity of convergence.

But is the rate of convergence the same as the Newton-Raphson method?<sup>1</sup> Alas, it is not — and we have the finite difference approximation to blame for that. In general, the Newton-Raphson method **always converges faster** than the secant method.

It turns out the rate of convergence of the secant method is **superlinear** — i.e. faster than linear (the bisection method) but slower than quadratic (Newton-Raphson), whilst retaining the Newton-Raphson conditions for convergence that initial guesses be within the neighborhood of the actual root, and that  $f''(x)$  must be continuous within the region of iteration. The actual rate of convergence for the secant method is of order  $\varphi \approx 1.618$

If the secant method converges slower than Newton's method (indeed, the secant method has a rate of convergence of  $1.618\dots$ , compared to 2 for Newton's method), why would we ever choose the secant method over Newton's method? Recall that Newton's method requires the computation of both  $f(x)$  and  $f'(x)$  — this is fine if  $f(x)$  is known and is relatively simple in form, as  $f'(x)$  can be easily calculated analytically and inserted directly into the algorithm. However, if  $f'(x)$  needs to be calculated *numerically*, then in practice, multiple iterations of the secant method can be undertaken in the same time it takes to calculate one Newton-Raphson iteration *plus* the numerical derivative  $f'(x)$ .

---

<sup>1</sup>Please let it be! we hear you shout. Not more I have to duly memorise!

**Example 4.1** (Fortran)

Find the root of  $\sqrt{x+1} \cos^3(x/2)$  in the region  $0 \leq x \leq 2\pi$  using the secant method

**Solution:**

```

program secant
    implicit none
    real(8) :: x0, x1, x2, fx0, fx1, interval, tol
    integer :: i, N
    logical :: rootFound = .false.

    x0 = 1           ! set initial estimate x0
    x1 = 2           ! set initial estimate x1
    tol = 1.d-15     ! set the tolerance as 10^-15

    ! loop 500 times OR convergence reached
    do i = 1,500
        ! iterate the secant method
        fx0 = sqrt(x0+1.d0)*cos(x0/2.d0)**3
        fx1 = sqrt(x1+1.d0)*cos(x1/2.d0)**3
        x2 = x1 - fx1 * (x1-x0)/(fx1-fx0)

        interval = abs(x2-x1)/abs(x1)

        ! check convergence
        if (interval <= tol) then
            N = i
            rootFound = .true.
            exit
        end if

        x0 = x1 ! iterate variables
        x1 = x2 ! ready for next loop
    end do

    if (rootFound) then
        write(*,*) 'After ',N,' iterations, the root is',x2
    else
        write(*,*) 'Secant method did not converge'
    end if
end program secant

```

**Output:** After 115 iterations, the root is 3.1415926535897851

**Example 4.2** (Python)

Find the root of  $\sqrt{x+1} \cos^3(x/2)$  in the region  $0 \leq x \leq 2\pi$  using the secant method

**Solution:**

```
#!/usr/bin/env python3
import numpy as np

x0 = 1      # set initial estimate x0
x1 = 2      # set initial estimate x1
tol = 1e-15 # set the tolerance as 10^-15

def f(x):
    return np.sqrt(x+1)*np.cos(x/2)**3

# loop 500 times OR convergence reached
for i in range(500):
    # store the function values
    fx0 = f(x0)
    fx1 = f(x1)

    # iterate the secant method
    x2 = x1 - fx1 * (x1-x0)/(fx1-fx0)

    interval = abs(x2-x1)/abs(x1)

    # check convergence
    if interval <= tol:
        N = i + 1
        rootFound = True
        break

    # iterate variables ready for next loop
    x0 = x1
    x1 = x2
else:
    rootFound = False

if rootFound:
    print('After {} iterations, the root is {}'.format(N,x2))
else:
    print('Secant method did not converge')
```

**Output:** After 115 iterations, the root is 3.141592653589785

- Why do we store the function values, instead of just using  $f(x_0)$  and  $f(x_1)$  in the iteration line?

Every time we make a function call, we use a bit more processing power; for a time consuming function, we optimise execution time by reducing the number of function calls.

## 4.6 False Position Method

So far we have considered the bisection method, which uses a **bracketed** approach around the root and the intermediate value theorem to ensure convergence, as well as the Newton-Raphson and secant methods that do *not* bracket the root but have a higher rate of convergence.

It turns out that it is quite easy to rewrite the secant method as a **bracketed** method, to significantly increase the rate of convergence – the result is the **method of false positions**.

### False Position Algorithm

Choose an appropriate interval  $x \in [a_0, b_0]$ , such that  $f(a_0)f(b_0) < 0$

Then, for  $n = 0, 1, 2, \dots, N$

1. Estimate the root using the secant formula

$$c_n = b_n - f(b_n) \frac{b_n - a_n}{f(b_n) - f(a_n)}$$

2. If  $|b_n - a_n|/|a_n| \leq \epsilon$  then it has converged to a root; **exit** here
3. Otherwise, if there is no convergence,
  - i. if  $f(a_n)f(c_n) > 0$ , then  $[a_{n+1}, b_{n+1}] = [c_n, b_n]$
  - ii. if  $f(a_n)f(c_n) < 0$ , then  $[a_{n+1}, b_{n+1}] = [a_n, c_n]$
4. Return to step 1.

This can also be thought of as a modification of the bisection method; rather than use  $c_n = (a_n + b_n)/2$  to find the midpoint, instead a ‘midpoint’ is calculated using the secant formula. In effect, we are using an algorithm that takes into account the actual function  $f(x)$  when determining the new endpoints of the bracket.

Take note however — in the paragraph above, we avoided saying that the method of false positions will ensure convergence, as the bisection method does. This is because there are edge cases where the method of false positions leads to a bracket size that does *not* converge to zero as the number of iterations tends towards infinity. In such cases, one end of the bracketed interval ends up being a fixed point, and will remain unchanged — as a result, the other end of the bracketed interval will approach the root, but never quite reach it; a badly timed real-world example of Zeno’s paradox.

In practice, we tend to handpick a collection of root-finding algorithms based on the system we wish to solve; for example, we may use the bisection method to initially narrow down the neighborhood of a root, before switching over to the Newton-Raphson method for faster convergence with reduced error.

### Reusing the wheel

If using Python, external modules provide some useful tools and pre-written algorithms for numerically calculating the root of non-linear functions. Before you sigh and promptly forget the preceding chapter, knowing the ins and outs of exactly how root finding algorithms work is incredibly important, and will give you an understanding of *when* and *how* to use each method to reduce the chance of instability and error.

On the other hand, this doesn't mean you have to hand code these algorithms every time — often, you'll find that well-established libraries (such as **NumPy** and **SciPy** for Python) have highly optimised implementations built-in.

For example, SciPy includes a whole sub-package, `scipy.optimize`, that provides various numerical root finding algorithms. These include in-built bisection, Newton-Raphson, and secant methods, as well as several others we have not covered here. Not only that, but SciPy also provides the ability to perform general purpose root finding on functions of more than one variable.

For example, to use the Newton-Raphson method to find the root of the function  $x^3 - 7x - 19 = 0$ , we can use the method `scipy.optimize.newton`:

```
>>> import numpy as np
>>> from scipy.optimize import newton
>>> f = lambda x: x**3-7*x-19
>>> df = lambda x: 3*x**2-7
>>> x0 = 5 #initial guess
>>> newton(f, x0, df)
3.5208545993194003
```

Note that, in the example above, if the derivative of the function is not provided, then the secant method is used instead.

## Exercises

- P4.1** Prove that the Secant method has a superlinear convergence given by

$$|x_{n+1} - L| \propto |x_n - L|^\phi$$

where  $\phi = (1 + \sqrt{5})/2$  is the golden ratio, and  $L$  is an exact root,  $f(L) = 0$ .

*Hint:* Show that

$$x_{n+1} - L = \frac{f''(L)}{2f'(L)}(x_n - L)(x_{n-1} - L)$$

by taking the Taylor expansion of the secant method. In the limit  $n \rightarrow \infty$ , this recurrence relation then has the solution

$$x_{n+1} - L = \left( \frac{f''(L)}{2f'(L)} \right)^\phi (x_n - L)^\phi$$

- P4.2** Use the bisection method to find a root of

$$g(x) = f(f(f(f(x))))$$

where

$$f(x) = x^3 - 2.2x.$$

- P4.3** Rewrite the secant method program in Example 4.1 to use the bisection method to find  $x_0$  and  $x_1$  in a neighborhood of the root, rather than the method of just guessing.

- P4.4** Rewrite the secant method program in Example 4.1 to use the method of false positions.

### P4.5 Realistic projectile motion

A good model for the force of air-resistance or drag on objects undergoing projectile motion is

$$\mathbf{F}_d(t) \approx -D|\mathbf{v}(t)|\mathbf{v}(t)$$

where  $D$  is the drag coefficient, and  $\mathbf{v} = \mathbf{r}'(t) = (x'(t), y'(t))$  is the velocity of the projectile.

Writing out Newton's equation of motion and accounting for gravitational force in the downward  $y$  direction,

$$\begin{aligned} m\mathbf{a}(t) &= \mathbf{F}_{total} \\ &= \mathbf{F}_D(t) + \mathbf{F}_g \\ &= -D|\mathbf{v}(t)|\mathbf{v}(t) - mg\hat{\mathbf{y}} \end{aligned}$$

► The drag coefficient  $D$  depends on the shape/resulting air flow over the projectile.

or,

$$\mathbf{a}(t) = -\frac{D}{m}|\mathbf{v}(t)|\mathbf{v}(t) - g\hat{\mathbf{y}}$$

- (a) Approximate the first order coupled ODEs

$$\mathbf{v}(t) = \frac{d\mathbf{r}}{dt} \quad \text{and} \quad \mathbf{a}(t) = \frac{d\mathbf{v}}{dt}$$

using a method of finite differences to derive expressions for  $\mathbf{r}(t + \Delta t)$  and  $\mathbf{v}(t + \Delta t)$  in terms of  $\mathbf{r}(t)$ ,  $\mathbf{v}(t)$  and  $\Delta t$

- (b) Write a program to compute and plot the trajectory of a  $m = 1$  kg canon shell for different values of the firing angle, with  $D = 0.00004$  kg/m and  $|\mathbf{v}(0)| = 700\text{ms}^{-1}$ .
- (c) Use the bisection method to work out the required firing angle  $\theta$  for the projectile to hit a target located at ground level 19 km away.
- (d) Calculate the **minimum** initial velocity and resulting firing angle required to hit a target located at ground level 32 km away.

*Hint:* You will need to bisect over initial velocity, testing maximum horizontal range at various values of  $\theta$ . If the initial velocity provides enough range, then bisect over the firing angle  $\theta$  to determine the angle required.

Note that this might not be optimised – you may need to repeat the two bisections with a finer grain and smaller intervals.

- (e) Solve equations of motion to get the analytical solution in the case of no drag, i.e.  $D = 0$ .  
Compare this to the output of your code when  $D = 0$ , and provide an error analysis

#### P4.6 Bonus question

An even more realistic model for projectile motion with drag should take into account changes in air density with height and wind velocity. Modify your code from Question 4.5 to include:

- (a) Change in air density with height

$$D(y) = D_0 e^{-y/y_0}$$

where  $y$  is the altitude,  $y_0 = 10\text{km}$ , and  $D_0 = 0.00004\text{kgm}^{-1}$

- (b) Constant wind velocity in a fixed direction

$$\mathbf{F}_D(t) = -D|\mathbf{v}(t) - \mathbf{v}_w|(\mathbf{v}(t) - \mathbf{v}_w)$$

where  $\mathbf{v}_w = (v_w, 0)$  is a constant vector in the  $x$  direction.



## Chapter 5

# Differentiation and Initial Value Problems

Differential equations describe a wide variety of physical phenomena, however not all systems of differential equations can be solved analytically. Thus, numeric approximations fill an important void, providing us with methods to model and analyse physical systems when analytic tools fall short.

In this chapter, we will provide a brief introduction to numerical methods of solving first order ordinary differential equations (ODEs) using finite-difference methods.

### 5.1 Method of Finite Differences

#### 5.1.1 Forwards Difference

Consider the derivative of a function  $f(x)$ , defined by

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}. \quad (5.1)$$

How might we evaluate this using a computer? Assuming we keep  $h$  small (i.e.  $h \ll 1$ ), we might think that a fairly reasonable approximation might be

$$f'(x) \approx \frac{f(x + h) - f(x)}{h}, \quad h \ll 1. \quad (5.2)$$

This is known as an example of a **finite-difference** approximation, with  $h$  referred to as the **step size**.

#### Example 5.1

Numerically calculate the derivative of  $f(x) = e^x \sin(x^2)$  at  $x = 3$ .

**Solution:**

Using the chain and product rule, the derivative of this function can be easily calculated:

$$\begin{aligned} f'(x) &= e^x \sin(x^2) + 2xe^x \cos(x^2) \\ &= e^x(1 + 2x \cos(x^2)). \end{aligned}$$

However, let us approximate the solution numerically with the finite-difference formula:

$$\begin{aligned} f'(x) &\approx \frac{f(x+h) - f(x)}{h} \\ &= \frac{e^{x+h} \sin((x+h)^2) - e^x \sin(x^2)}{h}. \end{aligned}$$

Choosing  $h = 0.001$  (such that  $h \ll 1$ ), we find

$$f_h(3) = \frac{e^{3.001} \sin(3.001^2) - e^3 \sin(3^2)}{0.001} = -101.798194$$

Comparing this to the exact result from the analytic solution,  $f'(3) = -101.525622$ , we can see there is an error of  $\sim 0.3\%$  in our finite-difference approximation for  $h = 0.001$ .

But can we do better, and actually quantify the error in the finite-difference approximation? Let us consider the Taylor series expansion of  $f(x+h)$ . Recall that the Taylor series is defined by

$$f(x) = \sum_{n=0}^{\infty} \frac{1}{n!} (x-x_0)^n f^{(n)}(x_0). \quad (5.3)$$

Now, expanding  $f(x+h)$  as a Taylor series of variable  $h$ , around the initial point  $x = x_0$ ,

$$f(x+h) = \sum_{n=0}^{\infty} \frac{1}{n!} (x+h-x)^n f^{(n)}(x) = \sum_{n=0}^{\infty} \frac{1}{n!} h^n f^{(n)}(x), \quad (5.4)$$

and writing out the first few terms,

$$f(x+h) = f(x) + h f'(x) + \frac{1}{2} h^2 f''(x) + \frac{1}{6} h^3 f'''(x) + \mathcal{O}(h^4), \quad (5.5)$$

where the term  $\mathcal{O}(h^4)$  indicates that we are neglecting terms of order  $h^4$  or smaller. Solving this equation for  $f'(x)$ , we recover the finite-difference equation, now with additional higher order terms:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2} h f''(x) - \frac{1}{6} h^2 f'''(x) + \dots, \quad (5.6)$$

or, using big O notation,

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h).$$

(5.7)

- For more information, see [Big O notation](#). This notation is commonly used in computer science to indicate how algorithms scale (e.g. their running time) depending on the size of input variables.

This tells us that this is a **first-order** finite-difference approximation – the error is proportional to our step size,  $h$ , and thus scales linearly (i.e. if we were to plot the error vs.  $h$ , we would expect to see a linear relationship).

### 5.1.2 Backwards Difference

In the previous section, we derived the **forward difference** formula; so-called because it depends on the terms  $f(x)$  and  $f(x + h)$  – i.e. we require the function value at the point we are interested in calculating  $f'(x)$  as well as the function value at some small step *forwards*.

By expanding  $f(x - h)$ , we can also define an analogous **backwards difference** formula:

$$f'(x) = \frac{f(x) - f(x - h)}{h} + \frac{1}{2}hf''(x) - \frac{1}{6}h^2f'''(x) + \dots, \quad (5.8)$$

or in big O notation,

$$f'(x) = \frac{f(x) - f(x - h)}{h} + \mathcal{O}(h). \quad (5.9)$$

Like the forward difference method, this is still a first order method, however it now depends on the current function value and a ‘previous’ function value.

### 5.1.3 Central Differences

If you look at the forward and backwards finite difference formulas, you may notice that they contain  $-hf''(x)/2$  and  $+hf''(x)/2$  terms respectively. Adding these expressions together, we get:

$$\begin{aligned} 2f'(x) &= \frac{f(x + h) - f(x)}{h} - \frac{1}{2}hf''(x) - \frac{1}{6}h^2f'''(x) \\ &\quad + \frac{f(x) - f(x - h)}{h} + \frac{1}{2}hf''(x) - \frac{1}{6}h^2f'''(x) + \dots. \end{aligned} \quad (5.10)$$

Cancelling out and simplifying terms,

$$f'(x) = \frac{f(x + h) - f(x - h)}{2h} - \frac{1}{6}h^2f'''(x) + \dots \quad (5.11)$$

we arrive at the so-called **central difference** formula:

$$f'(x) = \frac{f(x + h) - f(x - h)}{2h} + \mathcal{O}(h^2). \quad (5.12)$$

Unlike the forward and backwards methods, this is a **second-order** finite difference approximation – the numerical error will scale as a function of the *square* of the step size, resulting in a closer approximation than the forward or backwards methods for the same step size.

### Higher Order Approximations

By playing with the Taylor series for various forms of  $f(x + nh)$  (where  $n = \dots, -1, 0, 1, 2, \dots$ ), we can derive approximations to the derivative up to arbitrary order; just as we did above in deriving the central difference formula.

For example, we can eliminate the  $h^2$  order terms by adding together the two-step forward difference formula ( $f_{+2h}$ , formed from the series expansion of  $f(x + 2h)$ ), backwards difference formula ( $f_{-h}$ ), and the central difference formula ( $f_{\pm h}$ ):

$$\begin{aligned} f'(x) &= \frac{1}{-3} (f_{+2h} + 2f_{-h} - 6f_{\pm h}) \\ &= \frac{6f(x+h) - 2f(x-h) - f(x+2h) - 3f(x)}{6h} + \mathcal{O}(h^3). \end{aligned}$$

A fourth-order approximation can also be derived:

$$\begin{aligned} f'(x) &= \frac{1}{-6} (f_{+2h} + f_{-2h} - 8f_{\pm h}) \\ &= \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h} + \mathcal{O}(h^4). \end{aligned}$$

Many web resources exist for full tables of finite-difference coefficients for varying accuracy, including [Wikipedia](#), which has a nice summary available.

#### 5.1.4 The Second Derivative

As in the previous sections, we can also manipulate the Taylor series expansions of  $f(x + h)$  and  $f(x - h)$  to find a finite difference approximation to the second derivative:

$$f(x + h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + \dots \quad (5.13)$$

$$f(x - h) = f(x) - hf'(x) + \frac{1}{2}h^2 f''(x) - \frac{1}{6}h^3 f'''(x) + \dots \quad (5.14)$$

Summing these together,

$$f(x + h) + f(x - h) = 2f(x) + h^2 f''(x) + \mathcal{O}(h^4), \quad (5.15)$$

and rearranging to solve for  $f''(x)$ ,

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2). \quad (5.16)$$

This is the **second derivative central difference** formula, accurate to second order in  $h$ .

- You can also derive the finite difference formula for  $f''(x)$  by applying the first derivative forward difference, followed by the first derivative backwards difference – give it a go and see.

### Other Second Derivative Finite Difference Formulas

Like the first derivative approximations, there is also a forward finite difference formula for the second derivative

$$f''(x) = \frac{f(x+2h) - 2f(x+h) + f(x)}{h^2} + \mathcal{O}(h)$$

and a backwards difference formula

$$f''(x) = \frac{f(x) - 2f(x-h) + f(x-2h)}{h^2} + \mathcal{O}(h).$$

Again, as for the first derivative case, these are of *lower accuracy* than the central difference formula.

Similarly, higher order formulas can also be constructed:

$$f''(x) = \frac{-f(x+2h) + 16f(x+h) - 30f(x) + 16f(x-h) - f(x-2h)}{12h^2} + \mathcal{O}(h^4).$$

## 5.2 The Euler Method(s)

Suppose that we have an initial value problem

$$\frac{dy}{dx} = f(y(x), x), \quad y(x_0) = y_0. \quad (5.17)$$

- Using the forward difference approximation produces an **explicit** Euler method – the next point in the solution is determined by previous points.

You could also use backwards and central approximations to derive **implicit** Euler methods; these will be discussed further on.

How can we solve this ODE numerically? To start with, let's try applying the forward difference approximation to the left-hand side of the differential equation:

$$\frac{y(x+h) - y(x)}{h} \approx f(y(x), x). \quad (5.18)$$

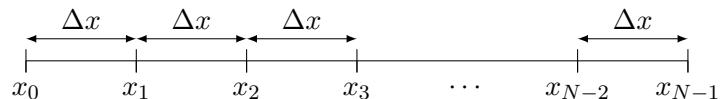
Rearranging, this gives us

$$y(x+h) \approx y(x) + h f(y(x), x). \quad (5.19)$$

That is, we can approximate the value of  $y(x+h)$  simply from the current value  $y(x)$  – this is known as the **forward Euler method**, and is sometimes referred to simply as *the* Euler method.

### 5.2.1 The Forward Euler Method

But the above quantities ( $y(x)$ ,  $x$ ) are *continuous* variables – how does this translate to a computer program? In order to do that, we need to **discretise** our system. To begin with, we choose a suitable step-size  $h = \Delta x$  (small enough to reduce truncation error from the finite difference approximation, but not so small that precision/rounding errors are significant), and discretise the  $x$ -coordinate to produce a **discrete  $x$  grid**:



- Note that, as step size  $\Delta x$  is constant along the  $x$ -grid, we have the relationship

$$x_n = x_0 + n\Delta x.$$

Thus, since we know that for the final point in the grid,

$$x_{N-1} = x_0 + (N-1)\Delta x,$$

we can rearrange to get an expression for  $N$  in terms of  $\Delta x$ :

$$N = \frac{x_{N-1} - x_0}{\Delta x} + 1.$$

That is, we have chosen  $N$  discrete values of  $x$ , with  $x_0 \leq x \leq x_{N-1}$ , each separated by step size  $\Delta x$  (i.e.  $x_{n+1} - x_n = \Delta x$ ).

We can now use the  $x$ -grid to discretise the function values,  $y$ :

$$\begin{aligned} y_0 &= y(x_0) \\ y_1 &= y(x_1) \\ &\vdots \\ y_{N-1} &= y(x_{N-1}) \end{aligned}$$

Thus, in terms of our new discretised variables  $x_n$  and  $y_n$ , and step size  $\Delta x$ , the Euler method can be written

$$y_{n+1} \approx y_n + \Delta x f(y_n, x_n). \quad (5.20)$$

**Example 5.2 (Fortran)**

Use the forward Euler method to solve the initial value problem  $y'(x) = xy(x)$ ,  $y(0) = 2$  between  $0 \leq x \leq 1$

**Solution:**

Solving this ODE analytically is quite simple; simply apply separation of variables, and integrate both sides,

$$\begin{aligned}\frac{dy}{dx} = xy &\Rightarrow \int \frac{1}{y} dy = \int x dx \\ &\Rightarrow \ln y = \frac{1}{2}x^2 + C \\ &\Rightarrow y(x) = Ke^{x^2/2}\end{aligned}$$

and apply the initial condition to get  $y(x) = 2e^{x^2/2}$ .

To solve this numerically, let's choose  $\Delta x = 0.01$ ; this results in a grid of size  $N = (1 - 0)/\Delta x + 1 = 101$ . Applying the Euler method, we get the iterative equation

$$y_n = y_{n-1} + \Delta x(x_{n-1}y_{n-1})$$

Writing a simple Fortran program to implement this:

```
program euler
    implicit none

    integer :: i, N
    real    :: xmin, xmax, x, y, dx

    ! set x-grid size and spacing
    dx = 0.01
    xmin = 0.
    xmax = 1.
    N = (xmax-xmin)/dx + 1

    ! set initial conditions
    x = 0.
    y = 2.

    ! write initial conditions
    open(10, file='results.txt')
    write(10, '(f8.2,f8.4)')x,y

    do i=1,N-1
        y = y + dx*(x*y) ! The Euler method
        x = x + dx        ! increment x
        write(10, '(f8.2,f8.4)')x,y
    end do
    close(10)
end program euler
```

Alternatively, we could implement this using Fortran arrays:

```

program eulerarray
  implicit none
  integer :: i, N
  real    :: xmin, xmax, dx
  real, allocatable :: x(:), y(:)

  ! set x-grid size and spacing
  dx = 0.01; xmin = 0.; xmax = 1.
  N = (xmax-xmin)/dx + 1

  ! allocate x and y arrays
  allocate(x(1:N), y(1:N))
  x = [(xmin + dx*i, i=0,N-1)]
  y = 0.
  y(1) = 2.

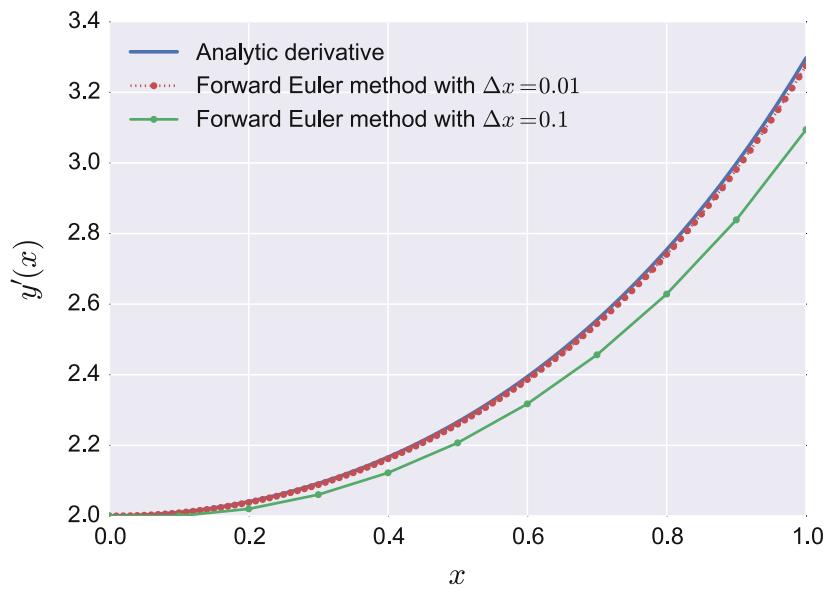
  ! The Euler method
  do i=1,N-1
    y(i+1) = y(i) + dx*(x(i)*y(i))
  end do

  ! write array to a file
  open(10, file='results.txt')
  do i=1,N
    write(10, '(f8.2,f8.4)')x(i),y(i)
  end do
  close(10)

  deallocate(x,y)
end program eulerarray

```

Below, the output of the Euler algorithm from the example above is plotted and compared to the exact analytic solution  $y'(x)$  of the differential equation.



**Figure 5.1** Numerical results of the Forward Euler method compared with exact analytic solution

**Example 5.3 (Python)**

Use the forward Euler method to solve the initial value problem  $y'(x) = xy(x)$ ,  $y(0) = 2$  between  $0 \leq x \leq 1$

**Solution:**

Solving this ODE analytically is quite simple; simply apply separation of variables, and integrate both sides,

$$\begin{aligned}\frac{dy}{dx} = xy &\Rightarrow \int \frac{1}{y} dy = \int x dx \\ &\Rightarrow \ln y = \frac{1}{2}x^2 + C \\ &\Rightarrow y(x) = Ke^{x^2/2}\end{aligned}$$

and apply the initial condition to get  $y(x) = 2e^{x^2/2}$ .

To solve this numerically, let's choose  $\Delta x = 0.01$ ; this results in a grid of size  $N = (1 - 0)/\Delta x + 1 = 101$ . Applying the Euler method, we get the iterative equation

$$y_n = y_{n-1} + \Delta x(x_{n-1}y_{n-1}).$$

Writing a simple Python program, making use of NumPy arrays to implement this:

```
#!/usr/bin/env python3
import numpy as np

# set x-grid size and spacing
dx = 0.01
x = np.arange(0, 1+dx, dx)
N = len(x)

# create the y-grid
y = np.zeros((N))

# set the initial condition
y[0] = 2

for i in range(0,N-1):
    # the Euler method
    y[i+1] = y[i] + dx**2*x[i]*y[i]

# save the results to a file
np.savetxt('results.txt', np.transpose([x,y]), fmt='%.4f')
```

### 5.2.2 The Backwards Euler Method

Using the backwards finite difference approximation, we can alternatively derive the **backwards Euler method**:

$$\begin{aligned} y'(x) = f(y(x), x) &\Rightarrow \frac{y(x) - y(x-h)}{h} \approx f(y(x), x) \\ &\Rightarrow y(x) \approx y(x-h) + hf(y(x), x). \end{aligned}$$

Making the variable transformation  $x \rightarrow x + h$ , this can be written

$$y(x+h) \approx y(x) + hf(y(x+h), x+h),$$

or in discrete form as

$$y_{n+1} \approx y_n + \Delta x f(y_{n+1}, x_{n+1}).$$

Whilst this may appear similar to the forwards Euler method, we now have a  $y_{n+1}$  term on the right hand side of the equation — the backward Euler method is an example of an **implicit** method. In some cases, we may still be able to rearrange the backward Euler method to get an **explicit** method (i.e. one where the value to be calculated depends only on previous values). For example, let  $f(y(x), x) = xy(x)$  as in Example 5.2:

$$y_{n+1} = y_n + \Delta x(x_{n+1}y_{n+1}) \Rightarrow y_{n+1} = \frac{y_n}{1 - x_{n+1}\Delta x}.$$

This is an **explicit** equation, as  $y_{n+1}$  only appears on the left hand side.

However, this is *not* the case for general  $f(y, x)$ . (For example, try the cases  $f(y(x), x) = e^{y(x)}$  or  $f(y(x), x) = \sin(x)$ .) Instead, we must rearrange the equation like so,

$$y_n - y_{n+1} + f(y_{n+1}, x_{n+1})\Delta x = 0,$$

and then find the roots of  $y_{n+1}$  numerically at each iteration.<sup>1</sup>

---

<sup>1</sup>See Chap. 4 for more details on root-finding algorithms.

**Example 5.4 (Fortran)**

Solve  $y'(x) = e^{-y}$ ,  $y(0) = 0$ , using the backwards Euler method for  $0 \leq x \leq 1$  with  $\Delta x = 0.01$

**Solution:** The solution to this ODE is  $y(x) = \log(x + 1)$  (try solving this yourself!). However, let's walk through the implicit backwards Euler method instead.

Applying the backwards Euler method for this system, we arrive at the following *implicit* set of equations we must solve for  $y_{n+1}$ :

$$\begin{aligned} y_0 &= 0 \\ y_n - y_{n+1} + e^{-y_{n+1}} \Delta x &= 0, \quad n = 1, 2, \dots, 101. \end{aligned}$$

In order to find the roots of this equation at each time step, let's use the Newton–Raphson method. As a quick reminder, to find the roots of  $g(x) = 0$  using the Newton–Raphson method, we first estimate a value of the root  $x^{(j)}$ , and then calculate a better estimate  $x^{(j+1)}$  using

$$x^{(j+1)} = x^{(j)} + \frac{g(x^{(j)})}{g'(x^{(j)})}.$$

This process is repeated until we have the required accuracy.

In our case, we have

$$\begin{aligned} g(y_{n+1}) &= y_n - y_{n+1} + e^{-y_{n+1}} \Delta x \\ g'(y_{n+1}) &= -1 - e^{-y_{n+1}} \Delta x, \end{aligned}$$

and thus the Newton–Raphson iterative process can be written as

$$y_{n+1}^{(j+1)} = y_{n+1}^{(j)} - \frac{y_n - y_{n+1}^{(j)} + e^{y_{n+1}^{(j)}} \Delta x}{-1 - e^{-y_{n+1}^{(j)}} \Delta x}.$$

If we choose  $\Delta x$  sufficiently small, it is reasonable to expect that  $\Delta y = y_{n+1} - y_n$  is also small, so let's choose as our first estimate of the root  $y_{n+1}^{(0)} = y_n$ . Allowing 5 Newton–Raphson iterations to find each value of  $y_{n+1}$ , the following Fortran program solves the ODE using the backwards Euler method:

```
program beuler
    implicit none
    integer :: i, j, N
    real     :: x, y, y0, dx

    ! set x-grid size and spacing
    dx = 0.01
    N = (1.0 - 0.0)/dx + 1

    ! set initial conditions
    x = 0.0;      y = 0.0
```

```

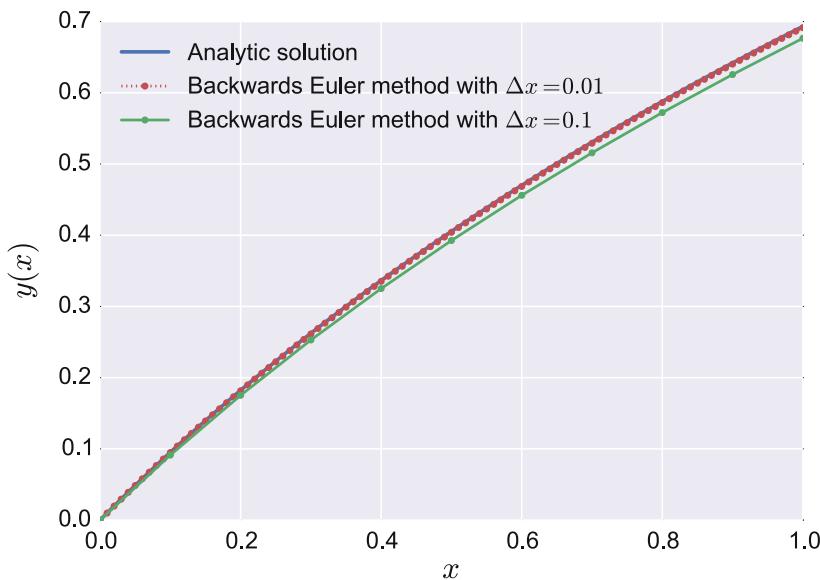
! write initial conditions
open(10, file='results.txt')
write(10, '(f8.2,f8.4)')x,y

do i=1,N-1
    y0 = y      ! estimate the root

    ! iterate the Newton-Raphson method
    do j=1,5
        y0 = y0 - (y - y0 + dx*exp(-y0)) / (-1. - dx*exp(-y0))
    end do

    y = y0      ! increment y
    x = x + dx  ! increment x
    write(10, '(f8.2,f8.4)')x,y
end do
close(10)
end program beuler

```



**Figure 5.2** Numerical results of the Backward Euler method compared with exact analytic solution

### Example 5.5 (Python)

Solve  $y'(x) = e^{-y}$ ,  $y(0) = 0$ , using the backwards Euler method for  $0 \leq x \leq 1$  with  $\Delta x = 0.01$

**Solution:** The solution to this ODE is  $y(x) = \log(x + 1)$  (try solving this yourself!). However, let's walk through the implicit backwards Euler method instead.

Applying the backwards Euler method for this system, we arrive at the following *implicit* set of equations we must solve for  $y_{n+1}$ :

$$\begin{aligned} y_0 &= 0 \\ y_n - y_{n+1} + e^{-y_{n+1}} \Delta x &= 0, \quad n = 1, 2, \dots, 101. \end{aligned}$$

In order to find the roots of this equation at each time step, let's use the Newton–Raphson method. As a quick reminder, to find the roots of  $g(x) = 0$  using the Newton–Raphson method, we first estimate a value of the root  $x^{(j)}$ , and then calculate a better estimate  $x^{(j+1)}$  using

$$x^{(j+1)} = x^{(j)} + \frac{g(x^{(j)})}{g'(x^{(j)})}.$$

This process is repeated until we have the required accuracy.

In our case, we have

$$\begin{aligned} g(y_{n+1}) &= y_n - y_{n+1} + e^{-y_{n+1}} \Delta x \\ g'(y_{n+1}) &= -1 - e^{-y_{n+1}} \Delta x, \end{aligned}$$

and thus the Newton–Raphson iterative process can be written as

$$y_{n+1}^{(j+1)} = y_{n+1}^{(j)} - \frac{y_n - y_{n+1}^{(j)} + e^{-y_{n+1}^{(j)}} \Delta x}{-1 - e^{-y_{n+1}^{(j)}} \Delta x}.$$

If we choose  $\Delta x$  sufficiently small, it is reasonable to expect that  $\Delta y = y_{n+1} - y_n$  is almost reasonably small, so let's choose as our first estimate of the root  $y_{n+1}^{(0)} = y_n$ . Allowing 5 Newton–Raphson iterations to find each value of  $y_{n+1}$ , the following Python program solves the ODE using the backwards Euler method:

```
#!/usr/bin/env python3
import numpy as np

# set x-grid size and spacing
dx = 0.01
x = np.arange(0, 1+dx, dx)
N = len(x)

# create the y-grid and initial conditions
y = np.zeros((N))
y[0] = 0
```

```
# the backward Euler method
for i in range(0,N-1):
    # estimate the root based on previous value
    y[i+1] = y[i]

    # iterate the Newton-Raphson method
    for j in range(5):
        num = y[i] - y[i+1] + dx*np.exp(-y[i+1])
        denom = -1 - dx*np.exp(-y[i+1])
        y[i+1] -= num/denom

# save the results to a file
np.savetxt('results.txt', np.transpose([x,y]), fmt='%.4f')
```

### 5.3 Numerical Error

**Step truncation error** is the **local** error that occurs at each time step. This can be calculated by taking the Taylor series of  $y(x + h)$ ,

$$y(x + h) = y(x) + hy'(x) + \frac{1}{2}h^2y''(x) + \mathcal{O}(h^3),$$

and subtracting the forward Euler formula  $y_{FE}(x + h) \approx y(x) + hf(y, x)$ :

$$\begin{aligned} y(x + h) - y_{FE}(x + h) &= \left[ y(x) + hy'(x) + \frac{1}{2}h^2y''(x) + \mathcal{O}(h^3) \right] - [y(x) + hy'(x)] \\ &= \frac{1}{2}h^2y''(x) + \mathcal{O}(h^3) \\ &= \mathcal{O}(h^2). \end{aligned}$$

- Recall that the ODE we are solving is

$$y'(x) = f(y, x)$$

so by definition

$$f(y, x) = y'(x).$$

Thus the forward Euler method has a local or step truncation error at each iteration that is **bounded** by  $h^2$  for  $h \ll 1$ .

**Global error** is the error that results due to the local or step error accumulating over numerous iterations. For example, say  $n$  steps of the Euler method have been iterated – we know that

$$x_{n-1} = x_0 + nh$$

or rearranging this

$$n = \frac{x_{n-1} - x_0}{h}$$

Thus, the total global error must be bounded by

$$n \times \text{step error} = \frac{x_{n-1} - x_0}{h} \times \frac{1}{2}h^2y'(x) = \mathcal{O}(h).$$

As the global error scales linearly with  $h$ , the forward Euler method is a **first-order** algorithm.

- Like the forwards Euler method, the backwards Euler method has a global error that scales like  $\mathcal{O}(h)$  i.e. it is **first-order**.

See if you can prove this!

## 5.4 Stability

When working with non-linear ODEs, as we saw in the previous section, the backwards Euler method can only be applied implicitly, requiring the use of numerical solvers to find the root at each time step, and causing the algorithm to be more computationally intensive.

We also saw that the two Euler methods are both first order approximations – you gain no additional accuracy by using the backwards Euler method. So why even bother using the backwards Euler method at all?

The answer: **stability**. The forward Euler method can be numerically unstable (that is, the numerical solution oscillates wildly and grows) for particular values of  $\Delta x$ , whereas the backwards Euler method has a much greater degree of stability.

To get a feel of this, consider the ODE  $y' = -ky$ ,  $k > 0$ ,  $y(0) = y_0$ .

► Stability is related to the concept of **stiffness**

Stiff differential equations are those that are *numerically unstable* for a wide range of numerical ODE methods, requiring careful thought over which methods to use.

### Case 1: Forward Euler Solution

Applying the forwards Euler scheme to express the ratio  $y_{n+1}/y_n$ ,

$$y_{n+1} = y_n - ky_n \Delta x = y_n(1 - k\Delta x) \Rightarrow \frac{y_{n+1}}{y_n} = 1 - k\Delta x$$

allows us to solve the recursion relation to find  $y_n$  in terms of  $n$ :

$$y_n = \frac{y_n}{y_{n-1}} \frac{y_{n-1}}{y_{n-2}} \dots \frac{y_1}{y_0} y_0 = y_0(1 - k\Delta x)^n.$$

Now, we know from  $\lim_{x \rightarrow \infty} e^{-kx} = 0$  that the behaviour of the solution should tend towards 0 as  $x$  increases. Since  $\lim_{n \rightarrow \infty} a^n = 0$  if and only if  $|a| < 1$ , we require

$$|1 - k\Delta x| < 1 \Rightarrow \Delta x < \frac{2}{k}$$

to ensure that the numerical solution remains stable as  $n$  increases. Otherwise, if  $\Delta x > 2/k$ , we will find that as  $n$  increases, the numerical solution oscillates and grows, with  $y_n \rightarrow \infty$  as  $n \rightarrow \infty$ .

### Case 2: Backwards Euler Solution

Applying the backwards Euler scheme to express the ratio  $y_{n+1}/y_n$ ,

$$y_{n+1} = y_n - ky_{n+1}\Delta x \Rightarrow \frac{y_{n+1}}{y_n} = \frac{1}{1 + k\Delta x}$$

allows us to solve the recursion relation to find  $y_n$  in terms of  $n$ :

$$y_n = \frac{y_n}{y_{n-1}} \frac{y_{n-1}}{y_{n-2}} \cdots \frac{y_1}{y_0} y_0 = \frac{y_0}{(1 + k\Delta x)^n}.$$

In this case, since  $k\Delta x > 0$  for all  $k > 0$ ,  $\Delta x > 0$ , we have the result

$$\lim_{n \rightarrow \infty} \frac{1}{(1 + k\Delta x)^n} = 0.$$

Thus, in this case the backwards Euler method is stable for *all* values of  $\Delta x$ .

#### 5.4.1 Forwards Euler Stability

Let's consider the stability in a slightly more rigorous fashion. Suppose we want to solve the general initial value problem  $y'(x) = f(y(x), x)$  with initial condition  $y(0) = y_0$ . Using the forward Euler method,

$$y_{n+1} = y_n + \Delta x f(y_n, x_n). \quad (5.21)$$

However, now lets assume there is some small numerical error present in the estimation of  $y_n$ ; this can be represented as  $\delta y_n$ :

$$(y_n + \delta y_n) + \Delta x f(y_n + \delta y_n, x_n). \quad (5.22)$$

Since  $\delta y \ll 1$ , we can neglect terms of order  $\delta y^2$  or higher; thus, doing the first order Taylor expansion of  $f$  around  $\delta y_n$

$$\Rightarrow y_n + \delta y_n + \Delta x \left[ f(y_n, x_n) + \delta y_n \frac{\partial}{\partial y_n} f(y_n, x_n) + \mathcal{O}(\delta y^2) \right] \quad (5.23)$$

$$\Rightarrow (y_n + \Delta x f(y_n, x_n)) + \left( 1 + \Delta x \frac{\partial f}{\partial y_n} \right) \delta y_n. \quad (5.24)$$

The first bracketed term we recognise as simply the forward Euler method applied to point  $y_n$ . Thus we can deduce that the second term represents the numerical error present in  $y_{n+1}$  due to the presence of  $\delta y_n$ :

$$\delta y_{n+1} = \left( 1 + \Delta x \frac{\partial f}{\partial y_n} \right) \delta y_n. \quad (5.25)$$

- If you are solving a **system** of ODEs

$$\mathbf{y}'(x) = \mathbf{f}(\mathbf{y}, x)$$

then replace  $f_y \equiv \partial f / \partial y$  in the stability equations here with the **eigenvalues**  $\lambda_i$  of the **Jacobian** matrix

$$(J_f)_{ij} = \frac{\partial f_i}{\partial y_j}$$

The system is stable if the stability inequality is satisfied for **all** eigenvalues.

Here,  $\delta y_{n+1}/\delta y_n$  is sometimes called the **stability function**. It is easy to see that the forward Euler method is only stable (i.e. the numerical error of successive steps does not grow unconditionally) if

$$\left| 1 + \Delta x \frac{\partial f}{\partial y} \right| \leq 1. \quad (5.26)$$

Solving this for a potentially complex function  $f(y, x)$ , and using the notation  $f_y \equiv \partial f / \partial y$ ,

$$0 < \Delta x \leq -\frac{2\operatorname{Re}(f_y)}{|f_y|^2}. \quad (5.27)$$

A couple of things to note:

|                                 |                                                                                                                                                                                                                                                                     |
|---------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                 | the system is <b>conditionally stable</b> – there exists some $\Delta x$ that satisfies $\Delta x \leq -2\operatorname{Re}(f_y)/ f_y ^2$                                                                                                                            |
| $\operatorname{Re}(f_y) < 0$    | Note that if $\operatorname{Im}(f_y)$ is large, then the value of $\Delta x$ required for stability will be exceedingly small, requiring an increase in computation time                                                                                            |
| $\operatorname{Re}(f_y) \geq 0$ | the system is <b>unconditionally unstable</b> – the right hand side is necessarily less than zero and there is <i>no</i> value of $\Delta x$ which satisfies the above equation. This includes <b>oscillating</b> first order ODEs, as $\operatorname{Re}(f_y) = 0$ |

**Table 5.1** Forward Euler stability

If we treat  $f_y \Delta x$  as a single complex variable  $z = f_y \Delta x$  (this allows for complex values of  $f_y$ ), then the forward Euler stability condition can be written

$$|1 + z| \leq 1. \quad (5.28)$$

If we plot this region in the complex plane, we get what is called a **stability diagram**; by calculating  $z = f_y \Delta x$  for a specific ODE, the stability diagram instantly tells us whether the forward Euler method will be stable.

► An oscillating ODE is one with the form

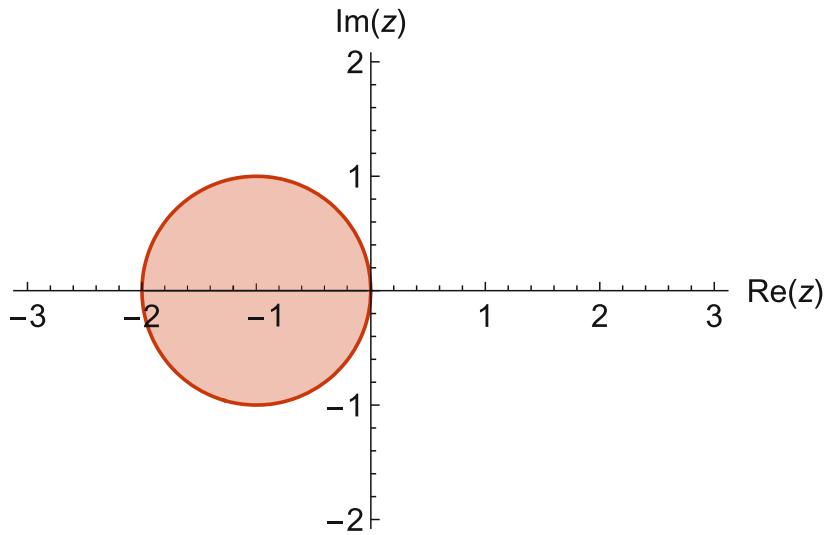
$$y'(x) = \pm i\omega y(x)$$

$$\Rightarrow f_y = \pm i\omega$$

Note that its general solution

$$y(x) = y(0)e^{\pm i\omega x}$$

does not vary in amplitude, only phase.



**Figure 5.3** Forward Euler method stability diagram

#### 5.4.2 Backwards Euler Stability

Following a similar process as in the previous section, the stability inequality for the backwards Euler method is

$$\frac{1}{\left|1 - \Delta x \frac{\partial f}{\partial y}\right|} \leq 1 \quad \Rightarrow \quad \left|1 - \Delta x \frac{\partial f}{\partial y}\right| \geq 1, \quad (5.29)$$

which, when solved, gives

$$\Delta x \geq \frac{2\text{Re}(f_y)}{|f_y|^2}. \quad (5.30)$$

It can be seen that the regions of stability for  $\Delta x$  and  $f_y$  are much larger than the forwards Euler method:

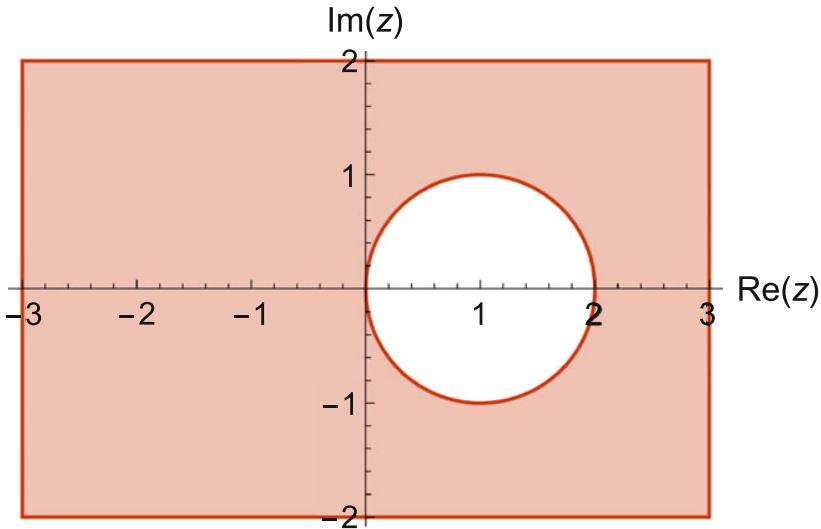
---

|                         |                                                                                                                                         |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| $\text{Re}(f_y) \leq 0$ | the system is <b>unconditionally stable</b> – any value of $\Delta x$ satisfies the stability equation. This includes oscillating ODEs! |
| $\text{Re}(f_y) > 0$    | the system is <b>conditionally stable</b> as long as $\Delta x$ satisfies $\Delta x \geq 2\text{Re}(f_y)/ f_y ^2$                       |

---

**Table 5.2** Backward Euler stability

Letting  $z = f_y \Delta x$  and plotting the stability diagram in the complex plane,  $|1 - z| \geq 1$ :



**Figure 5.4** Backward Euler method stability diagram

Compare how much larger the region of stability is compared to the forward Euler method! This is one of the main advantages of implicit numerical ODE techniques, they tend to exhibit significantly more stability at the cost of requiring a root finding algorithm.

In this case, since the entire region  $\text{Re}(z) < 0$  shows stability — note this region corresponds to decaying ODE solutions since  $f_y < 0$  — we say that the backwards Euler method is **asymptotically stable** or **A-stable**. Even more amazingly, note that if we consider the case where  $z \rightarrow \infty$ , the stability function tends towards zero:

$$\lim_{z \rightarrow \infty} \frac{1}{|1 - z|} = 0. \quad (5.31)$$

This implies that the backwards Euler method will provide a numerical solution of 0 if provided with a significantly large, single, step  $\Delta x$ . This is a stronger condition than **A-stability**, and is referred to as **L-stability**.

## 5.5 The Leap-Frog Method

Rather than use the first-order Euler finite difference formulas, let's instead solve a first order ODE using the central difference formula:

$$y'(x) = f(y(x), x) \Rightarrow \frac{y(x+h) - y(x-h)}{2h} \approx f(y(x), x) \quad (5.32)$$

$$\Rightarrow y(x+h) \approx y(x-h) + 2hf(y(x), x). \quad (5.33)$$

Or, in discrete form,

$$y_{n+1} \approx y_{n-1} + 2\Delta x f(y_n, x_n). \quad (5.34)$$

This is the **leap-frog** algorithm.

### 5.5.1 Leap-Frog Numerical Error

Subtracting the leap-frog formula from the Taylor series of  $y(x+h)$ ,

$$\begin{aligned} & y(x+h) - y_{LF}(x+h) \\ &= \left[ y(x) + hy'(x) + \frac{1}{2}h^2y''(x) + \mathcal{O}(h^3) \right] - [y(x-h) + 2hy'(x)] \\ &= \left[ y(x) - hy'(x) + \frac{1}{2}h^2y''(x) + \mathcal{O}(h^3) \right] - y(x-h), \end{aligned}$$

and substituting this into the Taylor series of  $y(x-h)$ ,

$$y(x-h) = y(x) - hy'(x) + \frac{1}{2}h^2y''(x) + \mathcal{O}(h^3),$$

we find that all terms of order  $h^2$  and less cancel out, leaving

$$y(x+h) - y_{LF}(x+h) = \mathcal{O}(h^3). \quad (5.35)$$

Therefore the leap-frog method has a step truncation error bounded by  $h^3$ . Calculating the global error after  $n$  time steps,

$$n \times \text{step error} = \frac{1}{h}(x_{n-1} - x_0) \times \mathcal{O}(h^3) = \mathcal{O}(h^2),$$

so the leap-frog method is accurate to **second-order**.

- ▶ In order to use the leap-frog method **explicitly**, two initial conditions are required,  $y_0$  and  $y_1$ .

$y_0$  is always given in initial value problems, whilst  $y_1$  can be calculated via one iteration of the forward Euler method.

### 5.5.2 Leap-Frog Stability

Let's try to work out the stability of the leap-frog method in a somewhat more rigorous manner than we have in the past. Assuming numerical error propagates each iteration of the form  $\delta y_n$ , substituting this into the leap-frog formula,

$$\begin{aligned} (y_{n+1} + \delta y_{n+1}) &= (y_{n-1} + \delta y_{n-1}) + 2\Delta x f(y_n + \delta y_n, x_n) \\ &= y_{n-1} + \delta y_{n-1} + 2\Delta x \left[ f(y_n, x_n) + \delta y_n \frac{\partial f}{\partial y_n} + \mathcal{O}(\delta y^2) \right] \\ &= (y_{n-1} + 2\Delta x f(y_n, x_n)) + \left( \delta y_{n-1} + 2\Delta x \frac{\partial f}{\partial y_n} \delta y_n \right), \end{aligned}$$

and it follows that

$$\delta y_{n+1} = \delta y_{n-1} + 2\Delta x \frac{\partial f}{\partial y_n} \delta y_n. \quad (5.36)$$

Now, we are assuming that the deviation from the exact solution grows/shrinks at the same rate with each iteration; thus, let's set  $\delta y_n = g\delta y_{n-1}$ , and  $\delta y_{n+1} = g\delta y_n = g^2\delta y_{n-1}$ :

$$g^2\delta y_{n+1} = \delta y_{n-1} + 2\Delta x \frac{\partial f}{\partial y_n} g\delta y_{n-1}. \quad (5.37)$$

Dividing through by  $\delta y_{n-1}$  and rearranging gives

$$g^2 - 2g \frac{\partial f}{\partial y} \Delta x - 1 = 0, \quad (5.38)$$

with solutions

$$g_{\pm} = \Delta x \frac{\partial f}{\partial y} \pm \sqrt{\left( \Delta x \frac{\partial f}{\partial y} \right)^2 + 1}. \quad (5.39)$$

So, what does this mean? If you multiply the two solutions together, you get  $g_+g_- = -1$ ; therefore they are related via the inverse relation

$$g_+ = -\frac{1}{g_-} \Rightarrow |g_+| = \frac{1}{|g_-|}. \quad (5.40)$$

Further, we know that in general  $g_+ \neq g_-$ , meaning **one of the solutions will have a magnitude greater than 1**, making the leap-frog method **unstable**.

But wait! What if we can find an exception where  $|g_+| = |g_-| = 1$ ? It turns out that we can – if  $\partial f / \partial y$  is *purely imaginary*, i.e.

$$\frac{\partial f}{\partial y} = i\omega, \quad \omega \in \mathbb{R}$$

Substituting this into  $g_{\pm}$ ,

$$g_{\pm} = i\omega\Delta x \pm \sqrt{1 - \omega^2\Delta x^2} \quad (5.41)$$

we see that if  $1 - \omega^2\Delta x^2 \geq 0$ , then

$$|g_{\pm}| = \sqrt{\omega^2\Delta x^2 + (1 - \omega^2\Delta x^2)} = 1. \quad (5.42)$$

Thus there is **conditional stability**, as long as the ODE is oscillating and  $\Delta x \leq 1/|\omega|$ .

---

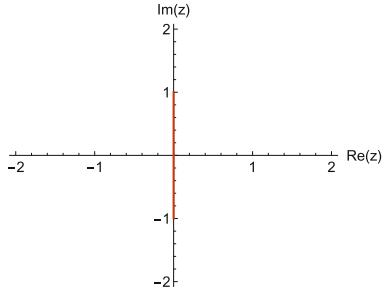
|                         |                                                                                                         |
|-------------------------|---------------------------------------------------------------------------------------------------------|
| $\text{Re}(f_y) = 0$    | the system is <b>conditionally stable</b><br>$\Delta x$ must satisfy $\Delta x \leq 1/ \text{Im}(f_y) $ |
| $\text{Re}(f_y) \neq 0$ | the system is <b>unconditionally unstable</b>                                                           |

---

**Table 5.3** Leap-frog stability

To see just how limiting the region of stability is for the leap frog method, let's plot the region of stability, given by the conditions  $|z| \leq 1$  and  $\text{Re}(z) = 0$ .

This highlights the limited region of stability compared to the forward and backward Euler methods. Like the forward Euler method, it is neither A-stable nor L-stable, and can *only* be safely used in the case of an oscillating ODE.



**Figure 5.5** Leap-frog method stability diagram

## 5.6 Round-Off Error

### Example 5.6 The atomic decay model

The decay equation models the decay of excited atoms with the first order ODE

$$N'(t) = -\frac{1}{\tau}N(t)$$

- (a) Over the domain  $0 \leq t \leq 25$ , decay constant  $\tau = 5$ , and with initial value  $N(0) = 100$ , solve for  $N(t)$  using the forwards Euler method
- (b) Using the same parameters, solve for  $N(t)$  using the leap-frog method
- (c) Calculate and compare the fractional error,

$$err(t, \Delta t) = \left| \frac{N_{\text{numeric}}(t, \Delta t) - N_{\text{exact}}(t)}{N_{\text{exact}}(t)} \right|$$

for both methods over various values of  $\Delta t$  at time  $t = 25$ .

What do you notice with small values of  $\Delta t$ ?

#### Solution:

Solving the system using the three methods required by the question,

- **Exact solution:**  $N(t) = 100e^{-t/5}$
- **Forwards Euler method:**  $N_{n+1} = N_n - \frac{1}{5}N_n\Delta t$
- **Leap-frog method:**  $N_{n+1} = N_{n-1} - \frac{2}{5}N_n\Delta t$

Note that for the leap-frog scheme, we require *two* initial conditions,  $N_0 = 100$  (already given in the question) and  $N_1$ . We can estimate  $N_1$  from the forward Euler scheme:

$$N_1 = N_0 - N_0\Delta t/5 = 100 - 20\Delta t.$$

Recall that  $t_n = t_0 + n\Delta t$ , and therefore time  $t_n = 25$  corresponds to  $n = 25/\Delta t$ . The fractional error at time  $t = 25$  can then be calculated using the formula

$$err(25, \Delta t) = \left| \frac{N_{25/\Delta t} - 100e^{-25/5}}{100e^{-25/5}} \right|.$$

Using Fortran or Python, we can implement both the leap-frog and Euler method as described here, and then calculate the fractional error. For consistency, both the Fortran and Python programs use single precision floating points numbers (`real` and `np.float32` respectively); the results are plotted in Fig. 5.6.

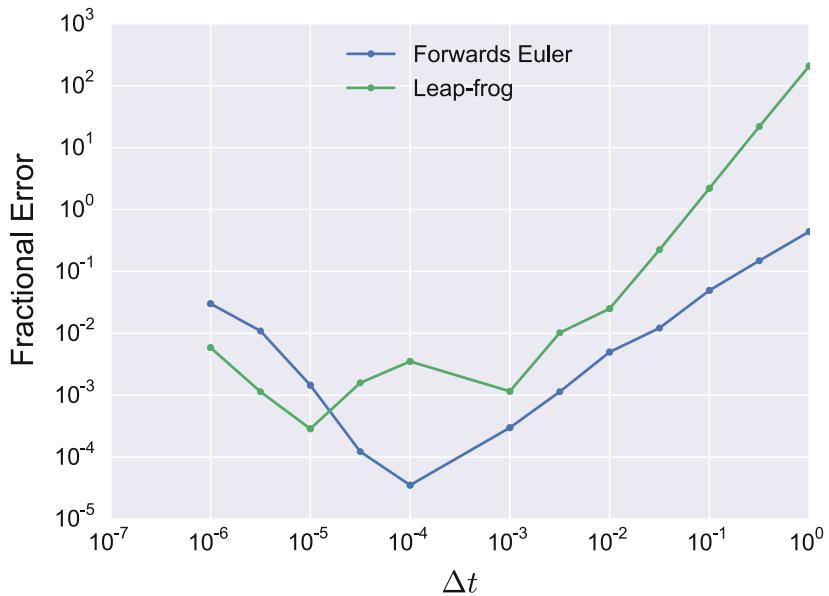


Figure 5.6 Fractional error as a function of  $\Delta t$

Looking at Fig. 5.6, we don't really get what we expect; for starters, the leap-frog algorithm fractional error is almost one order of magnitude larger than that of the Euler method! Even more troubling, for  $\Delta t < \sim 10^{-3}$ , the decrease in the error plateaus, and the error even starts to increase.

What is going on here?

The program used to output these results implemented the Euler and leap-frog method using single precision floating point numbers – what we see here is the effect of **round-off error**.

The unexpected results of Example 5.6 occurs because, when working on computers, we work with **fixed precision** real numbers, *not* always exact numbers. As we saw in Sect. 2.1.2, the use of fixed precision causes rounding errors to accumulate in the least significant digits; thus,

- for larger values of  $\Delta t$ , the algorithm's global truncation error dominates and the round-off error is negligible
- as  $\Delta t \rightarrow 0$ , the global truncation error decreases and the round-off error starts to dominate

In order to more easily avoid round-off error, let's attempt to quantify it. Recall from Sect. 2.1.2 that the upper-bound relative round-off error in floating point real numbers is given by the machine epsilon  $\epsilon$ . It turns out that after  $n$  time steps, the global round-off error is of order

$$\text{round-off error} = \mathcal{O}(\sqrt{n}\epsilon), \quad (5.43)$$

or, equivalently,

$$\text{round-off error} = \mathcal{O}\left(\frac{\epsilon}{\sqrt{\Delta x}}\right), \quad (5.44)$$

since  $n = x_0 + n\Delta x$  and thus  $n \propto 1/\Delta x$ . Therefore, for a  $k$ th order numerical method, we can sum the global truncation error and the round-off error in order to estimate the total error:

$$\text{total error} = \mathcal{O}\left(\Delta x^k + \frac{\epsilon}{\sqrt{\Delta x}}\right). \quad (5.45)$$

We can now see that round-off error begins to dominate when

$$\Delta x^k \lesssim \frac{\epsilon}{\sqrt{\Delta x}} \Rightarrow \Delta x \lesssim \epsilon^{2/(2k+1)}. \quad (5.46)$$

In Table 5.4, this has been evaluated for various values of  $k$  and  $\epsilon$  (where  $\epsilon_{sp} = 2^{-23}$  and  $\epsilon_{dp} = 2^{-52}$ ). Note that it matches pretty well with the results from Example 5.6: for the Euler ( $k = 1$ ) and the leap-frog ( $k = 2$ ), we observed round-off error dominating at  $\Delta x \sim 10^{-4}$  and  $\Delta x \sim 10^{-3}$  respectively!

|         |                  |                                |
|---------|------------------|--------------------------------|
| $k = 1$ | single precision | $\Delta x \lesssim 10^{-4.62}$ |
|         | double precision | $\Delta x \lesssim 10^{-10.4}$ |
| $k = 2$ | single precision | $\Delta x \lesssim 10^{-2.77}$ |
|         | double precision | $\Delta x \lesssim 10^{-6.26}$ |

**Table 5.4** Values of  $\Delta x$  where round-off error dominates for a  $k$ th order method

► While Fortran makes it easy to choose between single precision and double precision reals, Python and NumPy default to double precision floating point numbers.

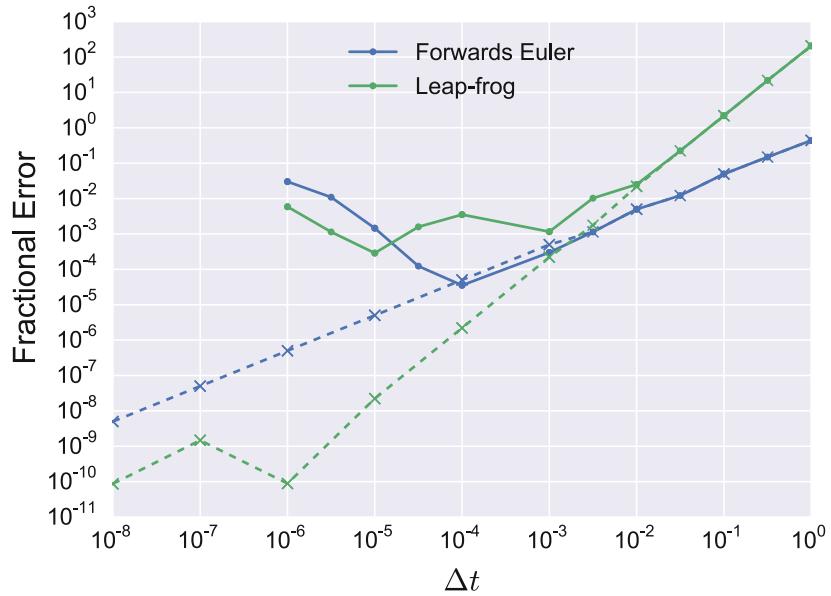
However, NumPy provides support for single precision real numbers; when creating your array, simply pass the keyword argument `dtype=np.float32`, or convert an existing array  $x$  using `np.float32(x)`.

### Example 5.7 The atomic decay model II: double precision

Repeat Example 5.6, this time using double precision `real(8)` variables

#### Solution:

By simply changing all our single precision variables to double precision (`real` → `real(8)` for Fortran, `np.float32` → `np.float64` for Python) in our code, we see a massive improvement in our accuracy for small  $\Delta t$ ; see Fig. 5.7.



**Figure 5.7** Fractional error as a function of  $\Delta t$ ; the dashed lines are calculated using double precision

This also verifies the calculation done in Table 5.4 – for the leap-frog method ( $k = 2$ ), round-off error begins to dominate for  $\Delta t \lesssim 10^{-6}$ .

Furthermore, it is now much easier to see how both methods scale with  $\Delta t$ . On the log-log plot, the first-order Euler method has a gradient of 1 (so error scales as  $\Delta t$ ), whilst the second-order leap-frog method has a gradient of 2, and thus the error scales as  $\Delta t^2$ .

## 5.7 Explicit Runge–Kutta Methods

In the previous sections of this chapter, we looked at Euler methods to solve first order initial value problems — the forwards Euler method, the backwards Euler method, and the leap-frog method. These methods are all single-step methods; only one computation at position  $(x_n, y_n)$  is required to calculate  $y_{n+1}$ . However, numerical instabilities and relatively large local truncation error means that other methods are preferred. In this chapter, we'll go over an example of one of the most used numerical methods for solving differential equations — the multi-stage Runge–Kutta method.

### 5.7.1 The Modified (Midpoint) Euler Method

Consider the first order initial value problem,

$$\frac{dy}{dx} = f(y(x), x), \quad y(x_0) = y_0. \quad (5.47)$$

Previously, we approximated the derivative using the finite-difference method — this estimates the derivative at the beginning (or end) of each discrete interval. Instead, let's solve this differential equation using the leap-frog method, but rather than use the points  $\{x - h, x, x + h\}$ , we will use  $\{x, x + h/2, x + h\}$  so as to calculate the derivative at the **midpoint** of the interval;

$$\frac{y(x+h) - y(x)}{h} = f\left(y\left(x + \frac{1}{2}h\right), x + \frac{1}{2}h\right) \quad (5.48)$$

$$\Rightarrow y(x+h) = y(x) + hf\left(y\left(x + \frac{1}{2}h\right), x + \frac{1}{2}h\right) + \mathcal{O}(h^3). \quad (5.49)$$

The problem is, we don't know what the value of  $y(x + h/2)$  is! We can find this using an intermediate forward Euler approximation:

$$y\left(x + \frac{1}{2}h\right) \approx y(x) + \frac{1}{2}hf(y(x), x). \quad (5.50)$$

This gives us a **two-step Runge–Kutta** algorithm, commonly referred to as the **midpoint two-step** or **modified Euler** algorithm:

$$y(x+h) \approx y(x) + hf\left(y(x) + \frac{1}{2}hf(y(x), x), x + \frac{1}{2}h\right). \quad (5.51)$$

Using an equally spaced discrete grid of  $N$  points  $x_0, x_1, \dots, x_{N-1}$ , where  $\Delta x = x_{n+1} - x_n \ll 1$  and  $y_n = y(x_n)$ , the two-step Runge–Kutta method is commonly written in the following form,

$$y_{n+1} = y_n + k_2 \Delta x + \mathcal{O}(\Delta x^3)$$

where

$$k_1 = f(y_n, x_n)$$

$$k_2 = f\left(y_n + \frac{1}{2}k_1 \Delta x, x_n + \frac{1}{2}\Delta x\right) \Delta x$$

(5.52)

► The Runge–Kutta method is named after two German mathematicians:

**Carl David Tolm  e Runge**  
(1856-1927) and

**Martin Wilhelm Kutta**  
(1867-1944).

For each time-step of a Runge–Kutta method, there are multiple **stages**.

► The values  $k_1$  and  $k_2$  are referred to as the **stages** of the Runge–Kutta algorithm.

If you think about it geometrically,

- $k_1$  is the slope or gradient of  $y(x)$  estimated using the first order Euler method.

- $k_2$  is the slope of  $y(x + h/2)$  using the leap-frog method.

### 5.7.2 Error

To calculate the error of the modified Euler method, we again turn to one of our favourite tools; the Taylor series. Expanding the Taylor series and simplifying successive terms is a little more involved than it was for previous methods, so instead, we will walk through the general steps.

- When we consider numerical integration, we will see another way of deriving the local error term — as well as an explanation of where the modified Euler method gets its (midpoint) nickname from!

Subtracting the modified Euler method from  $y(x + h)$ ,

$$\begin{aligned} y(x + h) - y_{ME}(x + h) \\ = y(x + h) - \left[ y(x) + hf \left( y(x) + \frac{1}{2}hf(y(x), x), x + \frac{1}{2}h \right) \right], \end{aligned}$$

and taking the Taylor series expansion to order  $\mathcal{O}(h^4)$ , gives

$$\begin{aligned} y(x + h) - y_{ME}(x + h) \\ = h(y'(x) - f(y(x), x)) + \frac{1}{2}h^2 \left( y''(x) - y'(x) \frac{\partial f}{\partial y} - \frac{\partial f}{\partial x} \right) + \\ \frac{1}{24}h^3 \left( 4y'''(x) - 3y''(x) \frac{\partial f}{\partial y} - 3y'(x)^2 \frac{\partial^2 f}{\partial y^2} - 6y'(x) \frac{\partial^2 f}{\partial x \partial y} - 3 \frac{\partial^2 f}{\partial x^2} \right) + \mathcal{O}(h^4). \end{aligned}$$

Now, applying the chain rule successively to the original differential equation, we know that

$$\begin{aligned} y'(x) &= f(y(x), x) \\ y''(x) &= \frac{d}{dx}f(y(x), x) = y'(x) \frac{\partial f}{\partial y} + \frac{\partial f}{\partial x} = f(y(x), x) \frac{\partial f}{\partial y} + \frac{\partial f}{\partial x}. \\ y'''(x) &= \frac{d^2}{dx^2}f(y(x), x) \end{aligned}$$

Substituting these back into the Taylor series expansion and simplifying, all terms cancel except the  $h^3y'''(x)$  term, or equivalently  $y'''(x)\Delta x^3$  in the discrete grid interpretation. The modified Euler method therefore is a **second-order** method (like the leap-frog method). In fact, the **local error** term of each step of the modified Euler method is given by

$$LE = \frac{1}{24}\Delta x^3 \frac{d^2}{dx^2}f(y_n, x_n). \quad (5.53)$$

Multiplying the local truncation error by  $N = (x_{N-1} - x_0)/\Delta x$  then gives us an upper bound on the absolute value of the **global error**:

$|E| \leq \frac{x_{N-1} - x_0}{24} \Delta x^2 \max_{x \in [x_0, x_{N-1}]} \left| \frac{d^2}{dx^2}f(y(x), x) \right|. \quad (5.54)$

### 5.7.3 Stability

To get an idea of the stability, we want to perturb  $y_n$  by  $\delta y_n \ll 1$  such that  $y_n \rightarrow y_n + \delta y_n$ . Substituting this into Eq. 5.52, and taking the Taylor series expansion to first order in  $\delta y_n$ , we find that

$$\delta y_{n+1} = g\delta y_n = \left[ \frac{1}{2}\Delta x^2 \left( \frac{\partial f}{\partial y} \right)^2 + \Delta x \frac{\partial f}{\partial y} + 1 \right] \delta y_n. \quad (5.55)$$

Requiring that  $|g| \leq 1$  for stability, we find that

---

|                      |                                                                                                                                                                                          |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\text{Re}(f_y) < 0$ | the system is <b>conditionally stable</b> .                                                                                                                                              |
|                      | the system is <b>unstable</b> in general                                                                                                                                                 |
| $\text{Re}(f_y) = 0$ | <i>However, as long as <math>\text{Im}(f_y)\Delta x &lt; 1</math>, the instability is so small that it can be ignored in practice. The system is ‘practically’ conditionally stable.</i> |
| $\text{Re}(f_y) > 0$ | the system is <b>unconditionally unstable</b> – there is <i>no</i> value of $\Delta x$ which allows for stability.                                                                       |

---

**Table 5.5** Modified Euler stability

As you can see, we inherit the best of both worlds from the forward Euler and the leap-frog stability – stability that is suited for both decay equations ( $y' = f_y = -\alpha$ ) and oscillating equations ( $y' = f_y = \pm i\omega$ ). Plotting the stability region  $|\frac{1}{2}z^2 + z + 1| \leq 1$ , we see that although we now have a second-order method that improves on the stability of both the forward Euler method *and* the leap-frog method, the modified Euler method is inferior to the first-order backwards Euler method.

### 5.7.4 General Runge–Kutta Methods

The two-step Runge–Kutta method can be written in a more general form:

$$y_{n+1} = y_n + \Delta x(b_1 k_1 + b_2 k_2) + \mathcal{O}(\Delta x^3), \quad (5.56)$$

with the **stages values** per time-step given by

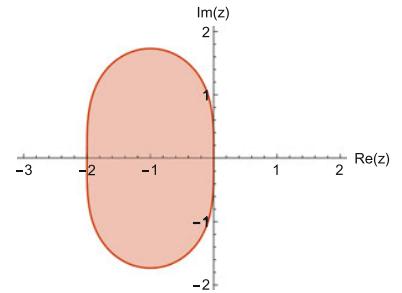
$$k_1 = f(y_n, x_n) \quad (5.57)$$

$$k_2 = f(y_n + a_{21}k_1\Delta x, x_n + c_2\Delta x). \quad (5.58)$$

In this general formulation,

- The elements  $a_{ij}$  form the **Runge–Kutta matrix**  $A$ , and in the case of explicit methods is *always* lower-triangular:

$$A = \begin{bmatrix} 0 & 0 \\ a_{21} & 0 \end{bmatrix},$$



**Figure 5.8** Modified midpoint method stability diagram

- $\mathbf{c} = (c_1, c_2)$  are the **Runge–Kutta nodes**, and for consistency each  $c_i$  must equal the sum of the  $i$ th row of  $A$ :

$$c_i = \sum_j a_{ij},$$

- $\mathbf{b} = (b_1, b_2)$  are the **Runge–Kutta weights**.

The Runge–Kutta matrix, nodes and weights are by convention written as a **Butcher tableau**:

$$\begin{array}{c|cc} \mathbf{c} & A \\ \hline \mathbf{b} & \end{array} \quad \text{or, for the second order case,} \quad \begin{array}{c|cc} 0 & 0 & 0 \\ \hline c_2 & a_{21} & 0 \\ \hline b_1 & b_2 \end{array}$$

### The Second-Order Runge–Kutta Order Conditions

Using Taylor series, let's try and solve Eq. 5.56 to find the values of  $a_{21}$ ,  $c_2$ ,  $b_1$ , and  $b_2$  that **minimises error**. To do this, we begin by taking the Taylor series expansion of  $k_2$  around  $\Delta x$ :

$$k_2 = f(y, x) + \Delta x \left( a_{21} k_1 \frac{\partial f}{\partial x} + c_2 \frac{\partial f}{\partial y} \right) + \mathcal{O}(\Delta x^2). \quad (5.59)$$

Substituting this, as well as  $k_1 = f(y, x)$  into the Runge–Kutta second-order approximation for  $y_{n+1} \equiv y(x + \Delta x)$ :

$$\begin{aligned} y(x + \Delta x) &= y(x) + f(y, x) \Delta x (b_1 + b_2) \\ &\quad + b_2 \Delta x^2 \left( a_{21} f(y, x) \frac{\partial f}{\partial x} + c_2 \frac{\partial f}{\partial y} \right) + \mathcal{O}(\Delta x^3). \end{aligned} \quad (5.60)$$

We can compare this to the Taylor series of  $y(x + \Delta x)$ . Recall that, from our initial value problem,

$$\Rightarrow y'(x) = f(y(x), x) \quad (5.61)$$

$$\Rightarrow y''(x) = y'(x) \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} = f(y, x) \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}, \quad (5.62)$$

thus

$$\begin{aligned} y(x + \Delta x) &= y(x) + y'(x) \Delta x + \frac{1}{2} y''(x) \Delta x^2 + \mathcal{O}(\Delta x^3) \\ &= y(x) + f(y, x) \Delta x + \frac{1}{2} \Delta x^2 \left( f(y, x) \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} \right) + \mathcal{O}(\Delta x^3). \end{aligned}$$

Compare this to Eq. 5.60 – you can see that for each term to be equal, we require

$$\begin{cases} b_1 + b_2 = 1 \\ b_2 a_{21} = \frac{1}{2} \\ b_2 c_2 = \frac{1}{2} \end{cases}. \quad (5.63)$$

Thus, any two-step Runge–Kutta method satisfying these conditions will be **second-order**, and have a local error term of order  $\Delta x^3$ .

The modified midpoint Euler method we derived in the previous section satisfies the Runge–Kutta conditions, with  $b_1 = 0$ ,  $b_2 = 1$ ,  $a_{21} = 1/2$ , and  $c_2 = 1/2$ ; its Butcher tableau is

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline 0 & 1 \end{array} \quad (5.64)$$

Another second-order Runge–Kutta method commonly seen, sometimes referred to as the **modified endpoint Euler method** or **Heun’s method**, has the Butcher tableau

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{array} \quad (5.65)$$

### 5.7.5 General $N$ -Step Explicit Runge–Kutta

The general notation used above can be extended to  $N$ -stages per time-step:

$$y_{n+1} = y_n + \Delta x \sum_{j=1}^N b_j k_j + \mathcal{O}(\Delta x^{N+1})$$

$$k_j = f \left( y_n + \Delta x \sum_{i=1}^{j-1} a_{ji} k_i, x_n + c_j \Delta x \right).$$

**Order conditions** such that an  $N$ -stage Runge–Kutta has order  $P$  can be calculated for  $N \leq 4$  but *not* for  $N \geq 5$ .

## 5.8 Implicit Runge–Kutta Methods

Recall that one of the reasons behind the high stability of the backwards Euler method is the fact that it is an implicit method — to calculate  $y_{n+1}$ , we need to know  $f(y_{n+1}, x_{n+1})$  — requiring the use of a root finding algorithm.

Let’s attempt to do the same thing with the modified midpoint Euler method. Again, we’ll consider the first order initial value problem,

$$\frac{dy}{dx} = f(y(x), x), \quad y(x_0) = y_0 \quad (5.66)$$

To start with, let’s approximate the left hand side simply using the forward Euler approximation. This time, however, rather than evaluate  $f(y(x), x)$  at the midpoint of the interval, let’s instead *average* the function across the entire interval:

$$\frac{y(x+h) - y(x)}{h} \approx \frac{1}{2} [f(y(x), x) + f(y(x+h), x+h)]. \quad (5.67)$$

▶ Letting  $c_2 = x$ , the Butcher tableau of the general second-order Runge–Kutta method can be written

$$\begin{array}{c|cc} 0 & 0 & 0 \\ x & x & 0 \\ \hline 1 - \frac{1}{2x} & \frac{1}{2x} & \frac{1}{2x} \end{array}$$

Rearranging, and using an equally spaced discrete grid of  $N$  points  $x_0, x_1, \dots, x_{N-1}$ , where  $\Delta x = x_{n+1} - x_n \ll 1$  and  $y_n = y(x_n)$ , we have the **second-order trapezoidal method**,

$$y_{n+1} = y_n + \frac{1}{2}\Delta x [f(y_n, x_n) + f(y_{n+1}, x_{n+1})] \quad (5.68)$$

As calculating  $y_{n+1}$  depends on knowing the value of the function  $f(y_{n+1}, x_{n+1})$ , the trapezoid method is an **implicit** method, like the backwards Euler method.

**Aside**

We can also write it in the form of a two-step method;

$$\begin{aligned} y_{n+1} &= y_n + \frac{1}{2}\Delta x(k_1 + k_2) + \mathcal{O}(\Delta x^3) \\ k_1 &= f(y_n, x_n) \\ k_2 &= f\left(y_n + \frac{1}{2}\Delta x(k_1 + k_2), x_n + \Delta x\right) \end{aligned} \quad (5.69)$$

The trapezoidal method equation, while short and seemingly simple on the surface, can then seem quite intimidating as soon as you need to actually use it to solve a differential equation. How are you meant to calculate  $y_{n+1}$ ? Just like we did with the backwards Euler method, we can rewrite the trapezoidal method in the form

$$g(y_{n+1}) = y_n + \frac{1}{2}\Delta x [f(y_n, x_n) + f(y_{n+1}, x_{n+1})] - y_{n+1} = 0, \quad (5.70)$$

and use a root finding algorithm such as the Newton–Raphson method to solve for  $y_{n+1}$ ,

$$y_{n+1}^{(j+1)} = y_{n+1}^{(j)} + \frac{g(y_{n+1}^{(j)})}{g'(y_{n+1}^{(j)})}, \quad (5.71)$$

and assuming that for  $\Delta x \ll 1$ ,  $y_{n+1}^{(0)} \approx y_n$  is a reasonable first estimate for  $y_{n+1}$ .

‘Now hang on a minute!’ we hear you say. ‘You derived the trapezoidal method using the first order Euler method! And just *averaged over the interval*?! No way this method is second order!’. Good point. Time to whip out those Taylor series and perform another error analysis! Subtracting the trapezoidal method from  $y(x+h)$  and taking the Taylor series expansion to order  $\mathcal{O}(h^4)$ , gives

$$\begin{aligned} &y(x+h) - y_{TM}(x+h) \\ &= h(y'(x) - f(y(x), x)) + \frac{1}{2}h^2 \left( y''(x) - y'(x)\frac{\partial f}{\partial y} - \frac{\partial f}{\partial x} \right) \\ &\quad - \frac{1}{12}h^3 \left( 2y'''(x) - 3y''(x)\frac{\partial f}{\partial y} - 3y'(x)^2\frac{\partial^2 f}{\partial y^2} - 6y'(x)\frac{\partial^2 f}{\partial x \partial y} - 3\frac{\partial^2 f}{\partial x^2} \right) + \mathcal{O}(h^4). \end{aligned}$$

The  $h$  and  $h^2$  term are identical to the error terms for the midpoint Euler method, so we know they cancel out. The  $h^3$  is also identical, just with a coefficient of  $-1/12$  rather than  $1/24$ . Therefore, the trapezoidal method is definitely a **second-order** method (despite how we derived it!). The **local error** term of each step of the trapezoidal method is given by

$$LE = -\frac{1}{12}\Delta x^3 \frac{d^2}{dx^2} f(y_n, x_n). \quad (5.72)$$

Multiplying the local truncation error by  $N = (x_{N-1} - x_0)/\Delta x$  then gives us an upper bound on the absolute value of the **global error**:

$$|E| \leq \frac{x_{N-1} - x_0}{12} \Delta x^2 \max_{x \in [x_0, x_{N-1}]} \left| \frac{d^2}{dx^2} f(y(x), x) \right|. \quad (5.73)$$

So, the trapezoidal method, like the modified Euler method, has a global error that scales with  $\Delta x^2$ ; however the trapezoidal error is approximately double that of the modified Euler method. So why use the trapezoidal method at all, when we could avoid using a root-finding algorithm, and just use an explicit Runge–Kutta method?

To get the full story, let's do a quick stability analysis. As always, we perturb  $y_n$  by  $\delta y_n \ll 1$  such that  $y_n \rightarrow y_n + \delta y_n$ , and  $y_{n+1} \rightarrow \delta y_{n+1}$ . Substituting this into Eq. 5.68, and taking the Taylor series expansion to first order in  $\delta y_n$ , we find that:

$$\delta y_{n+1} = g\delta y_n = \left( \frac{1 + \Delta x f_y/2}{1 - \Delta x f_y/2} \right) \delta y_n. \quad (5.74)$$

Requiring that  $|g| \leq 1$  for stability, we find that

---

|                         |                                                                                                               |
|-------------------------|---------------------------------------------------------------------------------------------------------------|
| $\text{Re}(f_y) \leq 0$ | the system is <b>unconditionally stable</b>                                                                   |
| $\text{Re}(f_y) > 0$    | the system is <b>unconditionally unstable</b> – there is<br>no value of $\Delta x$ which allows for stability |

---

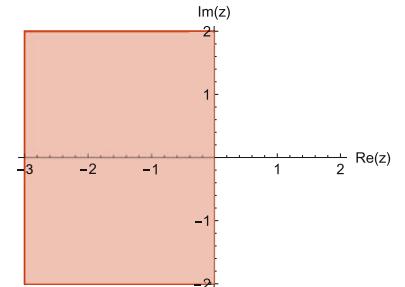
**Table 5.6** Trapezoidal stability

Plotting the region of stability, we can see that the implicit Trapezoidal method displays stability for a significantly higher number of potential ODEs; Not only that, but since the entire negative real plane lies within the region of stability, the Trapezoidal method is **A-stable**!

### 5.8.1 General Runge–Kutta Methods

As in the explicit case, the two-step implicit Runge–Kutta method can be written in the following, general, form:

$$\begin{aligned} y_{n+1} &= y_n + \Delta x(b_1 k_1 + b_2 k_2) + \mathcal{O}(\Delta x^3) \\ k_1 &= f(y_n + \Delta x(k_1 a_{11} + k_2 a_{12}), x_n + \Delta x c_1) \\ k_2 &= f(y_n + \Delta x(k_1 a_{21} + k_2 a_{22}), x_n + \Delta x c_2). \end{aligned} \quad (5.75)$$



**Figure 5.9** Trapezoid method stability diagram

- ▶ However, unlike the backwards Euler method, since  $\lim_{\Delta x \rightarrow \infty} |g| = 1 \neq 0$ , the trapezoidal method is not L-stable.

- ▶ Since the explicit Runge–Kutta method is a special case of the implicit Runge–Kutta method, we can consider the implicit Runge–Kutta method as the general Runge–Kutta method.

The Runge–Kutta matrix is no longer lower-triangular; this causes the stages to include components containing  $y > y_n$ , and is the mechanism behind their implicit nature. As before, we can represent implicit Runge–Kutta methods via their Butcher tableaux — for example, the Trapezoidal method has a Butcher tableau

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

To solve the two-step implicit Runge–Kutta order conditions such that a second order method is guaranteed, we take a similar approach to when we solved the order conditions for the explicit case. First, take the Taylor series of  $k_1$  and  $k_2$  to second order in  $\Delta x$ , solve for  $k_1$  and  $k_2$  and substitute them into each other, then into the general implicit Runge–Kutta equation. Finally, we solve for the coefficients  $\mathbf{A}$ ,  $\mathbf{b}$ , and  $\mathbf{c}$ ; such that it agrees with the Taylor series of  $y(x + \Delta x)$  up to second order. This is a much more involved process than in the explicit case, and results in many more degrees of freedom; nevertheless, after some lengthy algebra, you should find that

$$\begin{cases} b_1 + b_2 = 1 \\ b_1 c_1 + b_2 c_2 = \frac{1}{2} \\ a_{22} = a_{11} \left(1 - \frac{1}{b_2}\right) + a_{12} \left(1 - \frac{1}{b_2}\right) - a_{21} + \frac{1}{b_2} \end{cases}. \quad (5.76)$$

Note that, coupled with the node requirement that  $c_i = \sum_j a_{ij}$ , we have 5 equations with 8 variables; therefore, we have **3 free parameters**. If we choose as the free parameters  $a_{11} = a_{12} = a_{22} = 0$ , we simply recover the explicit Runge–Kutta second order conditions.

### 5.8.2 General $N$ -Step Implicit Runge–Kutta

The general notation used above can be extended to  $N$ -stages per time-step:

$$y_{n+1} = y_n + \Delta x \sum_{j=1}^N b_j k_j + \mathcal{O}(\Delta x^{N+1})$$

$$k_j = f \left( y_n + \Delta x \sum_{i=1}^N a_{ji} k_i, x_n + c_j \Delta x \right).$$

Note that, unlike in the explicit case, the sum over  $a_{ji} k_i$  now goes up to  $i = N$ , and not  $i = j-1$ , causing the implicit nature of the method. Order conditions such that an  $N$ -stage Runge–Kutta has order  $P$  can generally be calculated for larger  $N$  than the explicit case. However, it becomes increasingly difficult to solve the required systems of equations.

### 5.8.3 Runge–Kutta Stability

If we were to go through the process of perturbing the general Runge–Kutta equation Eq. 5.77 by  $\delta y_n$ , and then taking a very large Taylor series, we would find that

$$\delta y_{n+1} = [1 + f_y \Delta x (I - Af_y \Delta x)^{-1} \mathbf{b}] \cdot \mathbf{1} \delta y_n, \quad (5.77)$$

where  $\mathbf{1}$  is the all-ones vector. Thus, letting  $z = f_y \Delta x$ , the stability function of the Runge–Kutta method is

$$R(z) = [1 + z(I - Az)^{-1} \mathbf{b}] \cdot \mathbf{1} = \frac{|I - zA + z\mathbf{1}\mathbf{b}^T|}{|I - zA|}. \quad (5.78)$$

This may appear slightly intimidating, but allows us to extract some important results regarding Runge–Kutta stability. For example, consider the denominator — if the Runge–Kutta method is explicit, then  $A$  is strictly lower triangular with diagonals of zero, and  $I - zA$  will be lower triangular with constant value 1 along the diagonal. We know from linear algebra that the determinant of a triangular matrix is simply given by the product of the diagonals, and so

$$|I - zA| = \prod_i \delta_{ii} = 1. \quad (5.79)$$

Therefore, if the Runge–Kutta method is explicit, then the stability function is polynomial in  $z$ , and **cannot be A-stable** (since it is impossible for all  $z$  such that  $\text{Re}(z) < 1$  lies in the region of stability  $|R(z)| \leq 1$ ).

This fully encapsulates the power behind implicit Runge–Kutta stability — we have the ability to construct implicit Runge–Kutta methods of *any* order that exhibit A-stability. However, explicit Runge–Kutta methods still have their place in numerical ODE techniques, and in fact, some explicit Runge–Kutta methods are very nearly A-stable.

► Now you see the advantage of representing the Runge–Kutta coefficients as matrices and vectors!

## 5.9 RK4: The Fourth-Order Runge Kutta Method

- ▶ The original, you might say!
- ▶ Alternatively, Eq. 5.80 can be written in Butcher tableau form:

|               |               |               |               |               |
|---------------|---------------|---------------|---------------|---------------|
| 0             | 0             | 0             | 0             | 0             |
| $\frac{1}{2}$ | $\frac{1}{2}$ | 0             | 0             | 0             |
| $\frac{1}{2}$ | 0             | $\frac{1}{2}$ | 0             | 0             |
| 1             | 0             | 0             | 1             | 0             |
|               | $\frac{1}{6}$ | $\frac{1}{3}$ | $\frac{1}{3}$ | $\frac{1}{6}$ |

$$y_{n+1} = y_n + \frac{1}{6}\Delta x(k_1 + 2k_2 + 2k_3 + k_4) + \mathcal{O}(\Delta x^5)$$

where

$$\begin{aligned} k_1 &= f(y_n, x_n) \\ k_2 &= f\left(y_n + \frac{1}{2}k_1\Delta x, x_n + \frac{1}{2}\Delta x\right) \\ k_3 &= f\left(y_n + \frac{1}{2}k_2\Delta x, x_n + \frac{1}{2}\Delta x\right) \\ k_4 &= f(y_n + k_3\Delta x, x_n + \Delta x) \end{aligned} \quad (5.80)$$

Calculating the error of the RK4 method is slightly more difficult than it is in the second order case. Nevertheless, trawling through a similar Taylor series expansion as previously, we find that the **local error** in RK4 is given by:

$$LE = -\frac{1}{2880}\Delta x^5 \frac{d^4}{dx^4}f(y(x), x). \quad (5.81)$$

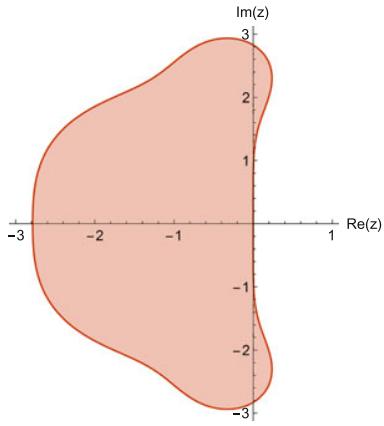
An estimated upper bound for the **global error** is therefore

$$|E| \leq \frac{x_{N-1} - x_0}{2880} \Delta x^4 \max_{x \in \{x_0, x_{N-1}\}} \left| \frac{d^4}{dx^4}f(y(x), x) \right|. \quad (5.82)$$

Using the Runge–Kutta stability function Eq. 5.78, the stability *polynomial* for the RK4 method is

$$R(z) = |I - zA + z\mathbf{1}\mathbf{b}^T| = 1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24}, \quad (5.83)$$

where  $z = f_y\Delta x$ . Plotting the stability region  $|R(z)| \leq 1$ , we see that the Runge–Kutta method, while not A-stable, incorporates a reasonable region of the complex plane (much more than that of the forward Euler method, and *significantly* more than the leap-frog method!). Coupled with its fourth order global error, and the fact that it is the largest possible explicit  $N$ -step  $N$ th order Runge–Kutta method, you can see why the RK4 method has remained so popular.



**Figure 5.10** RK4 method stability diagram

**Example 5.8** Implementing RK4 (Fortran)

Use the forward Euler method to solve the initial value problem  $y'(x) = xy(x)$ ,  $y(0) = 2$  between  $0 \leq x \leq 1$

**Solution:** Look familiar? This is identical to example 5.2, but back then we used the forward Euler method. Let's return to it, and solve the ODE, this time using the RK4 method.

As before, let's choose  $\Delta x = 0.01$ , resulting in a grid size of  $N = (1 - 0)/\Delta x + 1 = 101$ . Writing a Fortran program to implement the RK4 method:

```
program RK4
    implicit none

    integer :: i, N
    real :: xmin, xmax, dx, k1, k2, k3, k4
    real, allocatable :: x(:), y(:)

    ! set the x-grid size and spacing
    dx = 0.01; xmin = 0.; xmax = 1.
    N = (xmax-xmin)/dx + 1

    ! allocate the x and y arrays
    allocate(x(1:N), y(1:N))
    x = [(xmin + dx*i, i=0,N-1)]
    y = 0.
    y(1) = 2.

    ! the RK4 4-step method
    do i=1,N-1
        k1 = f(y(i), x(i))
        k2 = f(y(i) + 0.5*k1*dx, x(i) + 0.5*dx)
        k3 = f(y(i) + 0.5*k2*dx, x(i) + 0.5*dx)
        k4 = f(y(i) + k3*dx, x(i) + dx)

        y(i+1) = y(i) + dx*(k1 + 2*k2 + 2*k3 + k4)/6.
    end do

    contains
        function f(y,x)
            real, intent(in) :: x, y
            real :: f

            f = x*y
        end function
end program RK4
```

In this example, we are using arrays to store all calculated values of  $x$  and  $y$ , and using an internal function for easy implementation of  $f(y(x), x)$ .

**Example 5.9** Implementing RK4 (Python)

Use the forward Euler method to solve the initial value problem  $y'(x) = xy(x)$ ,  $y(0) = 2$  between  $0 \leq x \leq 1$

**Solution:**

Look familiar? This is identical to example 5.2, but back then we used the forward Euler method. Let's return to it, and solve the ODE, this time using the RK4 method.

As before, let's choose  $\Delta x = 0.01$ , resulting in a grid size of  $N = (1 - 0)/\Delta x + 1 = 101$ . Writing a Python program to implement the RK4 method:

```
#!/usr/bin/env python3
import numpy as np

# define the ODE
def f(y,x):
    return x*y

# set x-grid size and spacing
dx = 0.01
x = np.arange(0, 1+dx, dx)
N = len(x)

# create the y-grid and initial conditions
y = np.zeros((N))
y[0] = 2

# the RK4 method
for i in range(0,N-1):
    k1 = f(y[i], x[i])
    k2 = f(y[i] + k1*dx/2, x[i] + dx/2)
    k3 = f(y[i] + k2*dx/2, x[i] + dx/2)
    k4 = f(y[i] + k3*dx, x[i] + dx)

    y[i+1] = y[i] + dx*(k1 + 2*k2 + 2*k3 + k4)/6
```

In this example, we are using NumPy arrays to store all calculated values of  $x$  and  $y$ , and using an user-defined function for easy implementation of  $f(y(x), x)$ .

### 5.9.1 Second Order ODEs

The RK4 method can also easily be adapted to second-order ODEs (as can any of the other methods we have focused on in this chapter). To do so, consider an arbitrary second order ODE, of the form

$$y''(x) = f(y'(x), y(x), x), \quad y(0) = y_0, \quad y'(0) = z_0 \quad (5.84)$$

can be written as a **coupled system of first order ODEs**:

$$\begin{cases} y'(x) = z(x) \\ z'(x) = f(z(x), y(x), x) \end{cases} . \quad (5.85)$$

By interpreting the second order ODE in this manner, the 4th-order Runge–Kutta method for the coupled ODE becomes:

$$y_{n+1} = y_n + \frac{1}{6}\Delta x(k_1 + 2k_2 + 2k_3 + k_4) \quad (5.86a)$$

$$z_{n+1} = z_n + \frac{1}{6}\Delta x(l_1 + 2l_2 + 2l_3 + l_4), \quad (5.86b)$$

where the Runge–Kutta stages are

$$k_1 = z_n \quad l_1 = f(z_n, y_n, x_n) \quad (5.87a)$$

$$k_2 = z_n + \frac{1}{2}l_1\Delta x \quad l_2 = f\left(z_n + \frac{1}{2}l_1\Delta x, y_n + \frac{1}{2}k_1\Delta x, x_n + \frac{1}{2}\Delta x\right) \quad (5.87b)$$

$$k_3 = z_n + \frac{1}{2}l_2\Delta x \quad l_3 = f\left(z_n + \frac{1}{2}l_2\Delta x, y_n + \frac{1}{2}k_2\Delta x, x_n + \frac{1}{2}\Delta x\right) \quad (5.87c)$$

$$k_4 = z_n + l_3\Delta x \quad l_2 = f(z_n + l_3\Delta x, y_n + k_3\Delta x, x_n + \Delta x). \quad (5.87d)$$

At each step, we now apply the Runge–Kutta stages to **both** of the coupled ODEs, with  $k_i$  being the stage values for  $y'(x) = z(x)$ , and  $l_i$  being the stage values of  $z'(x) = f(z(x), y(x), x)$ . Note that the  $k_i$  and  $l_i$  stage values depend on each other, due to the fact of the ODEs being coupled — this determines the order in which they must be calculated (i.e. in this case, we need to calculate  $l_1$  in order to calculate  $k_2$ ).

Alternatively, we can rewrite this system of coupled ODEs using vector notation:

$$\mathbf{r}''(\mathbf{x}) = \mathbf{f}(\mathbf{r}(x), x), \quad \mathbf{r}(0) = (y_0, z_0), \quad (5.88)$$

where  $\mathbf{r}(x) = (y(x), z(x))$  and  $\mathbf{f}(\mathbf{r}(x), x) = (z(x), f(z(x), y(x), x))$ . This allows us to write the 4th-order Runge–Kutta method using notation that easily scales beyond simply two coupled ODEs, and can easily be extended to solve third order and higher ODEs. In this notation, discretising the system such that  $\mathbf{r}(x_n) \rightarrow \mathbf{r}_n$ , the only difference now is that  $\mathbf{k}_i$  become **stage vectors**:

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \frac{1}{6}\Delta x(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \quad (5.89a)$$

$$\mathbf{k}_1 = \mathbf{f}(\mathbf{r}_n, x_n) \quad (5.89b)$$

$$\mathbf{k}_2 = \mathbf{f}\left(\mathbf{r}_n + \frac{1}{2}\mathbf{k}_1\Delta x, x_n + \frac{1}{2}\Delta x\right) \quad (5.89c)$$

$$\mathbf{k}_3 = \mathbf{f}\left(\mathbf{r}_n + \frac{1}{2}\mathbf{k}_2\Delta x, x_n + \frac{1}{2}\Delta x\right) \quad (5.89d)$$

$$\mathbf{k}_4 = \mathbf{f}(\mathbf{r}_n + \mathbf{k}_3\Delta x, x_n + \Delta x). \quad (5.89e)$$

This ‘vectorisation’ of the Runge–Kutta method can be generalised to *any* of the methods we covered in this chapter, in order to appropriate them for coupled or higher order ODEs.

### Reusing the wheel

If using Python, external modules provide some extremely useful tools and pre-written algorithms for calculating derivatives and solving ODEs. Before you sigh and promptly forget the preceding chapter, knowing the ins and outs of how numerical differentiation algorithms work is very important, and will give you an understanding of *when* and *how* to use each algorithm to reduce the chances of instability and error.

On the other hand, this doesn’t mean you have to hand code these algorithms every time — often, you’ll find that well-established libraries (such as NumPy and SciPy for Python, and ODEPACK for Fortran) have highly optimised implementations built-in.

For example, the NumPy method `numpy.diff` calculates the  $n$ th discrete difference between array elements along a specific axis, while `numpy.gradient` applies the second-order central finite difference approximation to calculate the derivative of an array. For example, to calculate the derivative of an array representing  $x^2$ :

```
>>> dx = 0.1
>>> x = np.arange(0, 1+dx, dx)
>>> y = x**2
>>> np.gradient(y, dx)
array([ 0.1,  0.2,  0.4,  0.6,  0.8,  1.,
       1.2,  1.4,  1.6,  1.8,  1.9])
```

SciPy provides even more advanced methods; `scipy.misc.derivative` calculates the  $n$ th derivative using the central difference formula of a *function at a point*, avoiding needing to create a discrete array at all! Finally, `scipy.integrate.solve_ivp` contains a range of different ODE solvers (including explicit and implicit Runge–Kutta methods), and simply accepts the function  $f(y(x), x)$  and initial condition  $y_0$  as arguments.

### Further reading

It may have seemed like we covered a lot in this chapter, but you could fill whole textbooks on the study of numerical ODE techniques — and whole books *have* been filled! Aside from going into more depth on topics such as error analysis and stability, they also touch on areas of analysis we haven’t been able to cover, such as convergence.

The amount of work done studying Runge–Kutta methods is immense, and every year numerous papers get published analysing or presenting new solutions to order conditions or classes of Runge–Kutta methods. One class of Runge–Kutta methods we haven’t touched on are **adaptive Runge–Kutta methods**. These methods include *two* sets of stages at each time-step, one of order  $N$ , and one of order  $N - 1$  that is used to approximate the local error, and adapt the step size dynamically to compensate. An example of this is the Runge–Kutta–Fehlberg method, commonly known as RK45 — the creativity of this method is that *both* sets of stages are designed to use the same function calls, avoiding additional computational overhead.

In addition, there are a huge number of numerical algorithms outside the Runge–Kutta methods, including the class of **linear multi-step methods** (as opposed to the Runge–Kutta, which is a *single*-step method with multiple stages).

For all this (and more!), here are some textbooks for further reading:

- Butcher, John C. (2008), Numerical Methods for Ordinary Differential Equations, New York: John Wiley & Sons, ISBN 978-0-470-72335-7.
- Iserles, Arieh (1996), A First Course in the Numerical Analysis of Differential Equations, Cambridge University Press, ISBN 978-0-521-55655-2.
- Lambert, J.D (1991), Numerical Methods for Ordinary Differential Systems. The Initial Value Problem, John Wiley & Sons, ISBN 0-471-92990-5

## Exercises

- P5.1** Consider the following code, which uses the Euler method to compute the trajectory of a bouncing ball, assuming perfect reflection at the surface  $x = 0$ :

Fortran:

```
program bouncing_balls
    implicit none

    integer :: steps
    real     :: x, v, g, t, dt

    x = 1.0 ! initial height of the ball
    v = 0.0 ! initial velocity of the ball
    g = 9.8 ! gravitational acceleration
    t = 0.0 ! initial time
    dt = 0.01 ! size of time step

    open(10,file='bounce.dat') ! open data file

    do steps = 1, 300 ! loop for 300 timesteps
        t = t + dt
        x = x + v*dt
        v = v - g*dt

        ! reflect the motion of the ball
        ! when it hits the surface x=0
        if(x.lt.0) then
            x = -x
            v = -v
        endif

        ! write out data at each time step
        write(10,*) t, x, v
    end do
end program bouncing_balls
```

Python:

```
#!/usr/bin/env python3
```

```

x = 1.0      # initial height of the ball
v = 0        # initial velocity of the ball
g = 9.8      # gravitational acceleration
t = 0        # initial time
dt = 0.01    # size of time step

# loop for 300 timesteps
for steps in range(300):
    t = t + dt
    x = x + v*dt
    v = v - g*dt

    # reflect the motion of the ball
    # when it hits the surface x=0
    if x < 0:
        x = -x
        v = -v

    # write out data at each time step
    with open("bounce.dat", "a+") as f:
        f.write("{} {} {}\n".format(t, x, v))

```

- (a) Compile and run the program, and then plot and interpret the output. Are your numerical results physically correct? If not, can you identify a **systematic error** in the algorithm, and then fix the problem?
- (b) Change the time step `dt` in the code, but keep the same total evolution time. Explain the changes in the results.
- (c) Change the initial velocity and position of the falling ball in the code. Plot and interpret your results.
- (d) Consider inelastic collisions with the table (e.g. the ball loses 10% of its speed after every collision). Plot and interpret your results.

**P5.2** The force on a particle due to a magnetic field is given by

$$\mathbf{F} = q\mathbf{v} \times \mathbf{B}$$

If a particle with a charge  $q$  and mass  $m$  is moving in the  $x$ - $y$  plane with magnetic field  $\mathbf{B} = (0, 0, b)$  in the positive  $z$ -direction, the motion of the electron is confined to the  $x$ - $y$  plane.

- (a) Solve the Newtonian equations of motion to determine the velocity and position of the particle as a function of  $t$ .

- (b) Use the Euler method to solve the Newtonian equations of motion numerically, and determine approximate functions for the position and velocity of the particle.
- (c) Write a Fortran or Python program to model the motion of the particle, reading the initial position and velocity in the  $x$ - $y$  plane, propagation time  $t$ , time-step  $dt$ , mass  $m$  and magnetic field strength  $b$  from an input file.

The program should output to a file the position and velocity of the particle over time, allowing for straightforward plotting of the particles motion.

- (d) Plot your results, and compare them to your analytic results from part 1.

What are the most likely sources of error in your algorithm and/or program? How could you reduce or eliminate these sources of error?

- P5.3** Write your own Fortran or Python code to obtain numerical solutions to the atomic decay equation (see Examples 5.6 and 5.7), and discuss and analyze convergence and numerical errors.

For large values of  $\Delta t$  and  $t$ , why does the first-order Euler method appear more accurate than the second-order leap-frog method? What happens when you decrease  $t$ ?

Note: round-off errors are often processor and compile dependent.

- P5.4** Consider the differential equation

$$y'(x) = 1 + 2xy(x), \quad y(0) = 0$$

- (a) In Fortran or Python, find the numerical solution for  $0 \leq x \leq 1$ , using the second order Runge–Kutta modified Euler algorithm.

The exact solution to this differential equation is given by

$$y(x) = \frac{1}{2}\sqrt{\pi}e^{x^2} \operatorname{erf}(x)$$

where  $\operatorname{erf}(x)$  is the error function.

- (c) Calculate the numeric error in your solution for various values of  $\Delta x$ , and plot how the error scales with  $\Delta x$ .
- (i) How does this compare to the Euler method?
  - (ii) How does this compare to the leap-frog method?
- (d) Now solve the differential equation using the fourth-order Runge–Kutta method. Analyse your results. How does the error scaling compare to the second order Runge–Kutta method?

► In Fortran, the error function can be evaluated by using the `real` intrinsic function `erf(x)`

► In Python, you can import the error function from the standard math library,  
`from math import erf`  
 or from the SciPy library,  
`from scipy.special import erf`

- P5.5** Some of the numerical ODE solving methods we looked at the beginning of this chapter can also be written in Butcher tableau form. Rewrite the following methods in general Runge–Kutta form, and thus find their Butcher tableaux:

- (a) The forward Euler method
- (b) The backwards Euler method

Note: both of these are first order methods, so the Runge–Kutta matrix  $A$  will be  $1 \times 1$ . Since the forward Euler method is an *explicit* method,  $A$  must be lower-triangular!

- P5.6** Just as there are several solutions to the general second-order Runge–Kutta equations, there are also several solutions to the fourth-order equations. One such solution is known as the 3/8-rule method, and has the following Butcher tableau:

|               |                |               |               |               |
|---------------|----------------|---------------|---------------|---------------|
| 0             | 0              | 0             | 0             | 0             |
| $\frac{1}{3}$ | $\frac{1}{3}$  | 0             | 0             | 0             |
| $\frac{2}{3}$ | $-\frac{1}{3}$ | 1             | 0             | 0             |
| 1             | 1              | -1            | 1             | 0             |
|               | $\frac{1}{8}$  | $\frac{3}{8}$ | $\frac{3}{8}$ | $\frac{1}{8}$ |

- (a) For the general first-order initial value problem

$$y'(x) = f(y, x), \quad y(0) = y_0,$$

write out the discretized solution for  $y_{n+1}$  using the 3/8-rule.

- (b) Using Fortran or Python to implement the Runge–Kutta 3/8-rule method, solve the differential equation

$$y'(x) = -2xy(x)^2, \quad y(0) = 1$$

- (c) Solve the differential equation analytically, and use this to calculate the error in your method for various values of  $\Delta x$ . How does this compare to the classic RK4 method?
- (d) Use the Runge–Kutta stability equation to calculate the stability function for the 3/8-rule method. Plot the stability region in the complex plane — how does it compare to the classic RK4 method?

### P5.7 Terminal velocity

Consider a poncho dropped in free-fall, with non-negligible air-resistance. From Newton’s equations, the force felt by the object is

$$F = m \frac{d^2 r}{dt^2} = Dv^2 - g \tag{5.90}$$

where  $r$  is the height of the poncho,  $D$  is the drag coefficient,  $m = 1\text{kg}$  the mass of the poncho (it's a very heavy poncho), and  $g$  is the acceleration due to gravity.

Since the first derivative of displacement is velocity, this can be written as a pair of coupled first order differential equations:

$$\begin{cases} r'(t) = v(t) \\ v'(t) = \frac{D}{m}v(t)^2 - g \end{cases} \quad (5.91)$$

or, in vector notation, as

$$\mathbf{y}'(t) = \mathbf{f}(\mathbf{y}) \quad (5.92)$$

where

$$\mathbf{y}(t) = (r(t), v(t)) \quad \text{and} \quad \mathbf{f}(\mathbf{y}) = \left( v(t), \frac{D}{m}v(t)^2 - g \right) \quad (5.93)$$

The Runge–Kutta method easily generalises to systems of linear differential equations; since  $\mathbf{y}_n$  is now a vector, and  $\mathbf{f}$  is a vector-valued function, it follows that the stage values become **stage vectors**,  $\mathbf{k}_i$ .

- (a) The poncho starts off 100m above the ground with zero velocity,

$$\mathbf{y}(0) = (100, 0)$$

Letting  $D = 0.02\text{kg/m}$  and  $g = 9.81\text{m/s}$ , and choosing a reasonable number of intervals, use the fourth-order Runge–Kutta method to solve this differential equation for  $0 \leq t \leq 10$ .

- (b) Plot  $r(t)$  and  $v(t)$ .

- (i) When does the poncho hit the ground?
- (ii) What is the terminal velocity of the poncho?



## Chapter 6

# Numerical Integration

Over the previous 5 chapters, we have gradually built up the numerical tools we need in order to solve the Schrödinger equation in one dimension (finite-difference methods, root finding) as well as undertaking a crash course in all things Fortran and/or Python, depending on your programming language of choice. However, there is one tool we are missing. When it comes to normalising our discretized wavefunction solutions,

$$\langle \psi | \psi \rangle = \int_{-\infty}^{\infty} \psi(x)^* \psi(x) dx,$$

how exactly are we meant to compute the integral? If you haven't already guessed by the name of this chapter, the answer lies in the technique of numerical integration (surprise!).

Aside from calculating the integral of discretized functions, numerical integration can also be used to approximate integrals without analytical solutions, and in cases where the integrand (the function being integrated) itself does not have a known analytical form. In this chapter, we'll explore some common methods of numerical integration.

### 6.1 Trapezoidal Approximation

Consider the definite integral

$$\int_x^{x+\delta x} f(\xi) d\xi. \quad (6.1)$$

over an infinitesimal domain (i.e.  $\delta x \ll 1$ ). By defining  $F(\xi) = \int f(\xi) d\xi$  as the *antiderivative* of function  $f(\xi)$  (and equivalently,  $F'(\xi) = f(\xi)$ ), we can now compute the integral *symbolically*;

$$\int_x^{x+\delta x} f(\xi) d\xi = [F(\xi)]_x^{x+\delta x} = F(x + \delta x) - F(x). \quad (6.2)$$

But that doesn't really help us if we don't know the antiderivative  $F(\xi)$ , let alone the integrand  $f(\xi)$ !

### 6.1.1 Taylor Series Interpretation

To see how we might approximate this, let us start by taking the Taylor series expansion of the integral around  $\delta x$ :

$$\begin{aligned} \int_x^{x+\delta x} f(\xi) d\xi &= F(x + \delta x) - F(x) \\ &= \left[ F(x) + \delta x F'(x) + \frac{1}{2} \delta x^2 F''(x) + \frac{1}{6} \delta x^3 F'''(x) + O(\delta x^4) \right] - F(x) \\ &= \delta x f(x) + \frac{1}{2} \delta x^2 f'(x) + \frac{1}{6} \delta x^3 f''(x) + O(\delta x^4), \end{aligned}$$

where we have made use of the previous relation  $F'(\xi) = f(\xi)$ . If we take out the factor  $\delta x/2$ , we get

$$\int_x^{x+\delta x} f(\xi) d\xi = \frac{1}{2} \delta x \left[ 2f(x) + \delta x f'(x) + \frac{1}{3} \delta x^2 f''(x) + O(\delta x^3) \right]. \quad (6.3)$$

The trick here is to notice that the bracketed term is similar (but not exactly the same) as the Taylor expansion of  $f(x) + f(x + \delta x)$  around  $\delta x$ :

$$f(x) + f(x + \delta x) = 2f(x) + \delta x f'(x) + \frac{1}{2} \delta x^2 f''(x) + O(\delta x^3).$$

Multiplying this by  $\delta x/2$  to match the integral expansion; they differ in the  $\delta x^3$  term. Expanding the integral, the third order term is  $\delta x^3 f''(x)/6$ , whereas here we have  $\delta x^3 f''(x)/4$ .

We can therefore make the following approximation to the integral:

$$\boxed{\int_x^{x+\delta x} f(\xi) d\xi \approx \frac{1}{2} \delta x [f(x) + f(x + \delta x)]}. \quad (6.4)$$

Furthermore, by considering the Taylor series of both sides, we see that the method has an inherent error of  $\mathcal{O}(\delta x^3)$ , or more precisely

$$\epsilon_T = \frac{1}{6} \delta x^3 f''(x) - \frac{1}{4} \delta x^3 f''(x) = -\frac{1}{12} \delta x^3 f''(x), \quad (6.5)$$

where  $\epsilon_T$  refers to the **local trapezoidal error**.

#### Example 6.1 Trapezoidal local error (Fortran)

Consider the integral

$$\int_x^{x+\delta x} \sin^2 \left( \frac{1}{2} x \right) dx$$

Apply the trapezoidal approximation at the point  $x = 4$ , and compare it to the exact solution for various values of  $\delta x$ .

**Solution:** Calculating the exact solution to the integral at the point  $x = 4$ , we find that

$$\int_4^{4+\delta x} \sin^2\left(\frac{1}{2}x\right) dx = \frac{1}{2} [\delta 4 + \sin(\delta x) - \sin(4 + \delta x)].$$

Alternatively, applying the trapezoidal approximation,

$$\int_4^{4+\delta x} \sin^2\left(\frac{1}{2}x\right) dx \approx \frac{1}{2}\delta x [\sin^2(2) + \sin^2(2 + \delta x/2)].$$

Let's write a Fortran program to compare the trapezoidal approximation to the exact result, for different values of  $\delta$ :

```
program trapezoidal
    implicit none
    integer :: i
    real(8) :: dx(6), localerror(6)

    dx = [0.001d0, 0.01d0, 0.1d0, 0.2d0, 0.5d0, 1.d0]

    do i=1,6
        localerror(i) = abs(exact(dx(i)) - approx(dx(i)))
    end do

    write(*,*) localerror

contains
    function exact(dx)
        ! exact integral
        real(8), intent(in) :: dx
        real(8) :: exact
        exact = 0.5d0*(dx + sin(4.d0) - sin(4.d0+dx))
    end function

    function approx(dx)
        ! function that implements the trapezoidal approximation
        real(8), intent(in) :: dx
        real(8) :: approx
        approx = 0.5d0*dx*(sin(2.d0)**2 + sin(2.d0+0.5d0*dx)**2)
    end function
end program trapezoidal
```

Plotting the output of this program on a log-log plot, and examining the gradient, we find that the local error scales in the order of  $\sim \delta x^3$ .

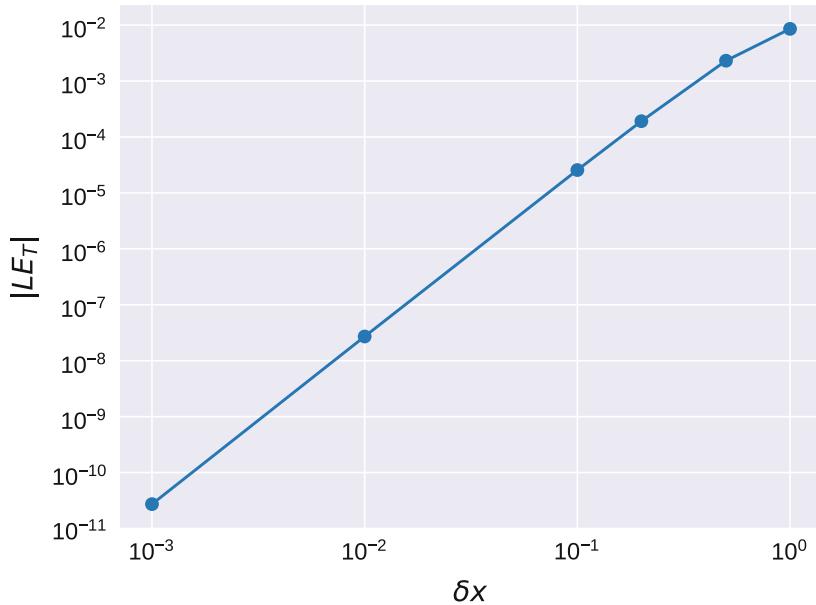


Figure 6.1 Numerical error of the trapezoidal approximation

### Example 6.2 Trapezoidal local error (Python)

Let's write a Python program to compare the trapezoidal approximation to the exact result, for various values of  $\delta$ :

```
#!/usr/bin/env python3
import numpy as np

# function to return exact result at x=4 for some dx
def exact(dx):
    return 0.5*(dx + np.sin(4) - np.sin(4+dx))

# trapezoidal approximation at x=4 for some dx
def trapezoid(dx):
    return 0.5*dx*(np.sin(2)**2 + np.sin(2+0.5*dx)**2)

# calculate the local error for various dx
local_error = []
for dx in [0.001, 0.01, 0.1, 0.2, 0.5, 1]:
    local_error.append([dx, np.abs(trapezoid(dx) - exact(dx))])
```

### 6.1.2 Geometric Interpretation

We can also interpret the trapezoidal approximation geometrically, by recognising that

$$\frac{1}{2}\delta x[f(x_0) + f(x_0 + \delta x)]$$

is the area of a trapezoid bounded by the points  $(x, 0)$ ,  $(x + \delta x, 0)$ ,  $(x, f(x))$ , and  $(x + \delta x, f(x + \delta x))$  – as shown in Fig. 6.2.

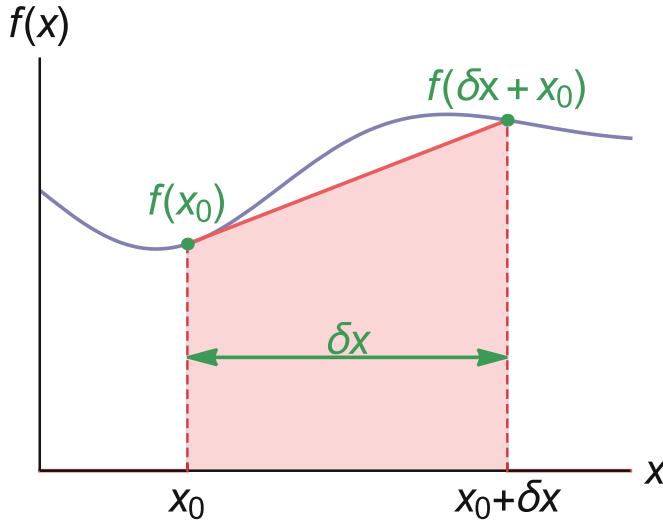


Figure 6.2

Essentially, what we are doing here is approximating the integral by integrating over a **linear interpolation between the boundary points**.

### 6.1.3 Composite Trapezoidal Rule

Everything we have done in this section has been over an infinitesimal domain with  $\delta x \ll 1$ . In practice, however, we rarely deal with problems like this, and we end up working with integrands with significant or even unknown variation between boundary points.

For example, consider integrating over a function  $f(x)$  from  $x = a$  to  $x = b$ . We could apply the trapezoidal approximation, with  $\Delta x = b - a$ ,

$$\int_a^b f(x)dx \approx \frac{1}{2}(b - a)[f(a) + f(b)], \quad (6.6)$$

however you can probably see that it is a *terrible* approximation – we are simply drawing a straight line between points  $f(a)$  to  $f(b)$  and integrating under it, ignoring any regions of non-linearities (for example oscillations). Instead, a better approximation is to break up the region  $[a, b]$  into  $N$  equally spaced intervals of size  $\Delta x$  (see Fig. 6.3), and apply the trapezoidal rule

► You may notice in Fig. 6.2 that the trapezoidal approximation is under-estimating the integral; this is because the example integrand is **concave down** (i.e.  $f''(x_0) < 0$ ) over the infinitesimal integration domain.

Analytically, a negative second derivative substituted into Eq. 6.5 results in

$$\text{error} = +\frac{1}{12}\delta x^3 f''(x_0)$$

a *positive* difference, and we can estimate the deviation from the exact solution.

On the other hand, if  $f(x)$  was **concave up** ( $f''(x_0) > 0$ ) over the integration domain, we can easily see that the trapezoidal method would *over-estimate* the integral.

locally to each interval before summing the results:

$$\int_a^b f(x)dx \approx \frac{1}{2}\Delta x [f(a) + f(a + \Delta x) + f(a + 2\Delta x) + \dots + f(a + (N - 1)\Delta x) + f(b)].$$

Rewritten using a summation,

$$\int_a^b f(x)dx \approx \frac{1}{2}\Delta x \sum_{n=0}^{N-1} [f(x + n\Delta x) + f(x + (n + 1)\Delta x)] \quad (6.7)$$

#### 6.1.4 Global Error

Recall that the error term for the trapezoidal approximation depends on  $f''(x)$ , and this will differ for each of the  $N$  intervals we are considering. So instead of computing the error term directly, we can deduce an upper bound for the total error.

Suppose that, over  $a \leq x \leq b$ , the maximum absolute value of  $f''(x)$  is  $M_2$  – that is  $|f''(x)| \leq M_2$ . Further, since the trapezoidal approximation is being applied  $N$  times, the total error  $E_T$  is bounded by

$$|E_T| \leq N \times \frac{1}{12}\Delta x^3 M_2. \quad (6.8)$$

Since  $N = (b - a)/\Delta x$ , this becomes

$$|E_T| \leq \frac{b - a}{12}\Delta x^2 M_2. \quad (6.9)$$

We can now also see that the total error will scale quadratically with  $\Delta x^2$ ; i.e. the trapezoidal rule is a **second order method**.

#### The trapezoidal method and ODEs

You may have noticed that we've already seen one trapezoidal method, when we were considering implicit Runge–Kutta methods for solving differential equations. Before you ascribe it to coincidence, let's integrate the initial value problem  $y'(x) = f(y(x), x)$ ,  $y(0) = y_0$ , using trapezoidal integration.

To start with, let's assume an infinitesimal integration domain,  $x_{n+1} = x_n + \delta x$ , where  $\delta x \ll 1$  and  $y(x_n) = y_n$ . Integrating the left hand side over the domain  $x \in [x_n, x_n + \delta x]$  is easy, we simply apply the

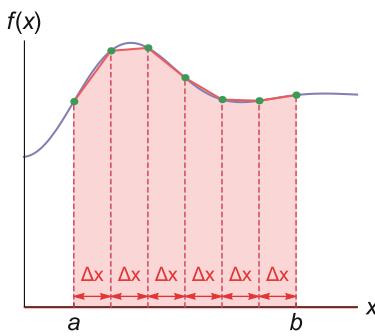


Figure 6.3

- When working with a discrete  $x$  grid in your program,

$$x_0, x_1, \dots, x_{N-1}$$

where  $x_0 = a$ ,  $x_{N-1} = b$ , and  $x_{n+1} - x_n = \Delta x$ , recall that

$$f(x + n\Delta x) \equiv f(x_n).$$

Similarly, if working with an already discrete function  $f(x_n) \equiv f_n$ , then

$$f(x + n\Delta x) \equiv f_n$$

fundamental theorem of calculus:

$$\int_{x_n}^{x_n + \delta x} y'(x) \, dx = y(x_{n+1}) - y(x_n) = y_{n+1} - y_n. \quad (6.10)$$

For the right hand side, let's use trapezoidal integration over an infinitesimal domain:

$$\int_{x_n}^{x_n + \delta x} f(y(x), x) \, dx = \frac{1}{2} \delta x [f(y(x_n), x_n) + f(y(x_{n+1}), x_{n+1})] + \mathcal{O}(\delta x^3). \quad (6.11)$$

Therefore equating both sides, we have

$$y_{n+1} = y_n + \frac{1}{2} \delta x [f(y(x_n), x_n) + f(y(x_{n+1}), x_{n+1})] + \mathcal{O}(\delta x^3). \quad (6.12)$$

We have arrived back at the trapezoidal method for solving ODEs! When solving a differential equation using the trapezoidal Runge–Kutta method, we are implicitly<sup>a</sup> using trapezoidal integration. Flick back to Sect. 5.8, and you will see that both methods share the same local and global error.

In fact, methods of numerical integration and differentiation are much more closely linked than we have presented here. This is to be expected since, in a certain sense, solving a differential equation is synonymous with integrating the equation. Due to their close relationship, results from one application of the technique can often be used to give insights into other cases — note how much easier it was to derive the local error term in the trapezoidal integration method?

---

<sup>a</sup>No pun intended.

## 6.2 Midpoint Rule

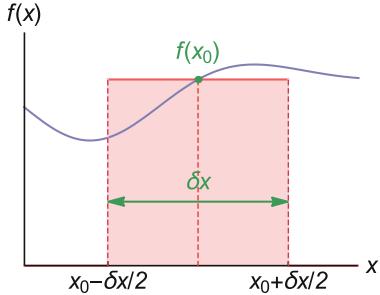


Figure 6.4

Let's return to the definite integral over an infinitesimal domain, but this time shift the integration limits:

$$\int_{x_0 - \delta x/2}^{x_0 + \delta x/2} f(\xi) d\xi. \quad (6.13)$$

Repeating our analysis of the previous section, taking the Taylor series around variable  $\delta x$ , we find

$$\int_{x_0 - \delta x/2}^{x_0 + \delta x/2} f(\xi) d\xi = \delta x f(x_0) + \frac{1}{24} \delta x^3 f''(x_0) + \mathcal{O}(\delta x^4). \quad (6.14)$$

Comparing this to what we have with the trapezoidal approximation (Eq. 6.3), notice that the  $f'(x_0)$  term vanishes. Therefore, we can make the so-called **midpoint approximation**,

$$\int_{x_0 - \delta x/2}^{x_0 + \delta x/2} f(\xi) d\xi \approx f(x_0) \delta x, \quad (6.15)$$

named as such because we are now approximating the integral by integrating over the **constant** value of  $f(x)$  evaluated at the *midpoint* of the domain – i.e. essentially approximating the area under the curve with a **rectangle** (see Fig. 6.4). This approximation, which is significantly simpler than the trapezoidal approximation, is still accurate to order  $\mathcal{O}(\delta x^3)$ !

In fact, if we compare the error terms explicitly, the **local midpoint error** term  $\epsilon_M$  is

$$\epsilon_M = \frac{1}{24} \delta x^3 f''(x), \quad (6.16)$$

Exactly half the value of the trapezoidal local error!

However, as before, we need to repeat the mid-point approximation  $N$  times over a non-infinitesimal interval in order to accurately approximate the integral,

$$\begin{aligned} \int_a^b f(x) dx &\approx \Delta x f(a + \Delta x/2) \\ &\quad + \Delta x f(a + \Delta x/2 + \Delta x) \\ &\quad + \dots \\ &\quad + \Delta x f(a + \Delta x/2 + (N-1)\Delta x). \end{aligned}$$

This gives rise to the **composite midpoint rule**:

$$\int_a^b f(x) dx \approx \Delta x \sum_{n=0}^{N-1} f\left(a + \left(n - \frac{1}{2}\right)\Delta x\right). \quad (6.17)$$

- The midpoint rule is much better suited for singularities than the trapezoidal rule, try sketching both approximations near a singularity.

- With only one function call, not two, per iteration, the midpoint rule offers another advantage over the trapezoidal rule.

### Discrete $x$ Grids and Functions

For an equally spaced discrete  $x$  grid

$$x_0, x_1, \dots, x_{N-1}$$

where

$$x_0 = a, \quad x_{N-1} = b, \quad x_{n+1} - x_n = \Delta x,$$

then the midpoint rule can be written

$$\int_a^b f(x) dx \approx \Delta x \sum_{n=0}^{N-1} f\left(\frac{x_{n+1} + x_n}{2}\right) \quad (6.18)$$

as  $(x_{n+1} + x_n)/2$  provides the midpoint between the  $n$ th and  $(n-1)$ th grid position.

However, unlike with the trapezoidal rule, if only a discretised integrand  $f(x_n) = f_n$  is provided, *and the function  $f(x)$  is unknown*, then it is *impossible* to use the midpoint method, as you cannot accurately determine the function value at the midpoint.

- ▶ We saw earlier that the trapezoidal method is linked to an implicit Runge–Kutta method.

The midpoint rule is simply the **leap-frog method** in integral form!

Consider  $y'(x) = f(y(x), x)$ . Integrating the left side using the fundamental theorem of calculus with  $h \ll 1$ ,

$$\begin{aligned} \int_{x-h}^{x+h} y'(\xi) d\xi \\ = y(x+h) - y(x-h), \end{aligned}$$

and the right side using the midpoint approximation,

$$\begin{aligned} \int_{x-h}^{x+h} f(y(\xi), \xi) d\xi \\ = 2hf(y(x), x) + \mathcal{O}(h^3), \end{aligned}$$

we get

$$\begin{aligned} y(x+h) &= y(x-h) \\ &\quad + 2hf(y(x), x) + \mathcal{O}(h^3) \end{aligned}$$

the leap-frog method!

#### 6.2.1 Global Error

Multiplying the local approximation error by  $N = (b-a)/\Delta x$ , and again letting  $M_2 = \max_{x \in [a,b]} |f''(x)|$ , we can find the upper bound of the absolute value of the total error  $|E_M|$ :

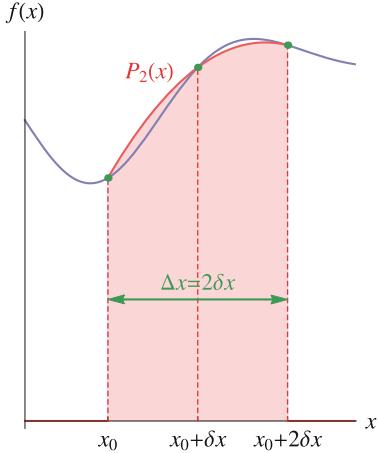
$$|E_M| \leq \frac{b-a}{24} \Delta x^2 M_2. \quad (6.19)$$

Thus the midpoint method is a **second order method**. When compared to the trapezoidal rule, we **generally** find that

$$|E_M| \approx \frac{1}{2} |E_T|.$$

However, it is important to note that our expressions for  $E_M$  and  $E_T$  are given as *upper bounds*, so this error relationship does not always hold – whilst for some functions we have the exact statement  $E_M = -\frac{1}{2}E_T$ , other functions can be constructed such that  $E_M > E_T$ .

### 6.3 Simpson's Rule



**Figure 6.5** Here you can see the geometric interpretation of Simpsons's rule. Note that the integrand  $f(x)$  (in blue) is being approximated by a **quadratic polynomial**  $P_2(x)$  (in red), which is found by interpolating between the three points  $x_0$ ,  $x_0 + \delta x$ , and  $x_0 + 2\delta x$ .

Looking at the trapezoidal and midpoint approximation, you may notice that it is relatively easy to cancel out the *local* errors by adding them in a simple linear combination:

$$2\epsilon_M + \epsilon_T = 2 \left( \frac{1}{24} \delta x^3 f''(x) \right) - \frac{1}{12} \delta x^2 f''(x) = 0. \quad (6.20)$$

It follows that by combining these two methods in the same way, we should end up with a higher order integral approximation. Let's give this a go, by considering an integral

$$I = \int_{x_0}^{x_0+2\delta x} f(\xi) d\xi. \quad (6.21)$$

Note that we have doubled the size of our interval, now of size  $2\delta x$ . This simply allows us to have a well defined ‘midpoint’  $x_0 + \delta x$  which we can use in the midpoint rule. The midpoint rule and trapezoidal rule applied to  $I$  therefore give:

$$M = 2\delta x f(x_0 + \delta x) \quad (6.22a)$$

$$T = \frac{1}{2}(2\delta x)[f(x_0) + f(x_0 + 2\delta x)] = \delta x[f(x_0) + f(x_0 + 2\delta x)]. \quad (6.22b)$$

We can now combine  $M$  and  $T$  using the ‘magic ratio’  $2M + T$  from Eq. 6.20 to approximate  $I$  whilst eliminating the third order error terms:

$$\begin{aligned} I &= \frac{1}{3}(2I + I) \approx \frac{1}{3}(2M + T) \\ &= \frac{1}{3}\delta x[f(x_0) + 4f(x_0 + \delta x) + f(x_0 + 2\delta x)]. \end{aligned} \quad (6.23)$$

This is the well-known Simpson's rule:

$$\int_{x_0}^{x_0+2\delta x} f(\xi) d\xi \approx \frac{1}{3}\delta x[f(x_0) + 4f(x_0 + \delta x) + f(x_0 + 2\delta x)]. \quad (6.24)$$

To see just how much more accurate Simpson's rule is, we can expand both sides of Eq. 6.24 as Taylor series expansions of  $\delta x$ , giving an (*local*) error term of

$$\epsilon_S = -\frac{1}{90} \delta x^5 f'''(x_0), \quad (6.25)$$

i.e. the local error is two orders of magnitude smaller than the trapezoidal and midpoint rule!

### 6.3.1 Composite Simpson's Rule

For finite intervals, we need to apply Simpson's rule successively  $N$  times in order to accurately approximate the integral (as in previous sections). This gives rise to the composite Simpson's rule:

$$\begin{aligned} \int_a^b f(x) dx &\approx \frac{1}{3}\Delta x[f(a) + 4f(a + \Delta x) + f(a + 2\Delta x) \\ &\quad + f(a + 2\Delta x) + 4f(a + 3\Delta x) + f(a + 4\Delta x) \\ &\quad + \dots \\ &\quad + f(a + (N - 2)\Delta x) + 4f(a + (N - 1)\Delta x) + f(b)] \\ &\approx \frac{1}{3}\Delta x[f(a) + 4f(a + \Delta x) + 2f(a + 2\Delta x) + \dots + 4f(a + (N - 1)\Delta x) + f(b)]. \end{aligned}$$

Or, in summation notation,

$$\int_a^b f(x) dx \approx \frac{1}{3}\Delta x \left[ f(a) + 2 \sum_{n=1}^{N/2-1} f(a + 2n\Delta x) + 4 \sum_{n=1}^{N/2} f(a + (2n - 1)\Delta x) + f(b) \right].$$

Note that, unlike the midpoint and trapezoidal rules, we require  $N$  be **even** for Simpson's Rule. This is due to the restriction that three points are required for the quadratic interpolation for each 'interval'.

### Discrete $x$ Grids and Functions

For an equally spaced discrete  $x$  grid

$$x_0, x_1, \dots, x_{N-1}$$

where

$$x_0 = a, \quad x_{N-1} = b, \quad x_{n+1} - x_n = \Delta x,$$

then Simpson's rule can be written

$$\int_a^b f(x) dx \approx \frac{1}{3}\Delta x \left[ f(x_0) + 2 \sum_{n=1}^{N/2-1} f(x_{2n}) + 4 \sum_{n=1}^{N/2} f(x_{2n-1}) + f(x_{N-1}) \right].$$

By making the appropriate substitutions above, this expression can also be used for an equally spaced discrete  $f$  grid where  $f(x_n) = f_n$ .

### 6.3.2 Global Error

Like the midpoint and trapezoidal rules, we can determine an upper bound to the global composite error. Multiplying the local approximation error by

$(b - a)/(2\Delta x)$  (since each iteration of the Simpson's rule takes place over an interval of size  $2\Delta x$ ), and letting  $M_4 = \max_{x \in [a,b]} |f^{(4)}(x)|$ , the upper bound of the absolute value of the total error is

$$|E_S| \leq \frac{b-a}{180} \Delta x^4 M_4 \quad (6.26)$$

i.e. Simpson's rule is a **fourth order method**.

### 6.3.3 Simpson's 3/8 Rule

- ▶ Sound familiar? Back in Problem 5.6, one of the problems had you work through the Butcher tableaux of a fourth-order Runge–Kutta method called the 3/8-rule method. Yep, you've guessed it, this is equivalent to integrating via Simpson's 3/8 rule!

By extending Simpson's rule to an infinitesimal interval of size  $3\delta x$ , we can use cubic interpolation to approximate the integrand – this leads to Simpson's 3/8 rule:

$$\int_{x_0}^{x_0+3\delta x} f(x) dx \approx \frac{3}{8} \delta x [f(x_0) + 3f(x_0 + \delta x) + 3f(x_0 + 2\delta x) + f(x_0 + 3\delta x)].$$

This has local error

$$E_{S3} = -\frac{3}{80} \delta x^5 f^{(4)}(x_0),$$

and so scales similarly to the regular Simpson's rule – both are fourth order methods when compounded. In fact, comparing the coefficients of the error –  $1/90$  vs.  $3/80$  – it even appears as if the Simpson's 3/8 method (using cubic interpolation) is *less* accurate than quadratic interpolation! This can't be the case, can it?

The issue arises due to the fact that we aren't comparing them over the same interval sizes. The quadratic case requires one midpoint, and thus an interval of  $\Delta x = 2\delta x$ , whereas the 3/8 case needs *two* midpoints for cubic interpolation, and thus uses a larger interval of  $\Delta x = 3\delta x$ . Substituting these expressions into the local error terms to write them in terms of  $\Delta x$  allows us to compare them fairly:

$$\begin{aligned} E_S &= -\frac{1}{90} \left(\frac{\Delta x}{2}\right)^5 f^{(4)}(x_0) = -\frac{1}{2880} \Delta x^5 f^{(4)}(x_0) \\ E_{S3} &= -\frac{3}{80} \left(\frac{\Delta x}{3}\right)^5 f^{(4)}(x_0) = -\frac{1}{6480} \Delta x^5 f^{(4)}(x_0). \end{aligned}$$

Thus, compared over the *same* interval, the 3/8 method is  $6480/2880 = 2.25$  times more accurate.

- ▶ Using this as a guide, can you see what the error term must be for the fourth order Runge–Kutta 3/8-rule?

In practice, however, we rarely use Simpson's 3/8 method, simply due to the increased complexity (requiring  $N$  be a multiple of 3 and the additional function calls per iteration) and there is no increase in scaling compared to quadratic interpolation.

**Example 6.3** Simpson's rule for normalisation (Fortran)

The wavefunction  $\psi(x)$  of a quantum system, determined by solving the Schrödinger equation, satisfies the **Born rule**. This states that the absolute value squared of the wavefunction,  $|\psi(x)|^2$ , provides the probability distribution of the system.

As such, we require the wavefunction to be **normalised**:

$$\int_{-\infty}^{\infty} |\psi(x)|^2 dx = 1.$$

Or, using the Dirac's bra-ket notation where  $\psi(x) = \langle x | \psi \rangle$ ,

$$\langle \psi | \psi \rangle = \int_{-\infty}^{\infty} \psi(x)^* \psi(x) dx = 1.$$

Use Simpson's rule to normalise the ground-state quantum harmonic oscillator wavefunction  $\psi(x) = e^{-x^2/2}$  over the domain  $x \in [-5, 5]$  with  $\Delta x = 0.01$ .

**Solution:** There are two possible approaches to implementing Simpson's rule.

First, we define a user-defined function and then make function calls:

```
program simpsons
    implicit none
    integer :: N, i
    real(8) :: dx, xmin, xmax, A

    ! grid spacing and grid points
    dx = 0.01d0
    xmin = -5.d0; xmax = 5.d0
    N = (xmax-xmin)/dx + 1

    ! Simpson's rule using function calls
    A = prob(xmax) + prob(xmin)
    do i=2,N-1
        if (mod(i,2)==0) then
            ! even terms
            A = A + 4*prob(xmin + i*dx)
        else
            ! odd terms
            A = A + 2*prob(xmin + i*dx)
        endif
    end do
    A = A*dx/3.d0

    write(*,*)A
    write(*,*)1.d0/sqrt(A)

contains
    function prob(x)
        real(8), intent(in) :: x
```

```

real(8) :: prob
prob = abs(exp(-x**2/2.d0))**2
end function
end program simpsons

```

Alternatively, we can represent the wavefunction to be normalised as an *array*, and use array slicing to perform the summation:

```

program simpsonsArray
    implicit none
    integer :: N, i
    real(8) :: dx, xmin, xmax, A
    real(8), allocatable :: prob(:)

    ! grid spacing and grid points
    dx = 0.01d0
    xmin = -5.d0; xmax = 5.d0
    N = (xmax-xmin)/dx + 1

    ! create probability grid
    allocate(prob(1:N))
    prob = [(abs(exp(-(xmin+i*dx)**2/2.d0))**2, i=0,N-1)]

    ! Simpson's rule using array slicing
    A = prob(1) + 4*sum(prob(2:N-1:2)) + 2*sum(prob(3:N-2:2)) + prob(N)
    A = A*dx/3.d0

    write(*,*)A
    write(*,*)1.d0/sqrt(A)
end program simpsonsArray

```

#### Example 6.4 Simpson's rule for normalisation (Python)

The wavefunction  $\psi(x)$  of a quantum system, determined by solving the Schrödinger equation, satisfies the **Born rule**. This states that the absolute value squared of the wavefunction,  $|\psi(x)|^2$ , provides the probability distribution of the system.

As such, we require the wavefunction to be **normalised**:

$$\int_{-\infty}^{\infty} |\psi(x)|^2 dx = 1.$$

Or, using Dirac's bra-ket notation where  $\psi(x) = \langle x | \psi \rangle$ ,

$$\langle \psi | \psi \rangle = \int_{-\infty}^{\infty} \psi(x)^* \psi(x) dx = 1.$$

Use Simpson's rule to normalise the ground-state quantum harmonic oscillator wavefunction  $\psi(x) = e^{-x^2/2}$  over the domain  $x \in [-5, 5]$  with  $\Delta x = 0.01$ .

**Solution:** There are two possible approaches to implementing Simpson's rule.

First, we define a user-defined function (for brevity, we use `lambda` functions) and then make function calls:

```
#!/usr/bin/env python3
import numpy as np

# Define the probability function
prob = lambda x: np.abs(np.exp(-x**2/2))**2

# grid spacing and grid points
dx = 0.01
a = -5; b = 5
N = int((b-a)/dx + 1)

# Simpson's rule using function calls
A = prob(a) + prob(b)
for n in range(1, N):
    # even terms
    if n % 2 == 0:
        A += 4*prob(a + n*dx)
    else:
        A += 2*prob(a + n*dx)
A *= dx/3

# normalise the wavefunction
psiNorm = lambda x: np.exp(-x**2/2)/np.sqrt(A)
```

Alternatively, we can represent the wavefunction to be normalised as an *array*, and use array slicing to perform the summation. Since we keep the NumPy arrays as they are, and do not apply any Python for loops, this has the potential to be much faster than the above simple Python loop.

```
#!/usr/bin/env python3
import numpy as np

# create the wavefunction grid
dx = 0.01
x = np.arange(-5, 5+dx, dx)
psi = np.exp(-x**2/2)
prob = np.abs(psi)**2

# Simpson's rule using NumPy slicing
A = (dx/3) * (prob[0] + 4*np.sum(prob[1:-1:2]) +
               2*np.sum(prob[2:-1:2]) + prob[-1])

# normalise the wavefunction
psiNorm = psi/np.sqrt(A)
```

## 6.4 Newton–Cotes Rules

All the methods we have considered so far involve polynomial interpolations of the integrand across the small interval  $\Delta x$ , using **equally spaced points** (each interpolation point is separated by a constant  $\delta x$ ). These are known as **Newton–Cotes formulas**, which can be extended to higher orders of interpolation, if need be. In general, the  $n$ th point Newton–Cotes formula over infinitesimal integration domain  $n\delta x$  has the following form:

$$\int_{x_0}^{x_0+n\delta x} f(x) dx = \delta x \sum_{i=0}^n a_{i,n} f(x_0 + i\delta x) \quad (6.27)$$

where  $\delta x \ll 1$ ,  $n$  is the number of points required for evaluation, and  $a_{i,n}$  are the coefficients or **quadrature weights**, which satisfy the relationship

$$\sum_{i=0}^n a_{i,n} = n - 1. \quad (6.28)$$

### 6.4.1 Boole's Rule

By using quadratic interpolation (5 equally spaced points  $x_0, x_1, \dots, x_4$  with spacing  $\delta x$ ) we get Boole's rule, a higher order Newton–Cotes rule:

$$\int_{x_0}^{x_0+4\delta x} f(x) dx \approx \frac{2}{45} \delta x [7f(x_0) + 32f(x_1) + 12f(x_2) + 32f(x_3) + 7f(x_4)].$$

This has local error

$$E_B = -\frac{8}{945} \delta x^7 f^{(6)}(x_0),$$

and, if we were to write the composite Boole's method, would have *global* error of

$$|E_B| \leq \frac{2(b-a)}{945} \Delta x^6 \max_{x \in [a,b]} |f^{(6)}(x)|. \quad (6.29)$$

We can see that the global error scales as  $\mathcal{O}(\Delta x^6)$ , and thus the composite Boole's rule is an example of a 6th order Newton–Cotes method.

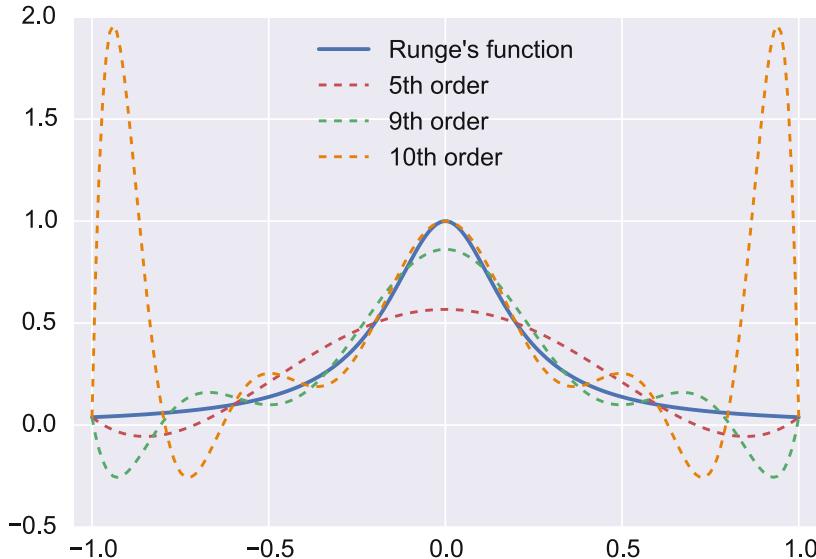
### 6.4.2 Runge's Phenomenon

It may seem intuitive that we should be able to increase the interpolation order indefinitely, in order to produce a Newton–Cotes rule with arbitrarily small error. However, this is not always the case! The reason is something we haven't touched on much this chapter, but that you'll be familiar with from the chapter on differential equation solves — **numerical stability**.

- Runge's phenomenon is named after the German mathematician Carl David Tolm  Runge.

For example, consider the Runge function, defined by

$$f(x) = \frac{1}{1 + 25x^2} \quad (6.30)$$



**Figure 6.6**

In Fig. 6.6, the Runge function is plotted alongside increasing orders of polynomial interpolations – all calculated using equally spaced points, as required by the Newton–Cotes methods. Whilst it does seem to get more accurate near the center of the function as the interpolation order increases, the opposite appears to be happening near the endpoints of  $x = \pm 1$ , and our interpolation starts to *diverge* from the exact solution.

Why does this occur? An important result in the field of numerical stability analysis states that, if the quadrature weights are all non-negative (i.e.  $a_{i,n} > 0$  for all  $i$  and  $n$ ), then the numerical integration method is stable. In the case of the Newton–Cotes methods, this is not the case; in fact, only the Newton–Cotes rules with  $n \in \{1, 2, 3, 4, 5, 6, 7, 9\}$  satisfy  $a_{i,n} > 0$  for all  $i$ .

As a result, equally-spaced polynomial interpolation is **unstable** for the Runge function, and the error will increase towards infinity as we increase the order of interpolation! In order to numerically integrate the Runge function to the required accuracy, we instead need to use a polynomial interpolation with non-equal spacing.

- This doesn't mean we can never use Newton–Cotes methods! For most cases they work just fine, the Runge function can be considered a 'stiff' function in the context of numerical integration.

## 6.5 Gauss–Legendre Quadrature

If the function to be integrated is known (that is, we have the power to evaluate the function at specified values, rather than just being provided a list of fixed data), then it is possible to numerically calculate the integral using a polynomial fit of non-equal spacing. In fact, by clustering the interpolation points near the endpoints of the function, we can avoid Runge’s phenomenon *as well as* producing an algorithm of **higher order** than the Newton–Cotes equivalent with the same number of function evaluations.

In this section, we’ll look at one such algorithm, known as **Gaussian Quadrature**, and explore several available function weightings.

### 6.5.1 2-Point Gauss–Legendre Quadrature

Similarly to what we did with the Newton–Cotes methods, let’s attempt to approximate an integral over an infinitesimal interval using Taylor series expansions. However, this time, let’s start with an arbitrary solution using **two** function evaluations and **unknown** evaluation points

$$x_- = x_0 + c_- \delta x, \quad x_+ = x_0 + c_+ \delta x$$

where  $x_{\pm} \in [x_0, x_0 + \delta x]$ . Thus:

$$\int_{x_0}^{x_0 + \delta x} f(x) dx = \frac{1}{2} \delta x [f(x_0 + c_- \delta x) + f(x_0 + c_+ \delta x)]. \quad (6.31)$$

Expanding both sides as Taylor series expansions around  $\delta x$ ,

$$\begin{aligned} \text{LHS: } & \delta x f(x_0) + \frac{1}{2} \delta x^2 f'(x_0) + \frac{1}{6} \delta x^3 f''(x_0) + \mathcal{O}(\delta x^4) \\ \text{RHS: } & \delta x f(x_0) + \frac{1}{2} \delta x (c_- + c_+) f'(x_0) + \frac{1}{6} \delta x^3 \left[ \frac{3}{2} (c_-^2 + c_+^2) \right] f''(x_0) + \mathcal{O}(\delta x^4). \end{aligned}$$

Comparing the coefficients of the two series, we can see that, in order for them to be equal up to at least fourth order, we must have

$$\begin{cases} c_- + c_+ = 1 \\ c_-^2 + c_+^2 = \frac{2}{3} \end{cases} \Rightarrow c_{\pm} = \frac{1}{2} \left( 1 \pm \frac{1}{\sqrt{3}} \right) \quad (6.32)$$

This gives us the **2-point Gauss–Legendre Quadrature formula**:

$$\boxed{\int_{x_0}^{x_0 + \delta x} f(x) dx = \frac{1}{2} \delta x [f(x_-) + f(x_+)]}$$

where  $x_{\pm} = x_0 + \frac{1}{2} \delta x \left( 1 \pm \frac{1}{\sqrt{3}} \right)$

If we extend the Taylor series to sixth-order in  $\delta x$ , we find that the local error is given by

$$\epsilon_{2GL} = \frac{1}{4320} \delta x^5 f^{(4)}(x_0). \quad (6.33)$$

This is the **same order** as the local error from Simpson's 3/8 rule (see Sect. 6.3.3) – in fact, both methods use cubic interpolation and provide exact results for cubics.

However, the 2-point Gauss–Legendre rule uses two function evaluations, as opposed to the four required for Simpson's rule, the equivalent Newton–Cotes method. This can lead to a *significant* reduction in computational time!

### Composite 2-Point Gaussian–Legendre Method

For finite intervals, we need to apply the 2-point Gauss–Legendre Quadrature formula successively  $N$  times, as in previous sections. This gives rise to the composite 2-point Gauss–Legendre Quadrature method:

$$\begin{aligned} \int_a^b f(x) dx &\approx \frac{1}{2} \Delta x [f(a + \frac{1}{2}c_- \Delta x) + f(a + \frac{1}{2}c_+ \Delta x) \\ &\quad + \dots \\ &\quad + f(a + (n-1)\Delta x + \frac{1}{2}c_- \Delta x) + f(a + (n-1)\Delta x + \frac{1}{2}c_+ \Delta x)], \end{aligned}$$

or, in summation notation,

$$\int_a^b f(x) dx \approx \frac{1}{2} \Delta x \sum_{n=0}^{N-1} \left[ f\left(a + (n + \frac{1}{2}c_-)\Delta x\right) + f\left(a + (n + \frac{1}{2}c_+)\Delta x\right) \right].$$

It can be easily seen (by multiplying the local error by  $N = (b-a)/\Delta x$ ) that the **global error** of the composite rule is given by

$$|E_{2GL}| \leq \frac{b-a}{4320} \Delta x^4 \max_{x \in [a,b]} |f^{(6)}(x)|,$$

i.e. the 2-point Gauss–Legendre method is a **fourth-order** method.

### 6.5.2 $N$ -Point Gauss–Legendre Quadrature

We can increase the accuracy of the 2-point Gauss–Legendre method above by increasing the number of mid-interval function evaluations; here is the general solution:

$$\int_{-1}^1 f(x) dx \simeq \sum_{n=1}^N w_n f(x_n) \quad (6.34)$$

where the function is evaluated  $N$  times at points  $x_n$ , with weightings given by  $w_n$ .

- ▶ Why is the Gauss–Legendre quadrature more stable than the Newton–Cotes methods? It can be proved that  $w_n > 0$  for all  $n$ , and as such all Gauss–Legendre methods are stable.

Hang on, you say. Why the sudden insistence on the integration bounds  $-1 \leq x \leq 1$ ? That's a bit inconvenient. And why is it called the Gauss–Legendre rule, anyway? To answer this question (and many more!) we need to take a quick detour into the world of Legendre polynomials.

#### Legendre Polynomials

Legendre polynomials are the solutions to the differential equation

$$(x^2 - 1)L_n''(x) + 2xL_n'(x) - n(n+1)L_n(x) = 0.$$

As this can be written as a Sturm–Liouville problem, the Legendre Polynomials are **orthogonal** to each other, satisfying the orthogonality relationship

$$\int_{-1}^1 L_m(x)L_n(x) dx = \frac{2}{2n+1}\delta_{mn}.$$

Here is a list of the first few Legendre polynomials:

| $n$ | $L_n(x)$                 |
|-----|--------------------------|
| 0   | 1                        |
| 1   | $x$                      |
| 2   | $\frac{1}{2}(3x^2 - 1)$  |
| 3   | $\frac{1}{2}(5x^3 - 3x)$ |

Note that the  $L_n(x)$  is a polynomial of order  $n$  – the Legendre polynomials can also be computed by orthogonalising the set  $\{1, x, x^2, x^3, \dots\}$  subject to the above orthogonality condition – and thus form a **complete basis** on the interval  $-1 \leq x \leq 1$ . This allows us to write *any* function within this interval in terms of the Legendre polynomials:

$$f(x) = \sum_{n=1}^{\infty} a_n L_n(x),$$

as well as any polynomial of degree  $n$  as a *finite* sum of Legendre polynomials:

$$P_n(x) = \sum_{n=0}^N c_n x^n = a_0 L_0(x) + a_1 L_1(x) + \dots + a_n L_n(x).$$

It is therefore easy to verify that the following property holds for **any** polynomial  $P_m(x)$ :

$$\int_{-1}^1 P_m(x) L_n(x) dx = 0, \quad m = 0, 1, \dots, n-1.$$

Now that we've recalled some basic properties of Legendre polynomials, let's consider an arbitrary polynomial  $P_{2N-1}(x)$ , of degree  $2N - 1$ . Using polynomial long division, we can divide it by the Legendre polynomial  $L_N(x)$  of order  $N$ , as below:

$$P_{2N-1}(x) = q(x)L_N(x) + r(x), \quad (6.35)$$

where  $q(x)$  is the **quotient polynomial**, of order  $N - 1$ ; and  $r(x)$  is the **remainder polynomial**, of maximum order  $N - 1$ .

Integrating this expression over the domain  $[-1, 1]$ :

$$\begin{aligned} \int_{-1}^1 P_{2N-1}(x) dx &= \int_{-1}^1 q(x)L_N(x) dx + \int_{-1}^1 r(x) dx \\ &= 0 + \int_{-1}^1 r(x) dx \\ &= \int_{-1}^1 r(x) dx. \end{aligned}$$

Using Legendre polynomials, we have reduced an integral over a  $2N - 1$  polynomial to an integral over a  $N - 1$  polynomial!

Furthermore, recall from previous discussion of Newton-Cotes methods that in order to interpolate a  $(N-1)$ th degree polynomial *exactly*, you need to perform an interpolation using a minimum of  $N$  points along the polynomial within the domain of interest. In the most general form, this can be written as:

$$r(x) = \sum_{n=1}^N r(x_n) P_N^{(n)}(x).$$

Performing the above definite integration gives

$$\begin{aligned} \int_{-1}^1 P_{2N-1}(x) dx &= \int_{-1}^1 r(x) dx \\ &= \sum_{n=1}^N \left( \int_{-1}^1 P_N^{(n)}(x) dx \right) r(x_n) \\ &= \sum_{n=1}^N w_n r(x_n), \end{aligned} \quad (6.36)$$

► The integral of  $q(x)L_N(x)$  over the domain  $x \in [-1, 1]$  vanishes as it is of order  $N - 1$  and thus orthogonal to  $L_N$ .

Thus, it is the **orthogonality property** of Legendre polynomials which requires the integration domain to be restricted to  $x \in [-1, 1]$  in Eq. 6.34.

► For more on polynomial interpolation, see exercise Problem 6.3

i.e. the integration over the interpolating polynomials is what provides the weighting constants  $w_n$ .

That's all well and good, but we can't expect you to do polynomial long division to determine  $r(x)$  every time you want to do perform a Gaussian quadrature method! To simplify it somewhat, we can make use of the fact that up until now, no constraints have been placed on the choice of the points  $x_1, x_2, \dots, x_N$ .

Consider the integrand again – from the polynomial long division we have

$$\sum_{n=1}^N w_n P_{2N-1}(x_n) = \sum_{n=1}^N w_n q(x_n) L_N(x_n) + \sum_{n=1}^N w_n r(x_n).$$

If we choose  $x_n$  to be the  $N$  **roots** of the Legendre polynomial  $L_n(x)$ ; i.e. they satisfy the equation  $L_n(x_n) = 0$ , then the first summation on the right hand side vanishes, and we are left with the equality

$$\sum_{n=1}^N w_n P_{2N-1}(x_n) = \sum_{n=1}^N w_n r(x_n).$$

Substituting this into Eq. 6.5.2, we now have the equality

$$\boxed{\int_{-1}^1 P_{2N-1}(x) dx = \sum_{n=1}^N w_n P_{2N-1}(x_n)}. \quad (6.37)$$

That is, **the integral of a polynomial of degree  $2N - 1$  can be calculated exactly via  $N$  function evaluations**. Of course, when dealing with *arbitrary* functions, we can no longer rely on there being an equality, and we recover the equation originally provided

$$\boxed{\int_{-1}^1 f(x) dx \simeq \sum_{n=1}^N w_n f(x_n)}. \quad (6.38)$$

However, we have now shown that, compared to a Newton–Cotes procedure of similar order, significantly fewer function evaluations/ data points are required.

### Weights and Zeros

- If implementing the Gauss–Legendre algorithm, **do not** evaluate the weighting function inside your algorithm!

Instead, pre-compute or **hard-code** the numeric values of  $x_n$  and the weights  $w_n$ .

The weights required for the Gauss–Legendre method depend on the number of points  $N$  that are to be used to approximate the integral in each infinitesimal region, *as well as* the value  $x_n$ . They can be calculated from the following expression:

$$w_n = \frac{2}{(1 - x_n)^2 [L'_N(x_n)]^2}, \quad n = 1, 2, \dots, N. \quad (6.39)$$

The zeros of the  $N$ th Legendre function, and weights  $w_n$  for the cases  $N = 1, 2$  and  $3$  are listed in Table 6.1.

| $N$ | $x_n$                   | $w_n$      | Local error                                | Global order |
|-----|-------------------------|------------|--------------------------------------------|--------------|
| 1   | 0                       | 2          | $\frac{1}{24}\delta x^3 f''(x_0)$          | Second order |
| 2   | $\pm\frac{1}{\sqrt{3}}$ | 1          | $\frac{1}{4320}\delta x^5 f^{(4)}(x_0)$    | Fourth order |
| 3   | 0<br>$\pm\sqrt{3/5}$    | 8/9<br>5/9 | $\frac{1}{2016000}\delta x^7 f^{(6)}(x_0)$ | Sixth order  |

**Table 6.1** Gauss–Legendre comparison for various values of  $N$ .  $x_n$  are the zeros of the  $N$ th Legendre function, and  $w_n$  the respective weights

### Shifted Gauss–Legendre: Arbitrary Integration Bounds

In most cases, the integrals we are trying to solve will not involve the integration bounds  $-1 \leq x \leq 1$ ; we need a method of transforming to an arbitrary domain  $a \leq x \leq b$ . This can be done quite easily by simply scaling the integrand  $f(x)$  through a change of integration variables, such that the region we are interested in integrating over occurs in the region  $x \in [-1, 1]$ .

Let the new integration variable be  $u(x)$ ; we require  $u(a) = -1$  and  $u(b) = 1$ . Therefore a suitable choice is simply an **affine transformation** – that is, a straight line through these two points:

$$u = \frac{1}{a-b}(a+b-2x) \Rightarrow x = \frac{b-a}{2}u + \frac{a+b}{2} \quad dx = \frac{1}{2}(b-a)du,$$

After making this substitution, it is straightforward to see that

$$\int_a^b f(x) \, dx = \frac{1}{2}(b-a) \int_{-1}^1 f\left(\frac{b-a}{2}x + \frac{a+b}{2}\right) \, dx, \quad (6.40)$$

and therefore, applying the  $N$ -point Gauss–Legendre formula,

$$\int_a^b f(x) \, dx \simeq \frac{1}{2}(b-a) \sum_{n=1}^N w_n f\left(\frac{b-a}{2}x_n + \frac{a+b}{2}\right). \quad (6.41)$$

There are a couple of things to look out for with Gauss–Legendre quadrature methods. Aside from the fact that, depending on previous algorithms, it is not always possible to use un-equally spaced  $x_n$ , the use of Legendre polynomials guarantees good results *only* when the scaled integrand  $f(u)$  can be well approximated by polynomials in the region  $u \in [-1, 1]$ .

In the case of singularities, infinite integration limits, and other ‘exotic’ behaviour, we need to redefine the Gaussian quadrature using slightly more applicable orthogonal functions.

- Using this result, you can verify the 2-point Gauss–Legendre approximation found previously using Taylor series.

## 6.6 Other Common Gaussian Quadratures

If the integrand function  $f(x)$  exhibits markedly non-polynomial behaviour (such as a singularity) and if we are able to write it in the form  $f(x) = \omega(x)g(x)$ , where  $g(x)$  *can* be well-approximated by a polynomial, then we can reformulate the Gaussian quadrature to be of the form

$$\int_a^b \omega(x)g(x) dx = \sum_{n=1}^{\infty} w_n f(x_n). \quad (6.42)$$

As with Gauss–Legendre quadrature, the weights and evaluation points depend on the orthogonality condition that allows for the reduction in interpolation order and thus function evaluation. In this generic case, the orthogonality condition is now

$$\int_a^b \omega(x)Y_n(x)Y_m(x) dx = A\delta_{nm}, \quad (6.43)$$

where  $\{Y_n(x)\}$  is a complete set of orthogonal functions with respect to the weighting function  $\omega(x)$  and integration limits  $a$  and  $b$ , and  $A$  is the normalisation constant.

As before, the evaluation points of the  $N$ th order Gaussian quadrature ( $\{x_n\} = \{x_1, \dots, x_N\}$ ) are given by the zeros of  $Y_N(x)$ . Some commonly used Gaussian quadratures are listed below.

### 6.6.1 Gauss–Hermite Quadrature

A weighting function of

$$\omega(x) = e^{-x^2}$$

requires **Hermite polynomials**, satisfying the orthogonality condition

$$\int_{-\infty}^{\infty} e^{-x^2} H_n(x)H_m(x) dx = \sqrt{\pi}2^n n! \delta_{nm}.$$

In this case, the quadrature weights for quadrature order  $N$  are given by

$$w_n = \frac{2^{N-1} N! \sqrt{\pi}}{N^2 [H_{N-1}(x_n)]^2}, \quad n = 1, 2, \dots, N$$

and the integration bounds are  $-\infty \leq x \leq \infty$ .

### 6.6.2 Gauss–Laguerre Quadrature

A weighting function of

$$\omega(x) = e^{-x}$$

requires **Laguerre polynomials**. In this case, the quadrature weights for order  $N$  are given by

$$w_n = \frac{x_n}{(N+1)^2 [L_{N+1}(x_n)]^2}, \quad n = 1, 2, \dots, N$$

and the integration bounds are  $0 \leq x \leq \infty$ .

- When calculating the weights, it is often convenient to use **recurrence relations** in order to calculate  $H_N(x)$  from  $H_{N-1}(x)$  and  $H_{N-2}(x)$ .

Alternatively, use online resources to find tables of numeric values of  $w_n$  and  $x_n$ .

### 6.6.3 Chebyshev–Gauss Quadrature

The Chebyshev–Gauss quadrature actually refers to two different weightings; one that corresponds to Chebyshev polynomials of the first kind  $T_n(x)$ , and one that corresponds to Chebyshev polynomials of the second kind.

#### Chebyshev Polynomials of the First Kind

These have a weighting function of

$$\omega(x) = \frac{1}{\sqrt{1-x^2}}.$$

The weights and zeros are then given by

$$w_n = \frac{\pi}{N}, \quad x_n = \cos\left(\frac{2n-1}{2N}\pi\right), \quad n = 1, 2, \dots, N.$$

#### Chebyshev Polynomials of the Second Kind

These have a weighting function of

$$\omega(x) = \sqrt{1-x^2}.$$

The weights and zeros are then given by

$$w_n = \frac{\pi}{N+1} \sin^2\left(\frac{n\pi}{N+1}\right), \quad x_n = \cos\left(\frac{n\pi}{N+1}\right), \quad n = 1, 2, \dots, N.$$

► Note that for the first Chebyshev–Gauss quadrature:

- There is an explicit analytical solution for the roots of the  $T_N(x)$
- The weights are independent of the summation index  $n$ :

$$w_n \equiv w$$

## 6.7 Monte Carlo Methods

Suppose we are working with a 2D, rather than 1D, system; how do we deal with the arising 2-dimensional integrals? It's rather simple, actually – using **Fubini's theorem**, we can treat the double integral as *two iterated single* integrals:

$$\iint_A f(x, y) \, dx \, dy = \int_X \left( \int_Y f(x, y) \, dy \right) dx. \quad (6.44)$$

We simply need to apply our chosen numerical algorithm to each dimension iteratively, slowly reducing the dimensions of the discrete  $f$  grid until we are left with a scalar.

- If accuracy requires  $N$  function evaluations per dimension, then  $D$  dimensions results in  $N^D$  evaluations!

The resulting scaling is  $\mathcal{O}(N^D)$  aka **exponential**.

We also find that the **error** term also increases with dimensionality, scaling like  $\mathcal{O}(N^{-1/D})$ .

This approach is fine if the numerical function values are already available in memory (say, from the output of a previous numerical algorithm), as weights just need to be applied to each element and the entire array summed. On the other hand, if the integrand is a known function that must be computed from scratch, the number of function evaluations scales almost exponentially with each additional dimension! This is the so-called ‘curse of dimensionality’.

One such solution to this conundrum is through the use of **Monte Carlo methods** – this is a *statistical* method that allows us to approximate the value of an integral by choosing a selection of random points. We'll start with a quick example to make it slightly easier to visualise.

### 6.7.1 Calculating $\pi$

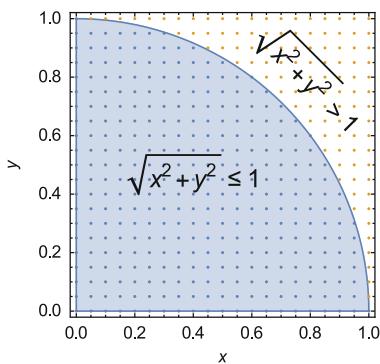
Consider the segment of a circle formed from the region function  $x^2 + y^2 \leq 1$  and bounded by the positive  $x$  and positive  $y$  axes. This is a quarter of a circle of radius 1, and thus will have area  $A = \pi r^2 / 4 = \pi/4$ . Although somewhat trivial analytically, let's attempt to find the area of the circle-segment (and thus approximate  $\pi$ ) using Monte Carlo methods.

To begin with, we might start by uniformly sampling the rectangular space  $x, y \in [0, 1]$  using  $N$  points in each dimension (see Fig. 6.7). The ratio of the number of points *within* the circle segment,  $N_{in}$ , to the total number of points  $N^2$  will provide a reasonable estimate of the ratio of the circular segment area  $A$  to total area  $A_{total}$ :

$$\frac{A}{A_{total}} \approx \frac{N_{in}}{N^2} \Rightarrow A \approx \frac{N_{in}}{N^2} A_{total}. \quad (6.45)$$

As you might expect, increasing the total number of points sampled increases the accuracy. But, we haven't avoided the curse of dimensionality! For every extra dimension in a given problem, we need an additional  $N$  points, resulting in  $N^D$  evaluations.

Instead of sampling uniformly, let's instead sample the region by choosing  $N$  points, denoted  $X_n$ , that are *randomly but uniformly distributed*.



**Figure 6.7** Uniformly sampling with  $\Delta x = 0.05$  and  $\Delta y = 0.05$

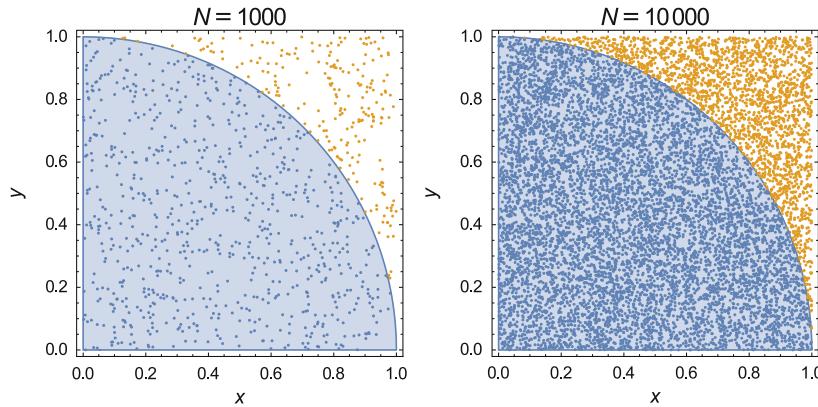


Figure 6.8

When  $N = 1000$ , we end up with 783 points inside the circular segment, and when  $N = 10000$ , we get 7854:

$$N = 1000 : A \approx \frac{783}{1000} \times 1^2 = 0.783$$

$$N = 10000 : A \approx \frac{7854}{10000} \times 1^2 = 0.7854$$

Comparing this to the actual area,  $A = \pi/4 = 0.785398$ , we can see we get pretty close. Of course, every time we run the code we will be choosing a different set of random points, altering our estimate of the circular segment area.

We can get a good estimate of the error of this statistical method by looking at the standard deviation. First, let's define the function

$$F(X_n) = \begin{cases} 0, & |X_n| > 1 \\ 1, & |X_n| \leq 1 \end{cases}, \quad (6.46)$$

i.e. if the random variable is located inside the circular segment, it gives it a value of 1, otherwise it gets a value of 0. Now, we know that the probability of a random point being located inside the circular segment is given by

$$p_{in} = \frac{\text{area of segment}}{\text{total area}} = \frac{\pi}{4} \quad (6.47)$$

and therefore the probability of the point being located outside the circular segment is  $p_{out} = 1 - \frac{\pi}{4}$ . It's easy now to calculate the expectation value of  $F$  and  $F^2$ , and the standard deviation of  $F$ :

$$\langle F \rangle = 0 \times p_{out} + 1 \times p_{in} = \frac{\pi}{4} \quad (6.48)$$

$$\langle F^2 \rangle = 0^2 \times p_{out} + 1^2 \times p_{in} = \frac{\pi}{4} \quad (6.49)$$

$$\sigma_F = \sqrt{\langle F^2 \rangle - \langle F \rangle^2} = \sqrt{\frac{\pi}{4} \left(1 - \frac{\pi}{4}\right)}. \quad (6.50)$$

► If a random variable  $X$  can take values  $x_i$  with probability  $p_i$ , then the **expectation value** of  $X$  is defined by

$$\langle X \rangle = \sum_i x_i p_i.$$

Recall that in our statistical method for finding the area of the circular segment, we are simply calculating  $N_{in}/N$ . This can be written in terms of the function  $F$  acting on our randomly generated points,  $\mathbf{x}$ :

$$\frac{N_{in}}{N} = \frac{1}{N} \sum_{n=1}^N F(\mathbf{x}_n) = \bar{F}, \quad (6.51)$$

so, in essence, we are basically measuring the **sample mean** of our random variable  $F$ . This tells us that we can estimate the error in our measurement of  $\bar{F}$  by calculating  $\sigma_{\bar{F}}$ , the standard deviation of the sample mean. Luckily, there is a useful result that relates the standard deviation of a sample mean to the standard deviation of the relevant random variable:

$$\sigma_{\bar{F}} = \frac{\sigma_F}{\sqrt{N}} = \frac{1}{\sqrt{N}} \sqrt{\frac{\pi}{4} \left( \sqrt{1 - \frac{\pi}{4}} \right)} \propto \frac{1}{\sqrt{N}}. \quad (6.52)$$

So we can see that our statistical method for calculating  $\pi$  has an error that scales with the inverse square root of the number of random points we sample.

Note that we have been calculating a two-dimensional integral in disguise – don’t look so shocked, we are in a chapter on numerical integration! By sampling the function  $F$  over the region  $R = x, y \in [0, 1]$ , we have numerically estimated the following integral:

$$\iint_R F(\mathbf{x}) d\mathbf{x}. \quad (6.53)$$

Surprise! And here’s the shocker: we found that the statistical Monte-Carlo method we used scales as  $\mathcal{O}(N^{-1/2})$  — no  $D$  in sight. We have managed to cure ourselves of the curse of dimensionality.

### 6.7.2 Monte-Carlo Integration

- ▶ Be careful – just because it’s free of the curse of dimensionality, doesn’t mean the Monte-Carlo method should be used for *all* numerical integrations!

As a general rule of thumb, the Monte-Carlo approximation tends to beat out other methods when the dimension is higher than 2.

Of course, most of the time you find yourself needing to use the Monte-Carlo method of integration, you’ll most likely be wanting to do something more complicated than calculate the area of a quarter-circle. In this section, we’ll generalise the Monte-Carlo integration method.

Say we wish to numerically estimate the following  $d$ -dimensional integral of an arbitrary function  $f : \mathbb{R}^d \rightarrow \mathbb{R}^d$  over the domain  $R$ ,

$$\int_R f(\mathbf{x}) d\mathbf{x}. \quad (6.54)$$

Let  $V(R)$  be the ( $d$ -dimensional) volume of the domain  $R$ . Then, choosing  $N$  randomly distributed points  $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\} \in R$ , the Monte-Carlo integration method is given by

$$\int_R f(\mathbf{x}) d\mathbf{x} \approx V(R) (\bar{f} \pm \sigma_{\bar{f}}), \quad (6.55)$$

where

$$\bar{f} = \frac{1}{N} \sum_{n=1}^N f(\mathbf{x}_n), \quad (6.56)$$

$$\sigma_{\bar{f}} = \frac{1}{\sqrt{N}} \sqrt{\sum_{n=1}^N f(\mathbf{x}_n)^2 - \bar{f}^2}. \quad (6.57)$$

Note that, unlike the deterministic grid based Newton–Cotes and quadrature methods in previous sections, we don't define a local or global error. Due to the statistical method of the sampling, it makes more sense to define the uncertainty in the approximation in terms of the sample standard deviation  $\sigma_{\bar{f}}$  – this will change in value with every new random sampling.

### Reusing the wheel

If using Python, external modules provide some very useful tools and pre-written algorithms for numerically calculating the integral. Before you sigh and promptly forget the preceding chapter, knowing the ins and outs of how exactly numerical integration algorithms work is extremely important, and will give you an understanding of *when* and *how* to use each algorithm to reduce the chance of instability and error.

On the other hand, this doesn't mean you have to hand code these algorithms every time — often, you'll find that well-established libraries (such as **NumPy** and **SciPy** for Python, and **QUADPACK** for Fortran<sup>a</sup>) have highly optimised implementations built-in.

For example, SciPy includes a whole sub-package, `scipy.integrate`, that provides a number of numerical integration techniques. These include inbuilt trapezoidal, Simpson's, and Newton–Cotes rules, as well as Gaussian quadrature, and several others — the provided methods can be used on user-defined functions, as well as arrays representing a discretised function. Not only that, but SciPy also provides the ability to perform general purpose double and triple integration.

For example, to use Simpson's rule to calculate the integral of the discretised function  $x^2$  over the  $x$ -grid  $0 \leq x_n \leq 1$ , we can use `scipy.integrate.simps`:

```
>>> import numpy as np
>>> from scipy.integrate import simps
>>> dx = 0.1
>>> x = np.arange(0, 1+dx, dx)
>>> y = x**2
>>> simps(y, x)
0.33333333333333337
```

► To generate the randomly distributed points, you can use the following NumPy function in Python,

`numpy.random.random`

If using Fortran, random numbers can be generated using the intrinsic function

`random_number()`

In both cases, the random floating point numbers are uniformly distributed in the region  $0 \leq x < 1$ .

<sup>a</sup>SciPy provides a Python interface to QUADPACK behind the scenes

---

**Further reading**

---

This chapter has been designed to provide a gentle introduction to the topic of numerical integration — but there is still so much more to discuss! You could (and people have) filled whole textbooks on the analysis of numerical integration. Beyond Newton–Cotes and Gaussian quadratures, there is a wealth of quadratures designed to solve different problems, including Romberg integration and ClenshawCurtis quadrature.

One topic we haven't covered in as much detail is the **stability of numerical integration**; for a great text that delves in-depth into all the proofs and derivations, see Hackbusch (2014).

On the computational side, a good place to find out about the numerical integration methods provided by SciPy and QUADPACK are from the source themselves — their online documentation.

- Hackbusch, Wolfgang (2014), *The Concept of Stability in Numerical Mathematics*, Springer Berlin Heidelberg, ISBN 978-3-642-39386-0.
-

## Exercises

**P6.1** Consider the integral

$$\int_a^b f(x) \, dx.$$

For a *single* subinterval (i.e.  $N = 1$  and  $\Delta x = b - a$ ),

- (a) calculate the integral analytically, using the trapezoidal rule, and using the midpoint rule, for
  - (i)  $f(x) = 1$
  - (ii)  $f(x) = x$
  - (iii)  $f(x) = x^2$
  - (iv)  $f(x) = x^3$
- (b) For each  $f(x)$  in part (a), calculate the error  $E_T$  and  $E_M$  of the trapezoidal and midpoint rule respectively by subtracting the exact solution. In each case, how are  $E_M$  and  $E_T$  related?
- (c) Thus, deduce that this relationship holds for all cubic polynomials.

**P6.2** (a) Write a numerical integration module in Python or Fortran that contains functions which perform midpoint, trapezoidal, and Simpson's rules. Each of these functions should accept four arguments:

- $f$ , a function to be integrated
- $a$ , the lower bound of the integral
- $b$ , the upper bound of the integral
- $dx$ , the step size of the  $x$ -axis

- (b) Write a program that contains an internal function to calculate  $f(x) = 1 + 2 \cos(2\sqrt{x})$ .

Using your numerical integration module from (a), calculate the integral

$$\int_0^1 (1 + 2 \cos(2\sqrt{x})) \, dx$$

using the midpoint, trapezoidal, and Simpson's methods.

- (c) Calculate the exact solution to the definite integral in (b), and use this result to analyse and compare the error in your numerical approximations.

How does the error of each method scale as  $dx$  changes?

- (d) Calculate the upper bound error ( $|E_T|$ ,  $|E_M|$ ,  $|E_S|$ ) of each method. How does this compare to your previous error analysis?

**P6.3** We can use Lagrange polynomials in order to interpolate a polynomial function between discrete points.

For the function  $f(x)$ , if we know 3 discrete points,

$$(x_0, f(x_0)), \quad (x_1, f(x_1)), \quad (x_2, f(x_2))$$

then using the method of Lagrange polynomials will provide a 2nd order or quadratic polynomial interpolating function,  $P_2(x)$ , as follows:

$$\begin{aligned} P_2(x) &= \sum_{j=0}^2 \left( f(x_j) \prod_{\substack{0 \leq m \leq 2 \\ m \neq j}} \frac{x - x_m}{x_j - x_m} \right) \\ &= f(x_0) \frac{x - x_1}{x_0 - x_1} \frac{x - x_2}{x_0 - x_2} + f(x_1) \frac{x - x_0}{x_1 - x_0} \frac{x - x_2}{x_1 - x_2} + f(x_2) \frac{x - x_0}{x_2 - x_0} \frac{x - x_1}{x_2 - x_1} \end{aligned}$$

The quadratic interpolating function  $P_2(x)$ , as calculated above, closely approximates  $f(x)$  in the vicinity  $x_0 \leq x \leq x_2$ , and satisfies  $P_2(x_i) = f(x_i)$  for the three points.

- (a) Calculate  $P_2(x)$  using the three points

$$(x_0, f(x_0)), \quad (x_0 + \delta x, f(x_0 + \delta x)), \quad (x_0 + 2\delta x, f(x_0 + 2\delta x))$$

- (b) Derive Simpson's rule by approximating the integral of  $f(x)$  over  $[x_0, x_0 + 2\delta x]$  by the integral of  $P_2(x)$ ,

$$\int_{x_0}^{x_0+2\delta x} f(x) dx \approx \int_{x_0}^{x_0+2\delta x} P_2(x) dx$$

Hint: evaluate the right hand side. This should be relatively easy, although the algebra gets quite tedious!

**P6.4** The electrostatic potential of a finite segment of charged wire of length  $L$  and constant linear charge density  $\lambda$  is given by

$$V(\mathbf{r}) = k\lambda \int_{-L}^L \frac{1}{|\mathbf{r}|} d\ell$$

where  $|\mathbf{r}|$  is the distance from the infinitesimal line charge  $d\ell$  to the observer, and  $k = \frac{1}{4\pi\epsilon_0}$ .

Consider an observer standing at distance  $H = 1\text{m}$  perpendicularly from the center of a  $2\text{m}$  wire (i.e.  $L = 1$ ); see Fig. 6.9.

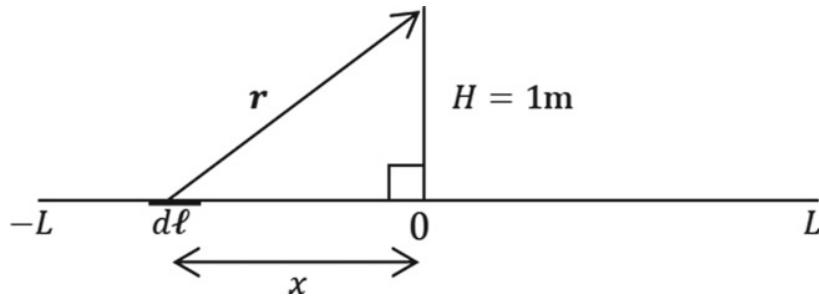


Figure 6.9

- (a) Calculate the potential measured by the observer by evaluating the definite integral exactly. Leave your answer in terms of  $k\lambda$ .  
Hint: use the change of variables  $x = \tan u$ .
- (b) Use Simpson's rule to numerically evaluate the definite integral, and compare it to your exact solution from part (a)

**P6.5** Consider the cubic polynomial

$$f(x) = x^3 + \sqrt{2}x^2 + x$$

- (a) Divide  $f(x)$  by the Legendre polynomial  $L_2(x) = \frac{3}{2}x^2 - \frac{1}{2}$  using polynomial long division to determine the quotient  $q(x)$  and the remainder  $r(x)$ .  
What is the polynomial order of  $r(x)$ ?
- (b) Verify that the following integral *vanishes*:

$$\int_{-1}^1 q(x)L_2(x) \, dx$$

- (c) Show that

$$\int_{-1}^1 f(x) \, dx = \int_{-1}^1 r(x) \, dx$$

by evaluating both integrals analytically.

**P6.6** (a) Write a Fortran program to calculate the integral

$$\int_0^1 (1 + 2 \cos(2\sqrt{x})) \, dx$$

using the shifted Gauss–Legendre quadrature method, of order  $N = 2$  and  $N = 3$ .

- (b) Calculate the exact solution to the definite integral in (b), and use this result to analyse and compare the error in your numerical approximations.  
How does the error of each method scale as  $\Delta x$  changes?

**P6.7** Consider the integral

$$\int_{-\infty}^{\infty} \cos^2(x) e^{-x^2/2} dx = \frac{1+e^2}{e^2} \sqrt{\frac{\pi}{2}}$$

- (a) Make an appropriate change of variable  $x = g(u)$  to transform this integral into the form

$$\int_{-\infty}^{\infty} f(u) e^{-u^2} du$$

- (b) The Hermite polynomial of order  $N = 6$  is given by

$$H_6(x) = 64x^6 - 480x^4 + 720x^2 - 120$$

Use a numerical root finding method in order to solve  $H_6(x_n) = 0$  and calculate the six roots of this polynomial,  $x_1, x_2, \dots, x_6$ .

Hint:  $H_6(x)$  is symmetric around the  $y$ -axis, and all roots occur in the region  $-3 \leq x \leq 3$ .

- (c) Calculate the respective Gauss–Hermite quadrature weights numerically using the formula

$$w_n = \frac{2^{N-1} N! \sqrt{\pi}}{N^2 [H_{N-1}(x_n)]^2}, \quad n = 1, 2, \dots, 6$$

Note that  $H_5(x) = 32x^5 - 160x^3 + 120x$ .

- (d) Thus, approximate the transformed integral from part (a) by applying the 6th order Gauss–Hermite quadrature:

$$\int_{-\infty}^{\infty} f(u) e^{-u^2} du \approx \sum_{n=1}^6 w_n f(x_n)$$

- (e) The original integral over  $x \in [-\infty, \infty]$  has been approximated by a summation over **just 6 terms!**

Compare your approximation with the analytical solution to determine the error. How well does this approximation compare to the analytical solution?

**P6.8** (a) Compute the 2-dimensional integral

$$\int_{-0.5}^1 \int_{-1}^1 \sin^2(x^3 y^2) dx dy$$

using Simpson's method, with grid spacing  $\Delta x = \Delta y = 0.005$ .

- (b) The integral is altered so that the integration limits on variable  $x$  now **depend** on the value of  $y$ :

$$\int_{-0.5}^1 \int_{-1}^y \sin^2(x^3 y^2) \ dx \ dy$$

Using the same parameters as before ( $\Delta x = \Delta y = 0.005$ ), what does the integral now evaluate to?

- P6.9** (a) Compute the 2-dimensional integral

$$\int_{-0.5}^1 \int_{-1}^1 \sin^2(x^3 y^2) \ dx \ dy$$

using the Monte-Carlo method by randomly sampling the integration region.

- (b) What is the uncertainty (the standard deviation) in your result? How does this change with  $N$ , the number of points sampled?



## Chapter 7

# The Eigenvalue Problem

When it comes to numerical computation, solving differential equations, integration, and root finding is only part of the story. What if you find yourself in a perilous life-or-death situation, your only salvation being able to numerically solve the set of linear equations  $\mathbf{y} = A\mathbf{x}$ ?

Solving systems of linear equations, matrix factorisation and operations, curve fitting and linear regression, finding eigenvalues and eigenvectors, linear map, and tensor algebra all fall under the umbrella term of linear algebra. This is a huge field, with numerous textbooks wholly devoted to the various facets of linear algebra. In this chapter, we'll focus on a brief subset that will be an important part of solving the Schrödinger equation to obtain the quantised energies (eigenvalues) and their corresponding wave-functions (eigenvectors).

### 7.1 Eigenvalues and Eigenvectors

At first glance, the eigenvector equation looks deceptively simple. Suppose that we have a square matrix  $A$ ; what vectors  $\mathbf{v}$  and scalars  $\lambda$  can we find that satisfy the equation

$$A\mathbf{v} = \lambda\mathbf{v}. \quad (7.1)$$

The solution(s)  $\mathbf{v}_i$  and the corresponding  $\lambda_i$  are referred to as the eigenvectors and eigenvalues respectively. When solving the eigenvalue equation, we commonly take the following approach and rearrange the equation to the following form:

$$(A - \lambda I)\mathbf{v} = 0. \quad (7.2)$$

We can see from the above that the solutions to the eigenvector equation amount to finding the nullspace of matrix  $A - \lambda I$ . As such, non-zero solutions only exist in the case  $|A - \lambda I| = (\lambda_1 - \lambda)(\lambda_2 - \lambda) \cdots (\lambda_n - \lambda) = 0$ , where  $|M|$  denotes the determinant of the enclosed matrix  $M$ . This is the **characteristic**

► Note that the eigenvectors are unique up to a certain factor; if  $\mathbf{v}$  is an eigenvector, then so is  $c\mathbf{v}$  where  $c$  is an arbitrary constant. Therefore a common constraint when calculating eigenvectors is

$$|\mathbf{v}| = \sqrt{\sum_i v_i^2} = 1.$$

**polynomial**, and the roots provide the eigenvalues; these can then be substituted back into the eigenvalue equation to calculate the corresponding eigenvectors.

That's most likely how you were shown to calculate the eigenvalues and eigenvectors in your undergraduate linear algebra course, and perhaps many a study session was spent tediously calculating the eigenvector and eigenvalues by hand. Luckily for you, this time we want to use numerical techniques and get computers to do it for us — but where to begin? We've already covered root finding algorithms, so perhaps a numerical method to calculate the determinant, followed by the Newton–Raphson method?

Not so fast! Fortunately (or unfortunately, depending on your point of view), eigenvalues and eigenvectors are so vastly important across computer science, mathematics, and physics, that specialised numerical techniques exist for calculating them, for various classes of matrices. In the following sections, we'll introduce a few of these algorithms, including the power iteration, Arnoldi iteration, and Lanczos iteration.

To start with, however, let's have a quick eigenvalue and eigenvector refresher.

### Eigenvalues and Eigenvectors Refresher

**Theorem 7.1.** *All square matrices have at least one eigenvalue,  $\lambda \in \mathbb{C}$ , and associated eigenvector  $\mathbf{v} \in \mathbb{C}^n$ .*

*Proof:* Consider an  $N \times N$  matrix  $A$ , and an arbitrary non-zero vector  $\mathbf{u} \in \mathbb{C}^N$ . From this, we can construct the set of vectors  $\mathcal{K} = \{\mathbf{u}, A\mathbf{u}, A^2\mathbf{u}, \dots, A^N\mathbf{u}\}$ . Since we have  $N + 1$  vectors spanning a  $N$ -dimensional vector space, the spanning set must be *linearly dependent*; that is, there exists some non-trivial solution to the equation

$$c_0\mathbf{u} + c_1A\mathbf{u} + c_2A^2\mathbf{u} + \cdots + c_NA^N\mathbf{u} = 0 \quad (7.3)$$

for constant coefficients  $c_i \in \mathbb{C}$ , with  $c_N \neq 0$ . Now, the **Fundamental Theorem of Algebra** states that every non-constant  $N$ th-degree polynomial  $P$  with complex coefficients has  $N$  (not necessarily unique) values  $\{z_1, z_2, \dots, z_N\}$  such that  $P(z_i) = 0$ . Therefore,

$$\begin{aligned} 0 &= (c_0I + c_1A + c_2A^2 + \cdots + c_NA^N)\mathbf{u} \\ &= c_N(A - z_1I)(A - z_2I) \cdots (A - z_NI)\mathbf{u}. \end{aligned} \quad (7.4)$$

Since  $c_N \neq 0$ , we must have  $(A - z_iI)$  for some  $z_i$ . Thus,  $A$  has at least one eigenvalue  $z_i$ , with the nullspace of  $(A - z_iI)$  the corresponding eigenvector.  $\square$

**Theorem 7.2.** *Eigenvectors with different eigenvalues must be linearly independent.*

*Proof:* Consider a matrix  $A \in \mathbb{C}^{N \times N}$  which has eigenvalues  $\lambda_1 \neq \lambda_2$  and associated eigenvectors  $\mathbf{v}_1, \mathbf{v}_2$ . Consider the linear combination

$$0 = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 \quad (7.5)$$

where  $c_1, c_2 \in \mathbb{C}$ . Multiplying this equation by  $A - \lambda_2 I$ , and recalling that, by definition,  $A\mathbf{v}_i = \lambda_i \mathbf{v}_i$ ,

$$\begin{aligned} 0 &= c_1(A - \lambda_2 I)\mathbf{v}_1 + c_2(A - \lambda_2 I)\mathbf{v}_2 \\ &= c_1(\lambda_1 - \lambda_2)\mathbf{v}_1 + c_2(\lambda_2 - \lambda_2)\mathbf{v}_2 \\ &= c_1(\lambda_1 - \lambda_2). \end{aligned} \quad (7.6)$$

Since  $\lambda_1 \neq \lambda_2$ , the only possible value of  $c_1$  to satisfy this equation is  $c_1 = 0$ . Repeating this process but pre-multiplying by  $A - \lambda_1 I$ , it can be shown that we must also have  $c_2 = 0$ . Thus, Eq. 7.5 only admits the trivial solution, and  $\mathbf{v}_1$  and  $\mathbf{v}_2$  must be linearly independent.  $\square$

**Corollary 7.3.** *An  $N \times N$  matrix  $A$  can have, at most,  $N$  distinct eigenvalues.*

The eigenvectors of  $A$  will span the vector space  $\mathbb{C}^N$  — thus there can be a maximum number of  $N$  linearly independent eigenvectors, each having a distinct eigenvalue as per Eq. 7.2.

**Definition 7.4. Similarity transform.** Two matrices  $A \in \mathbb{C}^N$  and  $B \in \mathbb{C}^N$  are related via a similarity transform  $S \in \mathbb{C}^N$  if  $A = S^{-1}BS$ . As a result, they share the same eigenvalues  $\lambda_i$ , and the eigenvectors of  $B$  are given by  $S\mathbf{v}$ , where  $\mathbf{v}$  are the eigenvectors of  $A$ .

**Theorem 7.5.** *A matrix  $A \in \mathbb{C}^{N \times N}$  is diagonalizable, that is, there exists a non-singular column matrix  $P$  and **diagonal matrix**  $\Lambda = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_N)$  such that*

$$A = P\Lambda P^{-1} \quad (7.7)$$

*if and only if  $A$  has  $N$  linearly independent eigenvectors.*

*Proof:* Assume  $A$  has  $N$  eigenvectors  $\mathbf{v}_i$  satisfying the eigenvalue equation  $A\mathbf{v}_i = \lambda_i \mathbf{v}_i$ . Now, let's construct the matrix  $P = [\mathbf{v}_1 \ \mathbf{v}_2 \ \cdots \ \mathbf{v}_N]$ , consisting of the eigenvectors along the columns. Using summation notation, let's consider the matrix product  $P\Lambda$ :

$$\begin{aligned} (P\Lambda)_{ij} &= \sum_k P_{ik}\Lambda_{kj} = \sum_k (\mathbf{v}_k)_i \lambda_k \delta_{kj} \\ &= \sum_k \sum_n A_{im}(\mathbf{v}_k)_m \delta_{kj} \\ &= \sum_m A_{im}(\mathbf{v}_j)_m \\ &= (AP)_{ij}. \end{aligned} \quad (7.8)$$

Now, the matrix  $P$  is non-singular and invertible if and only if the columns of  $P$ , and therefore the eigenvectors of  $A$ , are linearly independent. If this is satisfied, then

$$P\Lambda = AP \Rightarrow \Lambda = P^{-1}AP. \quad (7.9)$$

$\square$

► Recall that the matrix product  $AB$  is defined as

$$(AB)_{ij} = \sum_k A_{ik}B_{kj}$$

where  $ij$  represents the  $(i, j)$ th element of the matrix product.

►  $\delta_{ij}$  is the Kronecker delta, and is equal to 1 if  $i = j$ , and 0 otherwise.

### Symmetric and Hermitian Matrices

**Definition 7.6. Hermitian matrices.** If a matrix  $A \in \mathbb{C}^{M \times N}$  remains unchanged after taking the **conjugate transpose**,  $A^\dagger = A$ , then  $A$  is an Hermitian matrix. If the elements of  $A$  are all real, then this reduces to  $A^T = A$  and  $A$  is also a **symmetric matrix**.

**Theorem 7.7.** An Hermitian matrix  $A \in \mathbb{C}^{N \times N}$  has  $N$  real eigenvalues  $\lambda_1, \dots, \lambda_N \in \mathbb{R}$ .

*Proof:* Consider the matrix-vector product  $\mathbf{v}^\dagger A \mathbf{v}$ , where  $\mathbf{v}$  is an arbitrary eigenvector of  $A$  satisfying  $A\mathbf{v} = \lambda\mathbf{v}$ :

$$\mathbf{v}^\dagger (A\mathbf{v}) = \lambda \mathbf{v}^\dagger \mathbf{v}. \quad (7.10)$$

Alternatively,

$$(\mathbf{v}^\dagger A)\mathbf{v} = (A\mathbf{v})^\dagger \mathbf{v} = \lambda^* \mathbf{v}^\dagger \mathbf{v}. \quad (7.11)$$

Comparing these two results, we see that  $\lambda = \lambda^*$  and thus  $\lambda \in \mathbb{R}$ .  $\square$

**Theorem 7.8.** An Hermitian matrix  $A \in \mathbb{C}^N$  has  $N$  linearly independent and orthogonal eigenvectors.

*Proof:* Consider an Hermitian matrix  $A \in \mathbb{C}^N$ , with two distinct eigenvalues  $\lambda_1 \neq \lambda_2$ , and linearly independent eigenvectors  $\mathbf{v}_1, \mathbf{v}_2$ . Following a similar process to the previous proof, let's consider the matrix-vector product  $\mathbf{v}_1^\dagger A \mathbf{v}_2$ :

$$\begin{aligned} \mathbf{v}_1^\dagger (A\mathbf{v}_2) &= \lambda_2 \mathbf{v}_1^\dagger \mathbf{v}_2 \\ (\mathbf{v}_1^\dagger A)\mathbf{v}_2 &= (A\mathbf{v}_1)^\dagger \mathbf{v}_2 = \lambda_1 \mathbf{v}_1^\dagger \mathbf{v}_2. \end{aligned}$$

Thus, we have the relation  $\lambda_1 \mathbf{v}_1^\dagger \mathbf{v}_2 = \lambda_2 \mathbf{v}_1^\dagger \mathbf{v}_2$ . Since  $\lambda_1 \neq \lambda_2$ , the only way to satisfy this relationship is if  $\mathbf{v}_1^\dagger \mathbf{v}_2 = 0$ , i.e. eigenvectors with different eigenvalues are **orthogonal**.

**Corollary 7.9.** Since orthogonal eigenvectors are, by definition, linearly independent, all Hermitian matrices are diagonalizable.

Since we have the freedom to arbitrarily scale eigenvectors, scaling the eigenvectors of Hermitian matrix  $A \in \mathbb{C}^{N \times N}$  such that  $\mathbf{v}_i^\dagger \mathbf{v}_i = |\mathbf{v}_i| = 1$ , produces an orthonormal set of eigenvectors that span  $\mathbb{R}^N$ . Constructing the column matrix  $U = [\mathbf{v}_1 \ \mathbf{v}_2 \ \cdots \ \mathbf{v}_N]$  to diagonalise  $A$ , we have

$$\Lambda = U^{-1}AU \quad (7.12)$$

where  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_N)$  is the diagonal matrix of eigenvalues. Note that, since the eigenvectors are orthonormal, by definition

$$U^\dagger U = I \Rightarrow U^{-1} = U^\dagger. \quad (7.13)$$

A matrix satisfying this property is referred to as **unitary**, and thus Hermitian matrices are diagonalised by a unitary matrix:

$$\Lambda = U^\dagger AU. \quad (7.14)$$

## 7.2 Power Iteration

Power iteration is the simplest method to calculate the eigenvalues of a matrix; however, with this simplicity comes certain constraints. Consider a matrix  $A \in \mathbb{C}^{N \times N}$ , that has the following properties:

- it is diagonalisable, or equivalently, it has  $N$  linearly independent eigenvectors  $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_N$ ,
- its eigenvalues can be ordered like  $|\lambda_1| > |\lambda_2| \geq \dots \geq |\lambda_N|$ . The largest (absolute) eigenvalue and its associated eigenvector  $\mathbf{v}_1$  are referred to as the **dominant** eigenvalue and **dominant** eigenvector.

Since the eigenvectors are linearly independent, they span the vector space  $\mathbb{R}^N$ ; as such, we can write an arbitrary vector  $\mathbf{x}_0 \in \mathbb{R}^N$  as a linear combination of the eigenvectors:

$$\mathbf{x}_0 = \sum_{i=1}^N c_i \mathbf{v}_i. \quad (7.15)$$

Let's multiply this expansion by  $A^n$ :

$$\begin{aligned} x_n &= A^n \mathbf{x}_0 = \sum_{i=1}^N c_i A^n \mathbf{v}_i = \sum_{i=1}^N c_i \lambda_i^n \mathbf{v}_i \\ &= c_1 \lambda_1^n \mathbf{v}_1 + \lambda_1^n \sum_{i=2}^N c_i \left( \frac{\lambda_i}{\lambda_1} \right)^n \mathbf{v}_i, \end{aligned} \quad (7.16)$$

where we have factored out  $\lambda_1^n$  in the last step. Now, taking the limit  $n \rightarrow \infty$ , we find that

$$\lim_{n \rightarrow \infty} A^n \mathbf{x}_0 = c_1 \lambda_1^n \mathbf{v}_1 \quad (7.17)$$

since  $\lambda_i/\lambda_1 \rightarrow 0$ . This remarkable result shows that for (almost) any arbitrary vector we choose, applying  $A$  repeatedly a large number of times allows us to recover the eigenvector  $\mathbf{v}_1$  corresponding to the largest eigenvalue  $\lambda_1$ !

Of course, the sharp-eyed amongst you might have noticed a few constraints in the above derivation:

- For this procedure to work, we must have  $c_1 \neq 0$ . This is equivalent to requiring that our starting vector  $\mathbf{x}_0$  is not orthogonal to  $\mathbf{v}_1$ ; that is, we must have  $\mathbf{x}_0 \cdot \mathbf{v}_1 \neq 0$ . Of course, without knowing  $\mathbf{v}_1$  in advance, the best we can do is that hope our guess for  $\mathbf{x}_0$  is not!
- What if  $|\lambda_1| > 1$ ? While the power iteration should still approach the largest eigenvector, every successive iteration will result in a larger and larger vector. If this happens too quickly, we might approach the maximum value floating point number we can store in memory before convergence is reached. To avoid this, we can simply *normalise*  $A^n \mathbf{x}$  at each iteration — this is called the **normalised power iteration**.

► It can be shown that the power iteration converges **linearly**, that is,  $\exists C > 0$  such that

$$|\lambda_1^{(n+1)} - \lambda_1| \leq C |\lambda_1^{(n)} - \lambda_1|$$

where  $\lambda_1^{(n)}$  is the  $n$ th iterative approximation to the dominant eigenvalue.

After using the power iteration to approximate the largest (normalised) eigenvector, we can employ the **Rayleigh quotient** to determine the associated largest eigenvalue:

- If using the normalised power iteration, then

$$\mathbf{x}_n^* \cdot \mathbf{x}_n = 1.$$

$$\lim_{n \rightarrow \infty} \frac{\mathbf{x}_n^\dagger A \mathbf{x}_n}{\mathbf{x}_n^* \cdot \mathbf{x}_n} = \lambda_1. \quad (7.18)$$

### Example 7.1 Power iteration (Fortran)

- You might alternatively see the Rayleigh quotient written as

$$\mathbf{x}^* \cdot A \mathbf{x}$$

depending on whether the author is using vector dot-product notation, or column-vector/matrix multiplication notation.

**Solution:** Before converting it into a Python program, how do we know when to stop the iteration? Since we know that the Rayleigh quotient should converge to the dominant eigenvalue, we can use a **while** loop, and exit the loop once successive Rayleigh quotients differ by a small enough quantity:

```
program poweriteration
    implicit none
    complex(8) :: A(3,3), x(3), RQnew, RQold

    A(1,:) = [(4.d0,0.d0), (0.d0,-1.d0), (2.d0,0.d0)]
    A(2,:) = [(0.d0,1.d0), (2.d0, 0.d0), (2.d0,7.d0)]
    A(3,:) = [(2.d0,0.d0), (2.d0,-7.d0), (-2.d0,0.d0)]

    ! choose the starting vector
    x = [1., 1., 1.]
    x = x/norm(x)
    RQnew = rayleigh_quotient(A,x)
    RQold = 0

    ! power iteration
    do while (abs(RQnew-RQold)>1.d-6)
        RQold = RQnew
        x = matmul(A,x)
        x = x/norm(x)
        RQnew = rayleigh_quotient(A,x)
    end do

    write(*,*)"Dominant eigenvalue:", RQnew
    write(*,*)"Dominant eigenvector:", x

contains
    ! calculate the norm of a vector
    function norm(x)
```

```

    complex(8), intent(in) :: x(:)
    real(8) :: norm
    norm = sqrt(sum(abs(x)**2))
end function norm
! calculate the Rayleigh quotient
function rayleigh_quotient(A,x)
    complex(8), intent(in) :: A(:,:,), x(:)
    complex(8) :: rayleigh_quotient
    rayleigh_quotient = dot_product(x,matmul(A,x))
    rayleigh_quotient = rayleigh_quotient/dot_product(x,x)
end function rayleigh_quotient
end program poweriteration

```

### Example 7.2 Power iteration (Python)

Use the normalised power iteration to find the dominant eigenvalue and associated eigenvector of the Hermitian matrix

$$A = \begin{bmatrix} 4 & -i & 2 \\ i & 2 & 2+7i \\ 2 & 2-7i & -2 \end{bmatrix}$$

**Solution:** Before converting it into a Python program, how do we know when to stop the iteration? Since we know that the Rayleigh quotient should converge to the dominant eigenvalue, we can use a while loop, and exit the loop once successive Rayleigh quotients differ by a small enough quantity:

```

#!/usr/bin/env python3
import numpy as np

# function to calculate the Rayleigh quotient
def rayleigh_quotient(A,x):
    return np.dot(x, np.dot(A, x))/np.dot(x,x)

# function to normalise a vector
def normalise(x,eps=1e-10):
    N = np.sqrt(np.sum(abs(x)**2))
    if N < eps: # in case it is the zero vector!
        return x
    else:
        return x/N

A = np.array([[4, -1j, 2],
              [1j, 2, 2+7j],
              [2, 2-7j, -2]])

# choose the starting vector

```

```

x = normalise(np.array([1, 1, 1]))
RQnew = rayleigh_quotient(A, x)
RQold = 0

# perform the power iteration
while np.abs(RQnew-RQold) > 1e-6:
    RQold = RQnew
    x = normalise(np.dot(A, x))
    RQnew = rayleigh_quotient(A, x)

print("Dominant eigenvector:", x)
print("Dominant eigenvalue: {:.5f}".format(RQnew))

```

---

**Problem**

---

The example above returns a dominant eigenvalue of  $\lambda_1 = 8.45188$ , and dominant eigenvector

$$\mathbf{v}_1 = (0.3398 - 0.2345, 0.4913 + 0.5107i, 0.5011 - 0.2762i).$$

Have a go running the code snippets to verify this result.

Next, try modifying the above example to count how many loop iterations occur before convergence. Is it larger or smaller than you had expected?

An alternative to using the Rayleigh quotient to determine whether convergence is achieved is the convergence criterion

$$|x_{n+1} - x_n| \leq \epsilon.$$

This is especially useful if we do not need to compute the eigenvalue, and can save some computational time. Modify the code above to use this convergence criterion, and compare the overall wall time to the original version.

---

**Google PageRank**

---

Everyday you are likely interacting with a company that uses the power iteration — Google! To determine which websites are more useful when presenting you with search results, Google uses a **network centrality** algorithm called PageRank. By representing internet sites as a network of nodes with hyperlinks as connected edges, it turns out that information containing the importance or centrality of each site is contained with the dominant eigenvector. The power iteration is thus ideally suited for Google's needs.

---

While simple, there are many caveats in using the power method. For starters, we are restricted to using it with matrices we know to be diagonalisable; for example, Hermitian or real symmetric matrices. Secondly, the rate of convergence is entirely dependent on how close our starting vector  $\mathbf{x}_0$  is to the eigenvector  $\mathbf{v}_1$ . Finally, this only allows us to calculate the largest eigenvalue — usually we want to calculate all the eigenvalues, or specific eigenvalues! For example, if we are attempting to find the ground state in quantum mechanics, we are searching for the *smallest* magnitude eigenvalue.

### Power Method Convergence

We mentioned in the sidebar in Sect. 7.2 that the power method exhibits linear convergence for general diagonalizable matrices. What happens if we restrict ourselves to Hermitian or real symmetric matrices? Let's consider the simplest possible Hermitian matrix, a diagonal matrix.

### Hermitian Power Iteration Convergence

Consider the diagonal matrix

$$A = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \quad (7.19)$$

with  $\lambda_1 > \lambda_2$ , and the normalised vector  $\mathbf{x}_0 = (a, b)$  where  $\sqrt{a^2 + b^2} = 1$ . Performing  $n$  iterations of the power iteration, we have

$$A^n \mathbf{x}_0 = \begin{bmatrix} \lambda_1^n & 0 \\ 0 & \lambda_2^n \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = (a\lambda_1^n, b\lambda_2^n). \quad (7.20)$$

Calculating the Rayleigh quotient then gives us an estimate for the dominant eigenvalue,

$$\lambda_1^{(n)} = \lambda_1 \frac{a^2 + b^2(\lambda_2/\lambda_1)^{2n+1}}{a^2 + b^2(\lambda_2/\lambda_1)^{2n}}. \quad (7.21)$$

Using this to calculate the convergence limit,

$$\lim_{n \rightarrow \infty} \frac{|\lambda_1^{(n+1)} - \lambda_1|}{|\lambda_1^{(n)} - \lambda_1|} = \lim_{n \rightarrow \infty} \frac{a^2 + b^2(\lambda_2/\lambda_1)^{2n}}{a^2 + b^2(\lambda_2/\lambda_1)^{2n+2}} \left( \frac{\lambda_2}{\lambda_1} \right)^2 = \left( \frac{\lambda_2}{\lambda_1} \right)^2 \quad (7.22)$$

since  $\lim_{n \rightarrow \infty} r^n = 0$  if  $0 < r < 1$ . Therefore, the power iteration converges **quadratically**. This result generalises for any Hermitian or real symmetric matrices.

### 7.2.1 Inverse Power Iteration

With one simple trick, we can, in fact, use the power iteration to find the smallest eigenvalue of a matrix. This relies on a simple, well-known property of the eigenvalue. Consider again a matrix  $A \in \mathbb{C}^{N \times N}$ , that has the following properties:

- it is diagonalisable, or equivalently, it has  $N$  linearly independent eigenvectors  $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_N$ ,
- its eigenvalues can be ordered like  $|\lambda_1| > |\lambda_2| \geq \dots > |\lambda_N|$ .

From linear algebra, we know that since  $A$  is diagonalisable, it must be invertible, and as a result,

$$A\mathbf{x}_i = \lambda_i \mathbf{v}_i \Rightarrow \lambda_i^{-1} \mathbf{x}_i = A^{-1} \mathbf{x}_i, \quad (7.23)$$

i.e. the eigenvalues of  $A^{-1}$  are the *inverse* of the eigenvalues of  $A$ , with the **same** eigenvectors. Furthermore, we know that

$$|\lambda_1| > |\lambda_2| \geq \dots > |\lambda_N| \Leftrightarrow |\lambda_1^{-1}| < |\lambda_2^{-1}| \leq \dots \leq |\lambda_N^{-1}|. \quad (7.24)$$

This means, as long as we are able to easily invert  $A$ , we can use the normalised power method above on  $A^{-1}$  to calculate  $\lambda_N$  — the smallest magnitude eigenvalue of  $A$ , and the dominant eigenvalue of  $A^{-1}$ .

#### Problem

Modify the code from the previous example to use the inverse power iteration to find the *smallest* magnitude eigenvalue and eigenvector of matrix  $A$ .

When working with small  $2 \times 2$  or  $3 \times 3$  matrices, it is relatively simple to determine the matrix determinant and the inverse. In practice, however, matrix inversion is a difficult numerical computation, is prone to error, and can be quite unstable. A more common approach than direct matrix inversion is to view the inverse power iteration as a **system of linear equations** to be solved involving  $A$ :

1. Choose a normalised starting vector  $\mathbf{x}_0$  where  $|\mathbf{x}_0| = 1$ ;
2. Solve the system of linear equations  $A\mathbf{x}_{n+1} = \mathbf{x}_n$  to determine the next vector in the iteration, and normalise this vector;
3. Repeat step 2 until convergence is achieved,  $|\mathbf{x}_{n+1} - \mathbf{x}_n| \leq \epsilon$  for some small value of  $\epsilon$ . The Rayleigh quotient  $\mathbf{x}_n^\dagger A \mathbf{x}_n$  will provide the value of the smallest magnitude eigenvector of  $A$ .

Common numerical methods for solving linear systems of equations include Jacobi iteration and factorisation schemes such as QR and LU decompositions. We won't go into matrix factorisation and decompositions here — see the further reading at the end of the chapter for where to get more information.

### 7.2.2 Rayleigh Quotient Iteration

Recall that the convergence of the power iteration is completely dependent on the ratio  $|\lambda_2/\lambda_1|$ , where  $\lambda_2$  is the second highest eigenvalue, and our initial starting guess for the eigenvector  $\mathbf{x}_0$ . We can attempt to mitigate the first issue via the following observation from linear algebra; when shifting the matrix  $A$  by a constant  $\mu \in \mathbb{C}$ ,  $A - \mu I$ , the eigenvalues are now given by

$$\lambda_1 - \mu, \lambda_2 - \mu, \dots, \lambda_N - \mu. \quad (7.25)$$

If we manage to choose a  $\mu$  close to an eigenvalue of  $\lambda_i$ , then the corresponding eigenvalue of the shifted matrix,  $\lambda_i - \mu$ , will be close to zero and have the smallest magnitude:

$$|\lambda_1 - \mu| > |\lambda_2 - \mu| \geq \dots \geq |\lambda_i - \mu|. \quad (7.26)$$

As a result, we can use the inverse iteration to determine  $\lambda_i$ ; the closer our initial guess of  $\mu$  is to  $\lambda_i$ , the smaller the ratio  $|(\lambda_i - \mu)/(\lambda_{i+1} - \mu)|$  where  $|\lambda_{i+1} - \mu|$  is the second smallest magnitude eigenvalue, and the fewer iterations needed for convergence.

We can reduce the time taken to convergence even further by replacing  $\mu$  at each iteration by a *better* approximation to  $\lambda_i$ , namely, the Rayleigh quotient.

The Rayliegh quotient iteration therefore looks like this:

1. Choose a normalised starting vector  $\mathbf{x}_0$  where  $|\mathbf{x}_0| = 1$ , and an initial eigenvalue guess  $\mu_0$ . This can simply be the current Rayleigh quotient  $\mu_0 = \mathbf{x}^\dagger A \mathbf{x}$ .
2. Solve the system of linear equations  $(A - \mu I)x_{n+1} = x_n$  to determine the next vector in the iteration, and normalise the vector. This can be done by solving the linear system using a numeric method, or by finding the inverse and applying  $(A - \mu I)^{-1}x_n = x_{n+1}$ .
3. Determine the next eigenvalue “guess” by calculating the Rayleigh quotient,  $\mu_{n+1} = \mathbf{x}_{n+1}^\dagger A \mathbf{x}$
4. Return to step 2 and repeat until convergence is achieved; that is,  $|\mu_{n+1} - \mu_n| \leq \epsilon$  for some  $\epsilon$ . The resulting eigenvalue  $\lambda_i$  and associated eigenvector  $\mathbf{v}_i$  will be given by  $\mu_n$  and  $x_n$  respectively.

Unfortunately, while trying to fix the convergence of the power iteration, we have hit another issue — now, we also need to solve a system of linear equations or calculate the matrix inverse at every iteration! Thus, the Rayleigh quotient iteration is best suited when the initial guess  $\mu_0$  is *very* close to the true eigenvalue, and the low number of iterations required offsets the added complexity of calculating the matrix inverse.

► If we ignore the added requirement of having to calculate an inverse matrix with each iteration, the Rayleigh quotient method actually converges **cubically** for Hermitian and real symmetric matrices.

On first glance, it appears that the Rayleigh quotient method might allow us to calculate *all* the eigenvalues, however, it's not as clear cut as that. While we can choose various initial guesses of  $\mu_0$  to try and cover the entire eigenvalue spectrum, we have no idea whether we have missed particular eigenvectors — the Rayleigh quotient method is quite similar to the Newton–Raphson root finding method, in that regard.

### 7.2.3 Method of Deflation

At the very beginning of this chapter, we noted that the only major pitfall of the power method is accidentally choosing a starting vector  $\mathbf{x}_0$  that just happens to be orthogonal to the dominant eigenvector, destroying our initial assumption that we can expand the vector in the eigenbasis including the dominant eigenvector.

But what if we can use this to our advantage in determining the non-dominant eigenvalues and eigenvectors? Let's see what happens. Assume we have performed the power iterative method on an Hermitian matrix  $A \in \mathbb{C}^{N \times N}$ , and have determined (to relatively small error) the dominant eigenvalue  $\lambda_1$  and dominant eigenvector  $\mathbf{v}_1$ . We now want to choose a starting guess  $\mathbf{x}_0$  that is **deliberately** orthogonal to  $\mathbf{v}_1$ , such that

$$x_n = c_2 \lambda_2^n \mathbf{v}_2 + \lambda_2^n \sum_{i=3}^N c_i \left( \frac{\lambda_i}{\lambda_1} \right)^n \mathbf{v}_i \quad (7.27)$$

i.e. now  $\lambda_2$  is the quasi-dominant eigenvalue! But how do we do this? Consider the matrix

$$\mathcal{A} = A - \lambda_1 \mathbf{v}_1 \mathbf{v}_1^\dagger. \quad (7.28)$$

Applying this to the dominant eigenvector,

$$A\mathbf{v}_1 = A\mathbf{v}_1 - \lambda_1 \mathbf{v}_1 (\mathbf{v}_1^\dagger \mathbf{v}_1) = \lambda_1 \mathbf{v}_1 - \lambda_1 \mathbf{v}_1 = 0, \quad (7.29)$$

since we have assumed that the eigenvectors are normalised ( $\mathbf{v}_1^\dagger \mathbf{v}_1 = 1$ ). Meanwhile, applying  $\mathcal{A}$  to the remaining eigenvectors,

$$A\mathbf{v}_i = A\mathbf{v}_i - \lambda_1 \mathbf{v}_1 (\mathbf{v}_1^\dagger \mathbf{v}_i) = \lambda_i \mathbf{v}_i, \quad i \neq 1, \quad (7.30)$$

since the eigenvectors of an Hermitian matrix are orthogonal,  $\mathbf{v}_1^\dagger \mathbf{v}_i = 0$  for  $i \neq 1$ . Thus,  $\mathcal{A}$  has the same eigenspectrum as  $A$  but with  $\mathbf{v}_1$  projected out — so after determining the dominant eigenvalue  $\lambda_1$  and eigenvector  $\mathbf{v}_1$ , we can simply apply the power method to  $\mathcal{A}$  to determine  $\lambda_2$  and  $\mathbf{v}_2$ .

This follows for all remaining eigenvalues and eigenvectors, and therefore, using the method of deflation to calculate  $\lambda_i$  and  $\mathbf{v}_i$  works as follows:

- ▶ This is an example of an **orthogonalisation algorithm**.

- ▶ The method of deflation can also be used with the inverse iteration or Rayleigh quotient iteration methods.

1. Project out the previously calculated dominant eigenvector by calculating  $\mathcal{A}_i = \mathcal{A}_{i-1} - \lambda_{i-1} \mathbf{v}_{i-1} \mathbf{v}_{i-1}^\dagger$ .
2. Choose a normalised starting vector  $\mathbf{x}_0$ .
3. Apply the power iteration  $\mathcal{A}_i \mathbf{x}_n = \mathbf{x}_{n+1}$  to determine the next vector in the iteration, and normalise the vector.
4. Return to step 3 and repeat until convergence:  $|\mathbf{x}_{n+1} - \mathbf{x}_n| \leq \epsilon$  for some  $\epsilon$ . The resulting eigenvector  $\mathbf{v}_i$  is given by  $\mathbf{x}_n$ , and the eigenvalue  $\lambda_i = \mathbf{x}_n^\dagger \mathcal{A}_i \mathbf{x}_n$  is given by the Rayleigh quotient.

Note that the error  $\epsilon$  in calculating the first dominant eigenvalue propagates through to the second eigenvalue implicitly when calculating  $\mathcal{A}$ ; these errors can accumulate with each successive eigenvalue calculation, and result in significant error or stability issues.

### 7.3 Krylov Subspace Techniques

- ▶ Aleksey Nikolaevich Krylov (1863–1945) was a Russian engineer and mathematician.
- ▶ The Krylov basis contains the first  $r - 1$  power iterations, and so would be expected to converge to the dominant eigenvector of  $A$ . In fact, further analysis uncovers that increasing  $r$  provides better and better approximations to the ‘gradient’ of the Rayleigh quotient.

To avoid the accumulation of error that occurs when calculating successive eigenvalues in the method of deflation, it would be useful to have a method that approximates *all* eigenvalues with each iteration — this is the essence of decomposition or factorisation methods. A particularly popular set of methods are referred to as **Krylov subspace techniques**, as they make use of the so-called **Krylov** vector space, an  $n$ th order linear subspace.

#### The Krylov Subspace.

**Definition 7.10.** For a matrix  $A \in \mathbb{C}^{N \times N}$  and arbitrary non-zero vector  $\mathbf{x} \in \mathbb{C}^N$ , the  $r$ th order Krylov subspace is

$$\mathcal{K}_r(A, \mathbf{x}) = \text{span}\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \dots, A^{r-1}\mathbf{x}\}. \quad (7.31)$$

Any vector within the Krylov subspace can be represented in terms of coefficients of the Krylov basis vectors; for instance, if  $\mathbf{u} \in \mathcal{K}_r$ , then

$$\mathbf{u} = c_1\mathbf{x} + c_2A\mathbf{x} + \dots + c_rA^{r-1}\mathbf{x} = \sum_{i=1}^r c_iA^{i-1}\mathbf{x}. \quad (7.32)$$

However, the Krylov subspace isn’t an ideal vector basis representation, as its constituent vectors all tend to be oriented in the same direction, due to repeated application of  $A$ . Instead, we can use an orthogonalisation process (such as Gram–Schmidt orthogonalisation) to create an **orthonormal** basis for the Krylov subspace:

$$q_1 = \frac{\mathbf{x}}{|\mathbf{x}|},$$

$$q_{i+1} = Aq_i - \sum_{j=1}^i q_j(q_j^\dagger Aq_i).$$

That is, we are projecting the vector against the existing orthonormal basis, and subtracting it from the vector.

$$\mathcal{Q}_r = \{\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_r\} \quad (7.33)$$

where  $\mathcal{K}_r(A, \mathbf{x}) = \text{span}(\mathcal{Q}_r)$ . If  $A$  is Hermitian, then we call  $\mathcal{Q}_r$  the **Lanczos basis**; otherwise, they are referred to as the **Arnoldi basis**. The remarkable thing about the Krylov subspace can be seen if we wish to orthogonalise the *next* largest Krylov subspace  $\mathcal{K}_{r+1}$ . To start with, we simply set  $\mathbf{q}_1 = \mathbf{x}/|\mathbf{x}|$ :

$$\begin{aligned} \mathcal{K}_{r+1}(A, \mathbf{x}) &= \text{span}(\{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \dots, A^r\mathbf{x}\}) \\ &= \text{span}(\{\mathbf{q}_1, A\mathbf{q}_1, A^2\mathbf{q}_1, \dots, A^r\mathbf{q}_1\}). \end{aligned} \quad (7.34)$$

Now, we know that  $A\mathbf{q}_1 \in \mathcal{K}_2$ , so we can replace it with the expansion  $A\mathbf{q}_1 = c_1\mathbf{q}_1 + c_2\mathbf{q}_2$ :

$$\begin{aligned} \mathcal{K}_{r+1}(A, \mathbf{x}) &= \text{span}(\{\mathbf{q}_1, c_1\mathbf{q}_1 + c_2\mathbf{q}_2, A(c_1\mathbf{q}_1 + c_2\mathbf{q}_2), \dots, A^{r-1}(c_1\mathbf{q}_1 + c_2\mathbf{q}_2)\}) \\ &= \text{span}(\{\mathbf{q}_1, \mathbf{q}_2, A\mathbf{q}_2, \dots, A^{r-1}\mathbf{q}_2\}) \end{aligned} \quad (7.35)$$

where we have used the property that  $\text{span}(\{\mathbf{v}, \alpha\mathbf{v} + \beta\mathbf{u}\}) = \text{span}(\{\mathbf{v}, \mathbf{u}\})$ . Repeating this process for each successive element, we find that

$$\mathcal{K}_{r+1}(A, \mathbf{x}) = \text{span}(\{\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_r, A\mathbf{q}_r\}). \quad (7.36)$$

Therefore if we already know the orthonormal basis vectors  $\mathcal{Q}_r$  for  $\mathcal{K}_r$ , to calculate  $\mathcal{Q}_{r+1}$ , instead of having to orthogonalise  $A^r\mathbf{q}_1$  against the existing basis, we simply need to orthogonalise  $A\mathbf{q}_r$ ! This makes calculating successively larger Krylov bases significantly easier.

### 7.3.1 Lanczos Iteration

To understand the power of the Krylov subspace, lets look at one of the more popular methods used to approximate the eigenvalues of an Hermitian matrix — the **Lanczos iterative algorithm**. As its name suggests, it is an iterative procedure for calculating the vectors  $\mathbf{q}_i$  of the Lanczos basis. But, you might wonder, how does this connect to the eigenvectors and eigenvalues of the matrix? To see how, consider the following property of Hermitian matrices.

**Theorem 7.11.** *If  $A \in \mathbb{C}^{N \times N}$  is Hermitian, then  $\exists$  a unitary matrix  $Q \in \mathbb{C}^{N \times N}$ ,  $Q^\dagger Q = I$ , such that*

$$Q^\dagger A Q = T$$

where  $T$  is a **symmetric tridiagonal matrix**. That is,  $T_{ij} = 0$  for all  $i > j + 1$  and  $i < j - 1$ .

The proof of this follows from the fact that, since  $T$  is an Hermitian matrix (as  $Q^\dagger A Q$  must be) it is diagonalisable. It turns out that the matrix  $Q$  is simply constructed via the Lanczos basis.

**Theorem 7.12.** *The unitary matrix  $Q = [\mathbf{q}_1 \ \mathbf{q}_2 \ \dots \ \mathbf{q}_N]$ , with columns consisting of the orthonormal Lanczos basis vectors, tridiagonalises the Hermitian matrix  $A \in \mathbb{C}^{N \times N}$ .*

*Proof:* Consider the elements of the tridiagonal matrix,  $T_{ij} = \mathbf{q}_i^\dagger A \mathbf{q}_j$ . We know, from the definition of the Lanczos basis, that

$$A\mathbf{q}_j \in \mathcal{K}_i(A, \mathbf{x}) = \{\mathbf{q}_1, \dots, \mathbf{q}_{i-1}\} \quad \forall i > j + 1. \quad (7.37)$$

Since  $\mathbf{v}_i$  is constructed to be orthogonal to  $\mathcal{K}_i$ , we must have  $T_{ij} = \mathbf{q}_i^\dagger (A\mathbf{q}_j) = 0$ , for all  $i > j + 1$ . Since  $T$  is Hermitian, we also have  $T_{ij} = 0$  for all  $i < j - 1$ .

If we write the tridiagonal matrix  $T$  as

$$T = \begin{bmatrix} a_1 & b_1 & & & \\ b_1 & a_2 & \ddots & & \\ & \ddots & \ddots & b_{N-1} & \\ & & b_{N-1} & a_N & \end{bmatrix} \quad (7.38)$$

► **Cornelius Lanczos** (1893–1974) was a Hungarian mathematician and Physicist.

where  $a_i = \mathbf{q}_i^\dagger A \mathbf{q}_i$  and  $b_i = \mathbf{q}_{i+1}^\dagger A \mathbf{q}_i$ , and noting that we can then write the equation  $AQ_N = Q_NT$  as

$$(A\mathbf{q}_1, A\mathbf{q}_2, \dots, A\mathbf{q}_N) \\ = (a_1\mathbf{q}_1 + b_1\mathbf{q}_2, b_1\mathbf{q}_1 + a_2\mathbf{q}_2 + b_2\mathbf{q}_3, \dots, b_{N-1}\mathbf{q}_{N-1} + a_N\mathbf{q}_N), \quad (7.39)$$

we can actually solve this directly for the Lanczos vectors:

$$b_1\mathbf{q}_2 = (A - a_1)\mathbf{q}_1, \quad (7.40a)$$

$$b_{i+1}\mathbf{q}_i = (A - a_{i-1})\mathbf{q}_{i-1} - b_{i-2}\mathbf{q}_{i-2}, \quad 2 < i < N, \quad (7.40b)$$

$$0 = (A - a_N)\mathbf{q}_N - b_{N-1}\mathbf{q}_{N-1}. \quad (7.40c)$$

We know that we can choose  $\mathbf{q}_1$  to be any normalised vector; therefore, let's introduce the additional coefficient  $b_0$  such that  $b_0\mathbf{q}_1 = \mathbf{x}_0$ . We can therefore write this system as

$$b_{i-1}\mathbf{q}_i = \mathbf{x}_{i-1}, \quad 1 \leq i < N, \quad (7.41a)$$

$$0 = \mathbf{x}_N, \quad (7.41b)$$

where  $\mathbf{x}_0$  is a random vector,  $\mathbf{x}_i = (A - a_i)\mathbf{q}_i - b_{i-1}\mathbf{q}_{i-1}$  for  $1 \leq i \leq N$ , and  $a_i = \mathbf{q}_i^\dagger A \mathbf{q}_i$ . Furthermore, for the  $\mathbf{q}_i$  to be normalised, we must have  $b_i = |\mathbf{x}_i| > 0$  for  $0 \leq i < N$ . Note that, since  $b_0 \neq 0$  by this definition, in order for this to reduce to the proper form for  $\mathbf{q}_2$ , we introduce  $\mathbf{q}_0 = 0$  for convenience. This gives rise to the highly efficient, yet prone to instability, **Lanczos algorithm**:

1. Start with a randomly chosen eigenvector  $\mathbf{x}_0$ .
2. Calculate  $b_0 = |\mathbf{x}_0|$  and set  $\mathbf{q}_0 = 0$ .
3. For  $1 \leq i \leq N$ :
  - (a) Calculate the  $i$ th Lanczos vector:  $\mathbf{q}_i = \mathbf{x}_{i-1}/b_{i-1}$ .
  - (b) Calculate  $a_i = \mathbf{q}_i^\dagger A \mathbf{q}_i$ .
  - (c) Orthogonalise the next Krylov vector  $A\mathbf{q}_i$ :

$$\mathbf{x}_i = (A - \alpha_i)\mathbf{q}_i - b_{i-1}\mathbf{q}_{i-1}$$

- (d) If  $i < N$ , calculate  $b_i = |\mathbf{x}_i|$ .

With every  $i$ th loop iteration of this algorithm, we calculate the next Lanczos vector  $\mathbf{q}_i$ , as well as tridiagonalising the Hermitian matrix  $A \in \mathbb{C}^{N \times N}$  in the Krylov basis  $\mathcal{K}_{i-1}(A, \mathbf{x})$  to give  $Q_i^\dagger A Q_i = T \in \mathbb{C}^{i \times i}$ . As  $i \rightarrow N$ ,  $T' \rightarrow T$ , and we slowly recover the exact tridiagonal matrix.

We still haven't calculated the eigenvalues and eigenvectors, however! Unfortunately, this is the ending point of the Lanczos algorithm; it has

► The Lanczos algorithm improves on Eq. 7.36 by using the **symmetry of Hermitian matrices**.

Now, to calculate  $Q_{r+1}$  given  $Q_r$ , we need to orthogonalise  $A\mathbf{q}_r$  against *only*  $\mathbf{q}_{r-1}$  and  $\mathbf{q}_{r-2}$ .

► As  $b_i = |\mathbf{x}| > 0$  and  $a_i \in \mathbb{R}$  since the diagonal of an Hermitian matrix must always be real, it follows that  $T$  is also a real, symmetric matrix.

► Due to the relatively high instability of the Lanczos algorithm, it is particularly sensitive to round-off or precision error due to floating-point number representation.

One solution, at the cost of additional computation at each iteration, is to explicitly re-orthonormalise the full basis set  $\{\mathbf{q}_1, \dots, \mathbf{q}_i\}$  at each iteration.

provided an efficient method for reducing the Hermitian matrix down to a tridiagonal matrix, to which we now apply a specific eigenvalue method. The tridiagonal matrix, being similar to the Hermitian matrix, shares the same eigenvalues, whilst, at the same time, taking fewer iterations to find them due to it being slightly ‘more’ diagonalised.

However, that does not explain the ubiquity of the Lanczos algorithm and Krylov subspace techniques! Need a hint? It turns out, we don’t *need* to iterate a full  $N$  times in the Lanczos algorithm; in fact, performing only  $k$  iterations, and then using an eigenvalue solver on  $T' \in \mathbb{C}^{k \times k}$ , provides a surprisingly accurate estimate for the extreme (largest and smallest) eigenvalues. As a general rule of thumb, if you sort the (real) eigenvalues such that

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_N, \quad (7.42)$$

then  $k$  iterations of the Lanczos algorithm provides a satisfactory approximation of  $\lambda_1$  to  $\lambda_{\sim 0.3k}$ , and  $\lambda_{\sim (N-0.3k)}$  to  $\lambda_N$ .

### Example 7.3 Lanczos tridiagonalisation (Fortran)

Using the Lanczos algorithm, tridiagonalise the Hermitian matrix

$$\begin{bmatrix} 2 & -1+i & -0.5i & 4.25 \\ -1-i & 4 & 1 & 7 \\ 0.5i & 1 & 2 & -1+2i \\ 4.25 & 7 & -1-2i & 1.4 \end{bmatrix}$$

**Solution:**

```
program lanczos
    implicit none
    complex(8), parameter :: j=(0.d0,1.d0)
    real(8) :: Id(4,4), T(4,4), a(1:4), b(0:3)
    complex(8) :: H(4,4), x(4), qi(4), qii(4)
    integer :: i, N

    ! set up the matrix H
    H(1,:) = [2.d0+j, -1.d0+j, -0.5d0*j, 4.25d0+j]
    H(2,:) = [-1.d0-j, 4.d0+j, 1.d0+j, 7.d0+j]
    H(3,:) = [0.5d0*j, 1.d0+j, 2.d0+j, -1.d0+2.d0*j]
    H(4,:) = [4.25d0+j, 7.d0+j, -1.d0-2.d0*j, 1.4d0+j]
    N = size(H,1)

    ! create the identity matrix
    Id = 0.d0
    do i=1,N
        Id(i,i) = 1.d0
    end do
```

- ▶ Since the Krylov basis is composed of power iterations, we can place an upper bound on Krylov subspace methods — convergence should be no worse than the equivalent power iteration.

- ▶ Remember that, in Fortran, `dot_product(x,y)` performs  $x^* \cdot y$  if the arguments are both `complex`.

```

! random starting vector. random_number only accepts
! real arrays, so we will borrow the a array
call random_number(a)
x = a

! initialise coefficients
qii = 0.d0; a = 0.d0; b = 0.d0
b(0) = norm(x) ! since we discard b_0, b is indexed 0:3

! Lanczos iteration
do i=1,N
    ! calculate q_i
    qi = x/b(i-1)
    ! append a_i = q_i*H.q_i to the list
    a(i) = dot_product(qi, matmul(H, qi))
    ! orthogonalise A.q_i
    x = matmul(H-a(i)*Id, qi) - b(i-1)*qii
    ! append b_i = |x| if i<N
    if (i<N) b(i) = norm(x)
    ! store qi as qii for the next iteration
    qii = qi
end do

! construct the tridiagonal matrix
T = 0.d0
do i=1,N
    T(i,i) = a(i)
    if (i<N) then
        T(i,i+1) = b(i)
        T(i+1,i) = b(i)
    end if
end do

contains
    function norm(x)
        complex(8), intent(in) :: x(:)
        real(8) :: norm
        norm = sqrt(sum(abs(x)**2))
    end function norm
end program lanczos

```

#### Example 7.4 Lanczos tridiagonalisation (Python)

Using the Lanczos algorithm, tridiagonalise the Hermitian matrix

$$\begin{bmatrix} 2 & -1+i & -0.5i & 4.25 \\ -1-i & 4 & 1 & 7 \\ 0.5i & 1 & 2 & -1+2i \\ 4.25 & 7 & -1-2i & 1.4 \end{bmatrix}$$

Solution:

```
#!/usr/bin/env python3
import numpy as np
from numpy.linalg import norm
from numpy.random import random

A = np.array([[2, -1+1j, -0.5j, 4.25],
              [-1-1j, 4, 1, 7],
              [0.5j, 1, 2, -1+2j],
              [4.25, 7, -1-2j, 1.4]])
N = len(A)
I = np.identity(N) # identity matrix

# random starting vector
x = random(N)

# initialise coefficients
qii = 0          # q_{i-1}
a = [0]          # a_0
b = [norm(x)]   # b_0

# Lanczos iteration
for i in range(1,N+1):
    # calculate q_i:
    qi = x/b[i-1]
    # append a_i = q_i * A * q_i to the list
    a.append( np.conj(qi).dot(A.dot(qi)) )
    # orthogonalise A.q_i
    x = np.dot((A-a[i]*I), qi) - b[i-1]*qii
    # append b_i = |x| if i < N
    if i < N:
        b.append(norm(x))
    # store qi as qii for the next iteration
    qii = qi

# drop initial values, and remove any small
# complex round-off error present
a = np.real_if_close(a[1:])
b = np.real_if_close(b[1:])

# construct the tridiagonal matrix
T = np.diag(a) + np.diag(b,1) + np.diag(b,-1)
```

## 7.4 Stability and the Condition Number

One thing we haven't touched on yet is the **stability** of the eigenvalue numerical methods we have considered. Cast your mind back to when we discussed initial value problems and ODE solvers; the stability of the particular method, coupled with how **stiff** the differential equation was, determined which method was best suited for the problem at hand. The same is true for eigenvalue solvers, but how can we quantify this?

As we did when we were considering differential equation methods, let's perturb the matrix in the eigenvalue equation  $A\mathbf{v}_i = \lambda_i \mathbf{v}_i$  by  $|\delta A| \ll 1$ :

$$(A + \delta A)(\mathbf{v}_i + \delta \mathbf{v}) = (\lambda_i + \delta \lambda_i)(\mathbf{v}_i + \delta \mathbf{v}). \quad (7.43)$$

Here,  $|\delta \lambda_i| \ll 1$  and  $|\delta \mathbf{v}| \ll 1$  are the resulting perturbations in the eigenvalue and eigenvector respectively, due to the perturbation in  $A$ . Expanding this and substituting it in the eigenvalue equation, we get

$$\delta A \mathbf{v}_i + A \delta \mathbf{v}_i = \delta \lambda_i \mathbf{v}_i + \lambda_i \delta \mathbf{v}, \quad (7.44)$$

where we have neglected the quadratic error terms  $\delta A \delta \mathbf{v}$  and  $\delta \lambda_i \delta \mathbf{v}$ .

Before we can continue, we need to introduce a new concept — left eigenvectors.

### Left Eigenvectors

Rather than define the eigenvalue equation as we did before, by post-multiplying  $A$  by  $\mathbf{v}$ , we can instead construct an equivalent eigenvalue equation by *pre-multiplying*  $A$  by the so-called left eigenvectors  $\mathbf{w}$ :

$$\mathbf{w}^\dagger A = \mu \mathbf{w}^\dagger. \quad (7.45)$$

Taking the conjugate transpose of both sides,

$$A^\dagger \mathbf{w} = \mu^* \mathbf{w}, \quad (7.46)$$

that is, the left eigenvectors of a matrix  $A$  are the *same* as the right eigenvectors of the matrix  $A^\dagger$ , and the characteristic equation is given by

$$|A^\dagger - \mu^* I| = 0. \quad (7.47)$$

Furthermore, we know that the determinant of a matrix is unchanged by the conjugate transpose; so, taking the conjugate transpose of the characteristic equation,

$$|A^\dagger - \mu^* I| = |(A^\dagger - \mu^* I)^\dagger| = |A - \mu I| = 0. \quad (7.48)$$

Notice anything? This is the *same* characteristic equation as the standard (right) eigenvalue equation — therefore the left eigenvalues  $\mu$  are simply the right eigenvalues  $\lambda$ .

If we denote  $\mathbf{w}_i$  and  $\mathbf{v}_i$  as the left and right eigenvectors of  $A$  respectively such that  $\mu_i = \lambda_i$ , then an even stronger result holds for diagonalisable matrices.

**Theorem 7.13.** If a matrix  $A \in \mathbb{C}^{N \times N}$  is a diagonalisable matrix, that is, it has  $N$  linearly independent eigenvectors, then its left and right eigenvectors are biorthonormal:

$$\mathbf{w}_i^\dagger \mathbf{v}_j = \mathbf{w}_i^* \cdot \mathbf{v}_j = \delta_{ij}.$$

Continuing with the stability analysis, let's pre-multiply the perturbed eigenvector equation by the left eigenvector  $\mathbf{w}_i$  corresponding to the same eigenvalue  $\lambda_i$  as the right eigenvector  $\mathbf{v}_i$ :

$$\mathbf{w}_i^\dagger \delta A \mathbf{v}_i + (\mathbf{w}_i^\dagger A) \delta \mathbf{v} = \delta \lambda_i \mathbf{w}_i^* \cdot \mathbf{v}_i + \lambda_i \mathbf{w}_i^* \cdot \delta \mathbf{v}. \quad (7.49)$$

Substituting in the left eigenvector equation  $\mathbf{w}_i^\dagger A = \lambda_i \mathbf{w}_i^\dagger$ , and rearranging, gives

$$\delta \lambda_i = \frac{\mathbf{w}_i^\dagger \delta A \mathbf{v}_i}{\mathbf{w}_i^* \cdot \mathbf{v}_i}. \quad (7.50)$$

If we assume that the left and right eigenvectors have been normalised such that  $|\mathbf{v}| = |\mathbf{w}| = 1$ , then taking the norm of both sides,

$$|\delta \lambda_i| = C_i |\delta A|, \quad C_i = \frac{1}{|\mathbf{w}_i^* \cdot \mathbf{v}_i|}. \quad (7.51)$$

The constant  $C_i$  is known as the **condition number** of eigenvalue  $\lambda_i$ , and tells us how suited the matrix  $A$  is to numerical eigenvalue computation of  $\lambda_i$ , regardless of the numeric method used. That is, it provides us an upper bound on the inherent expected error due to the condition of the matrix, separate from the truncation error of the numerical method and the round-off error due to floating-point precision. In a way, it is analogous to the idea of 'stiffness' in differential equation.

For the purposes of numerical eigenvalue methods, we can generalise the condition number for several broad cases:

- If the matrix  $A$  is diagonalizable, then it has a biorthonormal eigenbasis  $|\mathbf{w}_i^* \cdot \mathbf{v}_i| = 1$  and it follows that  $C_i = 1$  for all  $i$ . The matrix is **well-conditioned** and well suited for most numerical eigenvalue methods, as  $|\delta \lambda_i|$  is limited only to precision errors in the representation of  $A$ .
- The matrix  $A$  is **ill-conditioned** if  $C_i \gg 1$ ; in this case, slight deviations in the representation of  $A$  lead to large errors in the calculated eigenvalue  $\lambda_i$ . Special care needs to be taken choosing the eigenvalue solver, making sure to choose one with high stability; otherwise, a high number of iterations may be needed for accuracy.
- The matrix  $A$  is singular (non-invertible) if  $C \rightarrow \infty$ .

As a general rule of thumb, the number of digits accuracy lost due to the **conditioning** of the matrix, aside from the truncation and round-off error, can be approximated by  $\log_{10} C$ .

► The condition number can also be written as

$$C_i = \frac{1}{|\cos(\phi_i)|},$$

where  $\phi_i$  is the angle between the left and right eigenvectors of  $\lambda_i$ .

► The concept of the condition number can also be extended to differential equations!

## 7.5 Fortran: Using LAPACK

- ▶ If you’re using Python, skip forward to Sect. 7.6.
- ▶ LAPACK is a dense linear algebra package, and also comes with C and C++ bindings.
- Examples of other linear algebra libraries include ATLAS, Intel MKL, PETSc/SLEPc (for parallelisation).
- ▶ In this chapter, we will only be covering a small subset of what LAPACK is capable of. For more information and details, see the [LAPACK User Guide](#).

**Types represented by the first letter X:**

```
S: real
D: real(8)
C: complex
Z: complex(8)
```

**Table 7.1** The first letter X in a LAPACK subroutine determines the input/output data type

Luckily, if you’re using Fortran, then there is a standard linear algebra library called LAPACK (**L**inear **A**lgebra **P**ackage, pronounced *(l-ay-pack)*), that can do all the hard work for us (and, is super optimised to boot). On the other hand, using LAPACK in Fortran can sometimes be a little difficult, as the subroutines and arguments aren’t always self-explanatory.

In this chapter, I’ll introduce a few of the LAPACK subroutines we’ll need when we tackle the Schrödinger equation.

### 7.5.1 The Naming Scheme

One of the most perplexing parts of using LAPACK for newcomers is the naming scheme they use for their subroutines. When you get used to the LAPACK acronyms, it makes it easier to see at a glance what the subroutine does. Let’s run through them quickly.

All LAPACK routines are named using the following convention:

$$\text{XXXYYZZ(Z)} \quad (7.52)$$

Here, the first letter X represents the data type/precision of the subroutine input and output, see Table 7.1. The next two letters, YY, then indicate what matrix the subroutine is designed to work with – for the full list, see Table 7.2.

Finally, the last two or three letters ZZ(Z) refer to the specific algorithm to be implemented. For example, the subroutine DGESVX (where ZZZ=SVX) computes the solution to  $Ax = B$  where A is a real nonsymmetric matrix.

Now that we have that out the way, let’s move on to actually using LAPACK.

### 7.5.2 Finding Eigenvalues and Eigenvectors Using LAPACK

Once LAPACK is installed, it is relatively easy to link your Fortran code with it. As LAPACK is built using FORTRAN77, it provides **external subroutines**; simply call the required LAPACK subroutine in your code like so,

```
program main
    implicit none
    ! variable declarations
    complex :: arg1, ...
    real     :: i, j, arg6, ...

    ! program code
    call ZGEEV(arg1,arg2,...,argN)
end program main
```

| YY | Matrix type                                                       |
|----|-------------------------------------------------------------------|
| BD | Bidiagonal                                                        |
| DI | Diagonal                                                          |
| GB | General band                                                      |
| GE | General (a square/non-rectangular matrix with no symmetry)        |
| GC | Pairs of general matrices                                         |
| GT | General tridiagonal                                               |
| HB | Hermitian band ( <b>complex</b> only)                             |
| HE | Hermitian ( <b>complex</b> only)                                  |
| HG | Upper Hessenberg matrix and a tridiagonal matrix                  |
| HP | Hermitian matrix using packed storage ( <b>complex</b> only)      |
| HS | Upper Hessenberg                                                  |
| OP | Orthogonal matrix using packed storage ( <b>real</b> only)        |
| OR | Orthogonal ( <b>real</b> only)                                    |
| PB | Symmetric/Hermitian positive definite band                        |
| PO | Symmetric/Hermitian positive definite                             |
| PP | Symmetric/Hermitian positive definite matrix using packed storage |
| PT | Symmetric/Hermitian positive definite tridiagonal                 |
| SB | Symmetric band ( <b>real</b> only)                                |
| SP | Symmetric matrix using packed storage                             |
| ST | Symmetric tridiagonal ( <b>real</b> only)                         |
| SY | Symmetric                                                         |
| TB | Triangular band                                                   |
| TG | Pair of triangular matrices                                       |
| TP | Triangular matrix using packed storage                            |
| TR | Triangular                                                        |
| TZ | Trapezoidal                                                       |
| UN | Unitary ( <b>complex</b> only)                                    |
| UP | Unitary matrix using packed storage ( <b>complex</b> only)        |

**Table 7.2** Matrix types referred to by the two letters YY in the LAPACK subroutine name. Note that some subroutines only have matrices of a certain data type. For more details, see the [LAPACK User Guide](#)

making sure that you have declared the argument variables as the data type required by the LAPACK subroutine. Then, when compiling your code, you simply need to let the compiler know that the external subroutines you are using can be found in the LAPACK libraries. This is achieved by using the `-llapack` compiler flag:

```
gfortran example.f90 -o example -llapack
```

In this book, we are mainly interested in calculating the eigenvectors and eigenvalues of a matrix, so we'll focus on that for the rest of the chapter. However, if you are interested in doing more, check out the LAPACK user guide – or perhaps even read up on some linear algebra numerical methods and try implementing them yourself!

- ▶ Trying to decode the name? Flip back to Sect. 7.5.1 – for a complex (Z) general (GE) (i.e. non-symmetric) matrix, this subroutine gives us the eigenvalues and eigenvectors (EV).

In LAPACK, the most general eigenvalue/eigenvector solving subroutine is `ZGEEV`,<sup>1</sup> allowing us to compute the eigenvalues and eigenvectors for a non-symmetric complex matrix. In general, when you don't know in advanced what type of matrix you'll be working with, `ZGEEV` is a good all purpose solution.

Note that `ZGEEV` calculates both the **left eigenvectors** and the **right eigenvectors** of a matrix. Thus, the left and right eigenvectors have the same eigenvalues. In most cases, however, we are only interested in the right eigenvectors – it is possible to instruct `ZGEEV` to compute only the right eigenvectors.

### The General Case: `ZGEEV`

- ▶ `ZGEEV` first uses Krylov subspace techniques to reduce the matrix to its upper Hessenberg form (a matrix that is *almost* triangular, with zero below the sub-diagonal). It then uses the implicit QR algorithm to determine the eigenvalues and eigenvectors.

Check out Table 7.3 for the full details on the arguments of `ZGEEV`. You may notice that there are *a lot* of arguments – this is a side effect of LAPACK being built using FORTRAN77, and therefore not having access to modern Fortran90 features such as modules and assumed-shape arrays. This just means slightly more work on your part, due to the need to explicitly provide **leading dimensions** and **work arrays**.

Finally, have a look at Examples 7.5 and 7.6 to get a feel of how `ZGEEV` is used in Fortran.

#### Example 7.5 Calculating eigenvalues using `ZGEEV`

Using LAPACK, calculate the eigenvalues of the matrix

$$A = \begin{bmatrix} 8.7 - 4i & -0.7 + 8.5i & 8.6 - 0.8i \\ -3.4 - 5.6i & 1.6 + 8.4i & -9.4 - 0.6i \\ -1.6 + 1.2i & 3.6 + 5.6i & 8.0 - 3.3i \end{bmatrix}$$

**Solution:**

---

<sup>1</sup>Pronounced zuh-GEEV<sup>[citation needed]</sup>

---

```
call ZGEEV(JOBVL, JOBVR, N, A, LDA, W, VL, LDVL, VR, LDVR, WORK, LWORK, RWORK, INFO)
```

- **JOBVL – character, intent(in)**  
if '**N**', the left eigenvectors are not computed. if '**V**', they are computed.
  - **JOBVR – character, intent(in)**  
if '**N**', the right eigenvectors are not computed. If '**V**', they are computed.
  - **N – integer, intent(in)**  
The number of rows of matrix  $A$
  - **A – complex(8), dimension(N,N), intent(inout)**  
The complex matrix for which the eigenvalues (and eigenvectors) are to be computed.
  - **LDA – integer, intent(in)**  
The *leading dimension* of matrix  $A$ . In general,  $LDA=N$ .
  - **W – complex(8), dimension(N), intent(out)**  
A complex one-dimensional array containing the computed eigenvalues.
  - **VL – complex(8), dimension(LDVL,N), intent(out)**  
A complex array containing the computed left-eigenvectors if  $JOBVL='V'$ .  
The  $j$ th left-eigenvector is given by  $u(j)=VL(:,j)$ .
  - **LDVL – integer, intent(in)**  
The leading dimension of array  $VL$ . if  $JOBVL='V'$ , then in general  $LDVL=N$ .
  - **VR – complex(8), dimension(LDVR,N), intent(out)**  
A complex array containing the computed right-eigenvectors if  $JOBVR='V'$ .  
The  $j$ th right-eigenvector is given by  $v(j)=VR(:,j)$ .
  - **LDVR – integer, intent(in)**  
The leading dimension of array  $VR$ . if  $JOBVR='V'$ , then in general  $LDVR=N$ .
  - **WORK – complex(8), dimension(LWORK), intent(out)**  
A complex array used as the workspace. When complete,  $WORK(1)$  contains the optimal value for  $LWORK$ .
  - **LWORK – integer, intent(in)**  
The dimension of the work array.  $LWORK \geq \max(1, 2N)$ .
  - **RWORK – real(8), dimension(2\*N), intent(out)**  
Another workspace array.
  - **INFO – integer, intent(out)**  
Signifies the status of the algorithm. If  $INFO=0$ : success  
If  $INFO=-i < 0$ : the  $i$ th argument had an illegal value.  
If  $INFO=i > 0$ : Some eigenvalues and all eigenvectors failed to compute.
- 

**Table 7.3** ZGEEV returns the eigenvalues and, optionally, the left and/or right eigenvectors of a complex  $N \times N$  matrix  $A$ , using the QR decomposition

```

program eigenvalue
  implicit none

  complex(8) :: A(3,3), ev(3), dummy(1,1), work(6)
  integer :: i, info

  A(1,:) = [(8.7, -4.0), (-0.7, 8.5), (8.6, -0.8)]
  A(2,:) = [(-3.4, -5.6), (1.6, 8.4), (-9.4, -0.6)]
  A(3,:) = [(-1.6, 1.2), (3.6, 5.6), (8.0, -3.3)]

  call ZGEEV('N', 'N', 3, A, 3, ev, dummy, 1, dummy, 1, &
             & work, 6, work, info)

  if (info==0) then
    do i=1, 3
      write(*,'(f8.3,a,f8.3,a)') real(ev(i)), ' + ', &
        & aimag(ev(i)), 'i'
    enddo
  else
    write (*,*) "An error occurred"
  endif

end program eigenvalue

```

**Output:**

```

-2.030 + 11.133i
13.680 + -3.787i
 6.650 + -6.245i

```

**Notes:** since we are only interested in the eigenvalues, the first two arguments of ZGEEV are 'N', to indicate no calculation of the left and right eigenvectors is required. Therefore, rather than supply  $N \times N$  arrays to store the eigenvectors, we can simply pass  $1 \times 1$  dummy arrays.

**Example 7.6** Calculating eigenvalues and eigenvectors using ZGEEV

Using LAPACK, calculate the eigenvalues *and* eigenvectors of the matrix

$$A = \begin{bmatrix} 8.7 - 4i & -0.7 + 8.5i & 8.6 - 0.8i \\ -3.4 - 5.6i & 1.6 + 8.4i & -9.4 - 0.6i \\ -1.6 + 1.2i & 3.6 + 5.6i & 8.0 - 3.3i \end{bmatrix}$$

**Solution:**

```
program eigenvectors
    implicit none

    integer :: N, i, info
    complex(8), allocatable :: A(:, :), ev(:, :), vecL(:, :), &
        & vecR(:, :), work1(:, :), work2(:)

    N = 3

    allocate(A(N,N), ev(N), vecL(N,N), vecR(N,N), &
        & work1(2*N), work2(2*N))

    A(1,:) = [(8.7, -4.0), (-0.7, 8.5), (8.6, -0.8)]
    A(2,:) = [(-3.4, -5.6), (1.6, 8.4), (-9.4, -0.6)]
    A(3,:) = [(-1.6, 1.2), (3.6, 5.6), (8.0, -3.3)]

    call ZGEEV('V', 'V', N, A, N, ev, vecL, N, vecR, N, &
        & work1, 2*N, work2, info)

    if (info/=0) write (*,*) "An error occurred"

end program eigenvectors
```

**Notes:**

In this example, we are now calculating both the left and right eigenvectors of  $A$ , so we pass '**V**' as the first two arguments. As such, we now need to also pass two  $N \times N$  arrays to store the left and right eigenvectors, as well as two separate work arrays of size  $2N$ .

After ZGEEV is called, the **right eigenvectors** (the solutions  $\mathbf{v}_i$  to the eigenvalue equation  $A\mathbf{v} = \lambda\mathbf{v}$ ) are stored as the *columns* of matrix vecR; i.e.  $\mathbf{v}_i = \text{vecR}(:, i)$ .

Similarly, the **left eigenvectors** (the solutions  $\mathbf{w}_i$  to the eigenvalue equation  $\mathbf{w}^* A = \lambda^* \mathbf{w}^*$  or equivalently  $A^\dagger \mathbf{w} = \lambda \mathbf{w}$ ) are stored as the *columns* of matrix vecL; i.e.  $\mathbf{w}_i = \text{vecL}(:, i)$ .

### Hermitian Matrices: ZHEEV and ZHBEV

- ZHEEV first uses Krylov subspace techniques, in this case the **Lanczos iteration**, to reduce the matrix to its tridiagonal form. It then uses the implicit QR algorithm to determine the eigenvalues and eigenvectors.

#### Other Hermitian eigenvalue subroutines:

ZHEEV/D/ZHBEVD

Uses the divide-and-conquer algorithm; it is faster than EV routines, but uses a larger workspace.

ZHEEVX/ZHBEVX

Computes a selected subset of eigenvalues/eigenvectors. The smaller the subset, the faster the computation.

ZHEEVR

Uses the relatively robust representation algorithm. Fastest algorithm, and uses the smallest workspace.

In some cases, you might find yourself working with specific types of matrices, ones where there may be a more specific, faster algorithm to find the eigenvalues and eigenvectors. In quantum mechanics specifically, we often work with Hermitian matrices, and so can take advantage of the Lanczos algorithm. In such cases, the LAPACK subroutines ZHEEV (Table 7.5, for general complex Hermitian matrices) and ZHBEV (Table 7.6, for complex **banded** Hermitian matrices) are quite useful. See Examples 7.7 and 7.8 for an idea of how these subroutines are used.

### Storing Hermitian Matrices in LAPACK

Due to the symmetry of Hermitian matrices, their elements have the following property:

$$A_{ij} = A_{ji}^*. \quad (7.53)$$

Thus, we only need to store the upper or lower triangular part (including the diagonal), and we can reconstruct the entire Hermitian matrix! The remaining elements below/above the diagonal for upper/lower triangular representation can just be set to zero.

If we have an Hermitian matrix where the only non-zero elements are on the diagonal and super/sub-diagonals, we call it a **banded Hermitian matrix**, and can use ZBHEV to find the eigensystem. The advantages of using ZBHEV is that we *only* need to pass the non-zero elements – potentially saving huge amounts of memory if working with large matrices.

The banded matrix elements must be stored using the following conventions, depending on whether you want to provide the super or sub-diagonal elements.

*Upper triangular banded Hermitian matrix with k super-diagonals:*

$$\begin{bmatrix} a_1 & b_1 & c_1 & 0 & \cdots & 0 \\ a_2 & b_2 & c_2 & \ddots & & \vdots \\ a_3 & b_3 & \ddots & & 0 & \\ a_4 & \ddots & c_{N-2} & & & \\ & \ddots & b_{N-1} & & & \\ & & a_N & & & \end{bmatrix}_{N \times N} \Rightarrow \begin{bmatrix} 0 & 0 & c_1 & \cdots & c_{N-2} \\ 0 & b_1 & b_2 & \cdots & b_{N-1} \\ a_1 & a_2 & a_3 & \cdots & a_N \end{bmatrix}_{k \times N}$$

*Lower triangular banded Hermitian matrix with k sub-diagonals:*

$$\begin{bmatrix} a_1 & & & & & \\ b_1 & a_2 & & & & \\ c_1 & b_2 & a_3 & & & \\ 0 & c_2 & b_3 & a_4 & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \\ 0 & \cdots & 0 & c_{N-2} & b_{N-1} & a_N \end{bmatrix}_{N \times N} \Rightarrow \begin{bmatrix} a_1 & \cdots & a_{N-2} & a_{N-1} & a_N \\ b_1 & \cdots & b_{N-2} & b_{N-1} & 0 \\ c_1 & \cdots & c_{N-2} & 0 & 0 \end{bmatrix}_{k \times N}$$

Table 7.4

```
call ZHEEV(JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, RWORK, INFO)
```

- **JOBZ – character, intent(in)**  
if '**N**', the eigenvectors are not computed. if '**V**', they are computed.
- **UPLO – character, intent(in)**  
if '**U**', the matrix  $A$  is upper-triangular. If '**L**', it is lower-triangular.
- **N – integer, intent(in)**  
The number of rows of matrix  $A$
- **A – complex(8), dimension(N,N), intent(inout)**  
The complex matrix for which the eigenvalues (and eigenvectors) are to be computed.  
Once computed, the eigenvectors are returned as the *columns* of  $A$ :  $\mathbf{v}_j = \mathbf{A}(:, j)$ .
- **LDA – integer, intent(in)**  
The *leading dimension* of matrix  $A$ . In general, **LDA=N**.
- **W – real(8), dimension(N), intent(out)**  
A real one-dimensional array containing the computed eigenvalues.
- **WORK – complex(8), dimension(LWORK), intent(out)**  
A complex array used as the workspace. When complete, **WORK(1)** contains the optimal value for **LWORK**.
- **LWORK – integer, intent(in)**  
The dimension of the work array. In general, **LWORK**  $\geq \max(1, 2N - 1)$ .  
If **LWORK=-1**, then *just* the optimal value of **LWORK** is computed and returned in **WORK(1)**.
- **RWORK – real(8), dimension(3\*N-2), intent(out)**  
Another workspace array.
- **INFO – integer, intent(out)**  
Signifies the status of the algorithm. If **INFO=0**: success  
If **INFO=-i<0**: the  $i$ th argument had an illegal value.  
If **INFO=i>0**: Some eigenvalues and all eigenvectors failed to compute.

**Table 7.5** ZHEEV returns the eigenvalues and, optionally, the eigenvectors of a complex  $N \times N$  Hermitian matrix  $A$

### Example 7.7 ZHEEV

Using ZHEEV, calculate the eigenvalues *and* eigenvectors of the Hermitian matrix

$$A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

**Solution:**

```
program eigenvectors
    implicit none

    integer          :: N, i, j, info, LWORK
    complex(8), allocatable :: A(:, :), WORK(:)
    real(8),  allocatable :: ev(:), RWORK(:)

    N = 3
    allocate(A(N,N), ev(N), RWORK(3*N-2))

    A = 0.d0
    do i = 1, N
        A(i,i) = 2.d0
        if (i<N) A(i,i+1) = -1.d0
    end do

    ! create a dummy work array
    allocate(WORK(1))

    ! determine the optimum size of the work array
    call ZHEEV('N', 'U', N, A, N, ev, WORK, -1, RWORK, info)
    LWORK = min(int(WORK(1)), 1000)

    ! reallocate work array with optimum size
    deallocate(WORK);  allocate(WORK(LWORK))

    call ZHEEV('V', 'U', N, A, N, ev, WORK, LWORK, RWORK, info)
end program eigenvectors
```

- It is *optional* to first call ZHEEV to determine the optimum work array size.  
If you wish to skip this step, simply set

$LWORK \geq 2N - 1$

(however this may not be the most optimum value).

**Notes:**

In this example, the matrix  $A$  is constructed in *upper triangular* form (hence the second argument '**U**' to ZHEEV). The first call to ZHEEV passes **-1** as the WORK leading dimension argument – this instructs the subroutine to simply calculate the optimum work array size which we then store in variable LWORK.

Re-allocating the work array to be of size LWORK, the second ZHEEV call then calculates the eigenvalues ( $\lambda_i = ev(i)$ ) in ascending order, and the equivalent eigenvectors (which are stored in A:  $v_i = A(:, i)$ ).

```
call ZHBEV(JOBZ, UPLO, N, KD, AB, LDAB, W, Z, LDZ, WORK, LWORK, RWORK, INFO)
```

- **JOBZ – character, intent(in)**  
if '**N**', the eigenvectors are not computed.  
If '**V**', they are computed.
- **UPLO – character, intent(in)**  
if '**U**', the upper-triangular elements of  $A$  are stored.  
If '**L**', the lower-triangular elements are stored.
- **N – integer, intent(in)**  
The number of rows of matrix  $A$
- **KD – integer, intent(in)**  
If  $\text{UPLO}=\text{'U'}$ , the number of superdiagonals of  $A$ .  
If  $\text{UPLO}=\text{'L'}$ , the number of subdiagonals of  $A$ .
- **AB – complex(8), dimension(LDAB,N), intent(inout)**  
 $A$   $(KD + 1) \times N$  matrix storing the super/subdiagonals and diagonal of  $A$ .  
See Sect. 7.5.2 for details on how  $AB$  is constructed from  $A$ .
- **LDAB – integer, intent(in)**  
The *leading dimension* of array  $AB$ . In general,  $\text{LDAB}=KD+1$ .
- **W – real(8), dimension(N), intent(out)**  
A real one-dimensional array containing the computed eigenvalues.
- **Z – complex(8), dimension(LDZ,N), intent(out)**  
A two-dimensional array containing the eigenvectors of  $A$  in the columns:  
 $v_j = Z(:,j)$
- **LDZ – integer, intent(in)**  
The *leading dimension* of matrix  $Z$ . If  $\text{JOBZ}=\text{'V'}$ ,  $\text{LDZ}=N$ .
- **WORK – complex(8), dimension(N), intent(out)**  
A complex array used as the workspace.
- **RWORK – real(8), dimension(3\*N-2), intent(out)**  
Another workspace array
- **INFO – integer, intent(out)**  
Signifies the status of the algorithm. If  $\text{INFO}=0$ : success  
If  $\text{INFO}=-i < 0$ : the  $i$ th argument had an illegal value.  
If  $\text{INFO}=i > 0$ : Some eigenvalues and all eigenvectors failed to compute.

**Table 7.6** ZHBEV returns the eigenvalues and, optionally, the eigenvectors of a complex  $N \times N$  banded Hermitian matrix  $A$

**Example 7.8 ZHBEV**

Using ZHBEV, calculate the eigenvalues *and* eigenvectors of the banded Hermitian matrix

$$A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

**Solution:**

```
program eigenvectors
    implicit none

    integer :: N, i, j, info
    complex(8), allocatable :: A(:, :, :), vec(:, :, :), WORK(:)
    real(8), allocatable :: ev(:, :), RWORK(:)

    N = 3

    allocate(A(2:N), ev(N), vec(N, N), WORK(N), RWORK(3*N-2))

    A = 0.d0
    do j = 1, 3
        A(2,j) = 2.d0
        if (j>1) A(1,j) = -1.d0
    end do

    call ZHBEV('V', 'U', N, 1, A, 2, ev, vec, N, WORK, RWORK, info)

    if (info/=0) write (*,*) "An error occurred"

end program eigenvectors
```

**Notes:**

In this example, the matrix  $A$  is converted to upper-triangular banded storage form,

$$AB = \begin{bmatrix} 0 & -1 & -1 \\ 2 & 2 & 2 \end{bmatrix},$$

hence the second argument '**U**' to ZHBEV. Calling ZHEEV then calculates the eigenvalues ( $\lambda_i = \text{ev}(i)$ ) in ascending order, and the respective eigenvectors ( $\mathbf{v}_i = \text{vec}(:, i)$ ).

---

**LAPACK95**


---

A Fortan95 interface to LAPACK, known as **LAPACK95**, makes calling LAPACK subroutines a lot easier in modern Fortran. Whilst it is not as ubiquitous as LAPACK, feel free to try it out!

Compared to the LAPACK FORTRAN77 interface, LAPACK95 has

the following advantages:

- It uses **modules** (Sect. 2.8) rather than external subroutines; to use it in your program, simply insert

```
use la_precision, ONLY: WP=>DP
use f95_lapack
```

before **implicit none**. Then, when you compile, use the flag

```
-I/path/to/lapack95/mod/files
```

to let the compiler know where `f95_lapack.mod` and `la_precision.mod` are located.

- It also uses **generic subroutines** (Sect. 2.7.4), eliminating the need to choose a particular subroutine based on the required data type.
- The use of assumed-shape arrays vastly reduces the number of arguments required, and eliminates the need to explicitly pass work arrays. For example, compare the LAPACK95 wrapper of `LA_GEEV` to the original `ZGEEV` subroutine Table 7.3:

```
call LA_GEEV(A, W, VL, VR, INFO)
```

14 arguments has been reduced to just 5!

## 7.6 Python: Using SciPy

- In this chapter, we will only be covering a small subset of what SciPy is capable of. For more information and details, see the [SciPy linear algebra documentation](#).

If you’re using Python, the SciPy module provides a quick and easy interface to a huge number of different eigenvalue and eigenvector arrays. In this section, we’ll introduce the `scipy.linalg` submodule, and go through a few examples of the included eigensolvers.

### scipy.linalg versus numpy.linalg

A lot of the functions available in the SciPy linear algebra submodule are *also* available in the NumPy linear algebra submodule — so why bother using the SciPy version? While many of the functions are identical, the SciPy versions are typically more optimised than the NumPy versions, as they directly interface with the Fortran LAPACK library.

### 7.6.1 The Linear Algebra Submodule

To use the methods available in the linear algebra submodule, there are two main approaches. The first is to simply import SciPy, as you have done before, and then access the method directly from the linear algebra submodule:

```
>>> import scipy as sp
>>> sp.linalg.norm([1,1])
1.4142135623730951
```

Alternatively, you can import the SciPy linear algebra submodule under a separate name:

```
>>> import scipy.linalg as la
>>> la.norm([1,1])
1.4142135623730951
```

Some useful functions included in the linear algebra submodule are listed in Table 7.7.

### 7.6.2 Finding Eigenvalues and Eigenvectors Using SciPy

#### General Matrices

SciPy makes it very easy to find eigenvalues and eigenvectors using some of the most popular and optimised eigenvalue methods. In SciPy, the most general eigenvalue/eigenvector function is

```
>>> l, v = scipy.linalg.eig(A, b=None, left=False, right=True)
```

This calculates the eigenvalues (`l`) and right eigenvectors (`v`) of the square  $N \times N$  matrix `A`. All arguments after `A` are **optional**:

| Function                                    | Description                                                                                                                                                                                                         |
|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>multi_dot([A,B,...])</code>           | Performs the multiple dot product $A \cdot B \cdots$ . This can be faster than using <code>np.dot</code> in a chain, as Python determines the most efficient placement of brackets for the dot product evaluations. |
| <code>matrix_power(A, n)</code>             | returns the matrix $A^n$                                                                                                                                                                                            |
| <code>norm(x[, ord=None, axis=None])</code> | returns the matrix or vector norm along a given axis (or the whole array if <code>axis=None</code> ). If <code>ord=None</code> the standard Frobenius norm $ A  = \sqrt{\sum_{i,j}  a_{ij} ^2}$ is used.            |
| <code>cond(A[, ord=None])</code>            | returns the condition number of matrix $A$ , using the matrix norm definition described by <code>ord</code> .                                                                                                       |
| <code>det(A)</code>                         | returns the determinant of matrix $A$ .                                                                                                                                                                             |
| <code>solve(A,b)</code>                     | returns the numerical solution $x$ to the system of linear equations defined by $Ax = b$ using the LAPACK routine XGESV                                                                                             |
| <code>inv(A)</code>                         | returns the matrix inverse $A^{-1}$ if it exists.                                                                                                                                                                   |
| <code>kron(A)</code>                        | returns the Kronecker product $A \otimes B$ .                                                                                                                                                                       |

**Table 7.7** Some useful `scipy.linalg` functions

- `b`: An  $N \times N$  matrix. If provided, the eigenvalues and eigenvectors of the **generalised eigenvalue problem**  $Av = \lambda bv$  are calculated.
- `left`: If `True`, the normalised left eigenvectors  $w^\dagger A = \lambda w^\dagger$  are also returned; the output of the function is the tuple `l, w, v`.
- `right`: If `True` (default), the normalised right eigenvectors  $Av = \lambda v$  are also returned.

► The generalised eigenvalue problem sometimes shows up when working with dynamical classical systems.

In general, if you don't know in advance the form of the matrix you will be working with, `scipy.linalg.eig` is a good all-purpose solution. In cases where you are only interested in calculating the eigenvalues but *not* the eigenvectors, you can use the function `scipy.linalg.eigvals`; this is simply a shortcut to `scipy.linalg.eig`, with `left=False` and `right=False`.

► Under the hood, `scipy.linalg.eig` uses the XGEEV LAPACK subroutine, where X is determined based on the input matrix data type.

### Hermitian and Real Symmetric Matrices

In cases where you know you will only be dealing with Hermitian or real symmetric matrices, you can instead utilise the `scipy` function

```
>>> l, v = scipy.linalg.eigh(A, b=None, eigvals_only=False, eigvals=None)
```

- ▶ Under the hood, `scipy.linalg.eig` uses the HEEVR (Hermitian) or SYEVR (real symmetric) LAPACK subroutines. These first use the Lanczos algorithm to reduce the matrix to tridiagonal form, then find the eigenvalues using the relatively robust representation algorithm.

This calculates the eigenvalues (1) and right eigenvectors (v) of the square Hermitian or real symmetric  $N \times N$  matrix A. All arguments after A are **optional**:

- b: An  $N \times N$  matrix. If provided, the eigenvalues and eigenvectors of the **generalised eigenvalue problem**  $Av = \lambda bv$  are calculated.
- eigvals\_only: If `True`, only the eigenvalues are returned.
- eigvals: this keyword argument accepts a 2-tuple of the form `(n,m)`, where n and m are the indices of the eigenvalues and corresponding eigenvectors to be returned. Note that  $0 \leq n \leq m \leq N - 1$ .

As with the general case, if you are only interested in calculating the eigenvalues (and *not*) the eigenvectors, you can also use the function `scipy.linalg.eigvalsh`; this is simply a shortcut to `scipy.linalg.eigh`, with `eigvals_only=True`.

### Banded and Tridiagonal Matrices

Due to the symmetry of Hermitian matrices, their elements have the following property:

$$A_{ij} = A_{ji}^* \quad (7.54)$$

Thus, we only need to store the upper or lower triangular part (including the diagonal), and we can reconstruct the entire Hermitian matrix. The remaining elements below/above the diagonal for upper/lower triangular representation can just be set to zero.

If we have an Hermitian matrix where the only non-zero elements are on the super/sub-diagonals, we call it a **banded Hermitian matrix**, and can potentially save huge amounts of memory when working with large matrices.

If you have converted your Hermitian matrix to banded form, you can make use of the following SciPy function:

```
>>> l, v = scipy.linalg.eig_banded(A_band, lower=False,
                                 eigvals_only=False, select='a', select_range=None, max_ev=0)
```

This calculates the eigenvalues (l) and right eigenvectors (v) of the square Hermitian or real symmetric  $N \times N$  matrix, represented by the band array A\_band. All arguments after A\_band are **optional**:

- lower: Set to `True` if A\_band is in lower triangular banded form. Otherwise, A\_band is in upper triangular banded form. See below for the conventions needed for specifying A\_band.
- eigvals\_only: If `True`, only the eigenvalues are returned.

- `select`: select either all the eigenvalues be returned (`'a'`, default), eigenvalues in the range  $n < \lambda \leq m$  (`'v'`), or eigenvalues within the index range  $\lambda_n \leq \lambda \leq \lambda_m$  (`'i'`).
- `select_range`: if `select='v'` or `select=i`, the required eigenvalue range.
- `max_ev`: If `select='v'`, this returns only the `max_ev` number of eigenvalues.

As before, if you are only interested in calculating the eigenvalues (and *not*) the eigenvectors, you can also use the function `scipy.linalg.eigvals_banded`; this is simply a shortcut to `scipy.linalg.eig_banded`, with `eigvals_only=True`.

### Banded Hermitian Matrix Convention

Under the hood, `eigs_banded` uses the LAPACK subroutines `HBEVD/SBEVD` or `HBEVX/SBEVX`, depending on whether some or all of the eigenvalues are to be calculated. Therefore, the banded matrix `A_band` must be stored using the following conventions: *Upper triangular banded Hermitian matrix with k super-diagonals*:

$$\begin{bmatrix} a_1 & b_1 & c_1 & 0 & \cdots & 0 \\ a_2 & b_2 & c_2 & \ddots & & \vdots \\ a_3 & b_3 & \ddots & 0 & & \\ a_4 & \ddots & c_{N-2} & & & \\ \ddots & & b_{N-1} & & & \\ & & a_N & & & \end{bmatrix}_{N \times N} \Rightarrow \begin{bmatrix} 0 & 0 & c_1 & \cdots & c_{N-2} \\ 0 & b_1 & b_2 & \cdots & b_{N-1} \\ a_1 & a_2 & a_3 & \cdots & a_N \end{bmatrix}_{k \times N}$$

*Lower triangular banded Hermitian matrix with k sub-diagonals*:

$$\begin{bmatrix} a_1 & & & & & \\ b_1 & a_2 & & & & \\ c_1 & b_2 & a_3 & & & \\ 0 & c_2 & b_3 & a_4 & & \\ \vdots & \ddots & \ddots & \ddots & & \\ 0 & \cdots & 0 & c_{N-2} & b_{N-1} & a_N \end{bmatrix}_{N \times N} \Rightarrow \begin{bmatrix} a_1 & \cdots & a_{N-2} & a_{N-1} & a_N \\ b_1 & \cdots & b_{N-2} & b_{N-1} & 0 \\ c_1 & \cdots & c_{N-2} & 0 & 0 \end{bmatrix}_{k \times N}$$

In addition, if the matrix under investigation is a real, symmetric  $N \times N$  tridiagonal matrix  $T$ , for instance, after applying the Lanczos algorithm to an Hermitian matrix  $A \times \mathbb{C}^{N \times N}$ , then SciPy provides the function

```
>>> l, v = scipy.linalg.eigh_tridiagonal(D, E, eigvals_only=False
    select='a', select_range=None, max_ev=0)
```

Here,  $D$  is an length  $N$  array containing the diagonal elements  $T_{ii}$ , and  $E$  is a length  $N-1$  array containing the sub/super diagonal elements  $T_{i,i+1} = T_{i+1,i}$ . The remaining (optional) arguments are identical to those in `scipy.linalg.eigs_banded`. As with the others, you can also use the function `scipy.linalg.eigvalsh_tridiagonal`; this is simply a shortcut to `scipy.linalg.eigh_tridiagonal`, with `eigvals_only=True`.

### Upper Hessenberg form

When the matrix  $A$  is non-Hermitian, we can still apply Krylov subspace techniques to approximate the eigenspectrum, however, we can no longer use the Lanczos iteration. Instead, we use the Arnoldi iteration; this is a very similar process, but instead of calculating the orthonormal Arnoldi basis  $\mathcal{Q}_{r+1}$  by orthogonalising  $A\mathbf{q}_r$  against  $\{\mathbf{q}_{r-1}, \mathbf{q}_r\}$  (as we can due to the symmetry of Hermitian matrices in the Lanczos iteration), instead we must orthogonalise  $A\mathbf{q}_r$  against the entire previous basis  $\{\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_r\}$ . As a result,

$$Q^\dagger A Q = H$$

no longer produces a tridiagonal matrix, but instead a matrix  $H$  which is zero **only below the subdiagonal**. This is referred to as an **upper Hessenberg matrix**, and can be thought of as very nearly triangular.

SciPy provides a function to perform the Arnoldi iteration to calculate the upper Hessenberg form of a matrix  $A$ :

```
>>> H = scipy.linalg.hessenberg(A, calc_q=False)
```

If `calc_q=True`, then the matrix containing the Arnoldi basis along the columns,  $Q$ , is also returned:

```
>>> H, Q = scipy.linalg.hessenberg(A, calc_q=True)
>>> H == np.conj(Q).dot(A).dot(Q)
True
```

If  $A$  is Hermitian, then this function uses LAPACK to perform the Lanczos iteration, returning the resulting tridiagonal matrix  $T$  and the respective Lanczos basis  $Q$ .

### Sparse arrays

Sometimes, you might find yourself working with a large matrix that is very sparsely populated with non-zero values — we call a matrix **sparse** when the fraction of non-zero elements to zero elements is less than 50%. In such cases, huge computational and memory savings can be found by storing them using *sparse* matrix formats. That is, unlike dense arrays — like those provided by NumPy and Fortran, which require you to specify the value of every element in the array — sparse arrays only store the value and location of the non-zero values.

It turns out that certain numerical matrix methods, including eigenvalue solvers, are much more efficient when applied to large sparse matrices as opposed to large dense matrices — a chief example are the Krylov subspace techniques, including the Lanczos and Arnoldi algorithms. So, if you find yourself needing to work with large sparse matrices, it is often worth your time to investigate using a sparse matrix library, for example ARPACK in Fortran or `scipy.sparse` in Python. (Note: just like a lot of `scipy.linalg` functions rely on LAPACK subroutines, the submodule `scipy.sparse.linalg` uses ARPACK subroutines).

### Further reading

There are very many topics in linear algebra we haven't covered in this chapter. Even of those we have covered, there is so much more we could explore — enough to fill another book! Alas, we have a mission here, to solve the Schrödinger equation, so we will take a break from linear algebra and move onto the next topic.

For more details on Krylov subspace methods, the power iteration, and methods of eigenvalue approximations we haven't covered here, such as Jacobi iteration, QR decomposition, relatively robust representation, and many others, see

- Kressner, D. (2005). Numerical methods for general and structured eigenvalue problems. Berlin: Springer, ISBN 978-3-540-28502-1.

If you have an itch to explore more of the numerical linear algebra world, such as solving systems of linear equations, calculating the determinant, and a whole host others, see

- Trefethen, L. N., & Bau, D. (1997). Numerical linear algebra. Philadelphia: Society for Industrial and Applied Mathematics.

## Exercises

**P7.1** Consider the Hermitian matrix

$$A = \begin{bmatrix} 4 & -i & 2 \\ i & 2 & 2+7i \\ 2 & 2-7i & -2 \end{bmatrix}$$

- (a) Use the normalised inverse power iteration to find the smallest magnitude eigenvalue and associated eigenvector  $A$ . Hint: calculate the inverse of  $A$  by hand, or write a function to do it for you, based on the rule for inverting a  $3 \times 3$  matrix.
- (b) For each iteration, modify your code to store the absolute error  $|\lambda^{(n)} - \lambda|$ , where  $\lambda$  is the eigenvalue you calculated in part (a). Plot the absolute error versus the iteration number  $n$ . How does the convergence scale?

**P7.2** Using the same matrix above, find all three eigenvalues and eigenvectors using the normalised power iteration and the method of deflation.

**P7.3** (a) Starting with a  $2 \times 2$  diagonal matrix  $A = \text{diag}(\lambda_1, \lambda_2)$ , and a normalised starting vector  $\mathbf{x}_0 = (a, b)$  such that  $a^2 + b^2 = 1$ , show that the starting eigenvalue approximation for the Rayleigh quotient iteration is

$$\mu_0 = \mathbf{x}^T A \mathbf{x} = a^2 \lambda_1 + b^2 \lambda_2,$$

and calculate the shifted matrix  $A - \mu_0 I$ .

- (b) Determine an expression for  $\mathbf{x}_n = (A - \mu_0 I)^n \mathbf{x}_0$ .
- (c) Using the results of parts (a) and (b), show that the Rayleigh quotient iterative method converges such that

$$\lim_{n \rightarrow \infty} \frac{|\lambda_1^{(n+1)} - \lambda_1|}{|\lambda_1^{(n)} - \lambda_1|} = \left( \frac{\lambda_2}{\lambda_1} \right)^3$$

**P7.4** (a) Use the Lanczos algorithm provided in Sect. 7.3.1 to tridiagonalise the Hermitian matrix

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & -4 & 6 \\ 0 & 0 & 0 & -4 & 6 & -2 \\ 0 & 0 & -4 & 6 & -2 & 0 \\ 0 & -4 & 6 & -2 & 0 & 0 \\ -4 & 6 & -2 & 0 & 0 & 0 \\ 6 & -2 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- (b) Once you have found  $T$ , calculate the dominant eigenvector  $\mathbf{v}_1$  and eigenvalue  $\lambda_1$  via the power iteration.
- (c) Perform the power iteration on the non-tridiagonal matrix  $A$ , and calculate the dominant eigenvector  $\mathbf{u}_1$  and eigenvalue  $\mu_1$ . Compare the eigenvalue to part (b). How many iterations did it take to converge to the eigenvalue, compared to the tridiagonal power iteration?

You should find that, while the two dominant eigenvalue calculations provide almost identical values, the eigenvectors are significantly different.

This is because we are doing a *change of basis* transformation during the tridiagonalisation;  $Q^\dagger A Q = T$  where  $Q$  is the matrix with the Lanczos vectors along the columns. Since  $A$  and  $T$  are related by a similarity transform, they share eigenvalues, however you need to perform a change of basis to convert the eigenvectors of  $T$  to eigenvectors of  $A$ .

- (d) Modify your Lanczos iteration code to store each Lanczos vector  $\mathbf{q}_i$ . Construct  $Q$ , and verify that  $Q\mathbf{u}_1 = \mathbf{v}_1$ .

**P7.5** Generate a random  $10 \times 10$  Hermitian matrix.

- (a) Using NumPy or LAPACK, calculate the eigenvalues of this matrix.
- (b) Use the Lanczos algorithm to tridiagonalise the matrix.
- (c) At every Lanczos iteration  $2 \leq k \leq N$ , use NumPy or LAPACK to calculate the eigenvalues of the partial tridiagonal matrix  $T' \in \mathbb{R}^{k \times k}$ .
  - (i) After how many iterations do you have reasonably accurate approximations for the largest and smallest eigenvalue?
  - (ii) After how many iterations do you have reasonably accurate approximations for *all* the eigenvalues?

**P7.6** The non-symmetric matrix

$$A = \begin{bmatrix} 0 & 9 & 0 \\ 1 & 0 & 0 \\ 0 & 2 & 2 \end{bmatrix}$$

has real eigenvalues and 3 linearly independent eigenvectors. Matrices with these properties have what is known as an biorthogonal eigenbasis – a complete eigenbasis formed from the left and right eigenvectors.

Using Fortran and LAPACK,

- (a) Compute the eigenvalues  $\lambda_i$  of  $A$ . Verify that they are real.
- (b) Calculate the (right) eigenvectors  $\mathbf{v}_i$ , and show that they are linearly independent.
- (c) Calculate the left eigenvectors  $\mathbf{u}_i$ , and show that they are orthogonal to the right eigenvectors. That is,

$$\mathbf{u}_i \cdot \mathbf{v}_j = \delta_{ij}, \quad i, j = 1, 2, 3$$

where  $\delta_{ij}$  is the Kronecker-delta, and  $\mathbf{u}_i$  and  $\mathbf{v}_i$  have the same eigenvalue  $\lambda_i$ .

**P7.7** Consider the following Hermitian banded or tridiagonal matrix

$$L = \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & -1 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix}$$

- (a) Write a function in Fortran that constructs this  $N \times N$  matrix for input size  $N$  using band structure data storage (see Sect. 7.5.2 for details).
- (b) Using the LAPACK subroutine `ZHBEV` or the SciPy function `eig_banded`, calculate the eigenvalues and eigenvectors of  $L$ .

**P7.8 The quantum harmonic oscillator**

As we will see in later chapters, a fundamental problem in quantum mechanics is the calculation of the discrete spectrum of a bound Hamiltonian. By discretising the Hamiltonian using a suitable basis of quantum states, the Hamiltonian can be converted from an equation composed of operators to an equation based on matrices — this is the basis of matrix mechanics.

The quantum harmonic oscillator represents the dynamics of quantum particle in a quadratic potential, and in one-dimension has the following Hamiltonian:

$$\hat{H} = \frac{\hat{p}^2}{2m} + \frac{1}{2}m\omega^2\hat{x}^2$$

Here,  $\hat{x}$  is the position operator,  $\hat{p}$  is the momentum operator,  $m$  is the particle mass, and  $\omega$  the angular frequency of the oscillator.

In order to find the matrix representation of this Hamiltonian, a useful basis is the set of eigenstates of the Hamiltonian, representing

the various energy levels. This is an *infinite* basis, denoted  $|n\rangle$ ,  $n = 0, 1, \dots$ .

In this basis, the matrix representation of the position and momentum operators have the following form:

$$X = \langle m | \hat{x} | n \rangle = \sqrt{\frac{\hbar}{2\omega m}} \begin{bmatrix} 0 & \sqrt{1} & 0 & 0 & \cdots \\ \sqrt{1} & 0 & \sqrt{2} & 0 & \cdots \\ 0 & \sqrt{2} & 0 & \sqrt{3} & \cdots \\ 0 & 0 & \sqrt{3} & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$$P = \langle m | \hat{p} | n \rangle = i\sqrt{\frac{\hbar m \omega}{2}} \begin{bmatrix} 0 & -\sqrt{1} & 0 & 0 & \cdots \\ \sqrt{1} & 0 & -\sqrt{2} & 0 & \cdots \\ 0 & \sqrt{2} & 0 & -\sqrt{3} & \cdots \\ 0 & 0 & \sqrt{3} & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

- (a) Use the matrix forms of the operators  $\hat{x}$  and  $\hat{p}$  to construct the matrix representation of the quantum harmonic oscillator Hamiltonian  $\hat{H}$ . Note that matrix multiplication must be taken into account;  $\hat{x}^2 \equiv X^2 = X \cdot X$ .
- (b) Let  $\hbar = m = \omega = 1$ . Using LAPACK or SciPy, calculate the eigenvalues of this Hamiltonian in the case of a truncated  $N \times N$  matrix, for  $N = 6, 7, 8, \dots, 10$ . How do the eigenvalues converge as  $N$  increases?



## Chapter 8

# The Fourier Transform

Of all the numerical methods we have seen so far, the Fourier transform has arguably had the most significant impact over the course of the 20th century. It is an integral component across disciplines as diverse as signal processing, engineering, differential analysis, and of course, quantum mechanics, where the Fourier transform is used to relate the position and momentum space:

$$\psi(x, t) = \frac{1}{\sqrt{2\pi\hbar}} \int_{-\infty}^{\infty} \phi(p, t) e^{ipx/\hbar} dp, \quad (8.1)$$

$$\phi(p, t) = \frac{1}{\sqrt{2\pi\hbar}} \int_{-\infty}^{\infty} \psi(x, t) e^{-ipx/\hbar} dx. \quad (8.2)$$

It is no surprise then that the discovery of efficient numerical methods to compute the Fourier transform have had a profound effect on the aforementioned fields and numerous others that rely on their results. In this chapter, we will introduce the discrete Fourier transform, as well as the efficient Fast Fourier Transform (FFT) algorithm. We will then walk through some common implementations you can use in Fortran and Python.

### 8.1 Approximating the Fourier Transform

As we have seen earlier, the integrals found in the definition of the quantum wavefunction, Fourier transform are troublesome — we need to discretise the system to replace the continuous function  $\psi(x, t)$  and  $\phi(p, t)$  with discretised functions. However, in this case, it's even more troublesome due to the infinite domain of the integration! Let's attempt to both discretise these expressions, and find a way to deal with the infinities.

Before we do that, let's clear up some terms and remind ourselves what we mean by the position and momentum space.

### Position and Momentum Space

In quantum mechanics, the Fourier transform provides the relationship between the **position space** and the **momentum space** of the position and momentum operators  $\hat{x}$  and  $\hat{p}$ . In our case, this translates to:

- **Position space:**  $x$ , representing the particle's position,
- **Momentum space:**  $k = 2\pi/\lambda$ , representing the wavenumber (where  $\lambda$  is the de Broglie wavelength, in units  $\text{m}^{-1}$ ).

We use the name ‘momentum space’ since the wavenumber is directly related to the momentum in quantum mechanics; from the **de Broglie** equation, the momentum is  $p = \hbar k$  where  $\hbar = h/2\pi$  is the reduced Planck’s constant. As such,  $\psi(x, t)$  is typically described as the position-space representation, while  $\phi(k, t)$  is the momentum-space representation.

In other fields, such as electrical engineering and signal processing, the Fourier transform is typically used to transform signals between the time domain  $t$  and the frequency domain  $\omega$ , performing a frequency analysis of the system:

- **Time domain:**  $t$ , representing the signal time
- **Frequency domain:**  $\omega = 2\pi/T$ , representing the angular frequency (where  $T$  is the period of the signal, in units  $\text{s}^{-1}$ )

Due to the similarity between  $k$  and  $\omega$ , this interpretation of the Fourier transform is sometimes referred to as a spatial frequency analysis.

Therefore, using the de Broglie equation, we can write the quantum mechanics Fourier transform between position and momentum space as a transformation from position  $x$  to wavenumber  $k$  — this is the form in which it is most commonly seen.

$$\psi(x, t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \phi(k, t) e^{ikx} dk, \quad (8.3)$$

$$\phi(k, t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \psi(x, t) e^{-ikx} dx. \quad (8.4)$$

Note that, in addition to letting  $p = \hbar k$  and  $dp = \hbar dk$ , we have set  $\phi(k, t) = \sqrt{\hbar}\phi(p, t)$  so that the normalization factors in front of the two integrals are identical. This simple rescaling does not affect the result, and is merely a matter of convention; we’ll see more on this later.

To discretise these two equations, let’s begin by pacifying the infinities. Let’s assume that there exists two values  $x_{max}$  and  $k_{max}$  such that, for  $|k| > k_{max}$  and  $|x| > x_{max}$ ,  $|\phi(k, t)| \approx 0$  and  $|\psi(x, t)| \approx 0$ . It follows that a reasonable approximation is to simply integrate across  $-x_{max} \leq x \leq x_{max}$  and  $-k_{max} \leq k \leq k_{max}$  respectively — if the assumption holds true, integrand values outside these domains are essentially zero and do not contribute

to the integral. Therefore,

$$\psi(x, t) \approx \frac{1}{\sqrt{2\pi}} \int_{-k_{max}}^{k_{max}} \phi(k, t) e^{ikx} dk, \quad (8.5)$$

$$\phi(k, t) \approx \frac{1}{\sqrt{2\pi}} \int_{-x_{max}}^{x_{max}} \psi(x, t) e^{-ikx} dx. \quad (8.6)$$

We can now discretise our continuous variables and functions, using  $N$  discrete values:

$$\begin{aligned} x_n &= n\Delta x + x_0, & n = 0, 1, \dots, N-1 &\Rightarrow \psi(x_n, t) = \psi_n \\ k_m &= m\Delta k + k_0, & m = 0, 1, \dots, N-1 &\Rightarrow \phi(k_m, t) = \phi_m \end{aligned} \quad (8.7)$$

Assuming  $\Delta x \ll 1$  and  $\Delta k \ll 1$ , we can now approximate the definite integral using Riemann sums:

$$\int_a^b f(x) dx \approx \Delta x \sum_{n=0}^{N-1} f(x_n). \quad (8.8)$$

Applying this to both integrals, we get

$$\boxed{\psi_n = \frac{\Delta x}{\sqrt{2\pi}} \sum_{m=0}^{N-1} \phi_m e^{ik_m x_n} \Leftrightarrow \phi_m = \frac{\Delta k}{\sqrt{2\pi}} \sum_{n=0}^{N-1} \psi_n e^{-ik_m x_n}.} \quad (8.9)$$

### 8.1.1 Momentum Space Sampling

To ensure that our approximations are consistent, let's substitute the Fourier transform of  $\phi_m$  into the inverse Fourier transform definition of  $\psi_n$ :

$$\begin{aligned} \psi_\ell &= \frac{\Delta x}{\sqrt{2\pi}} \sum_{m=0}^{N-1} \phi_m e^{ik_m x_\ell} = \frac{\Delta x}{\sqrt{2\pi}} \sum_{n=0}^{N-1} \left( \frac{\Delta k}{\sqrt{2\pi}} \sum_{m=0}^{N-1} \psi_n e^{-ik_m x_n} \right) e^{ik_m x_\ell} \\ &= \frac{\Delta x \Delta k}{2\pi} \sum_{n=0}^{N-1} \psi_n \left( \sum_{m=0}^{N-1} e^{ik_m(x_\ell - x_n)} \right). \end{aligned} \quad (8.10)$$

In order for the left-hand side to equal the right-hand side, we require the double summation to collapse into a single term. To do so, we must make use of the **orthogonality** of the complex exponentials,

$$\delta_{\ell n} = \frac{1}{N} \sum_{m=0}^{N-1} e^{2\pi i m(\ell-n)/N} = \begin{cases} 0, & \ell \neq n, \\ 1, & \ell = n \end{cases}, \quad (8.11)$$

where  $\delta_{\ell n}$  is the Kronecker Delta; from its definition, it is easy to see that it extracts only a single term from a summation:  $\sum_{n=0}^{N-1} \psi_n \delta_{\ell n} = \psi_\ell$ . But how do we apply this to Eq. 8.10?

There is an additional constraint on our discrete grids that we haven't yet taken into account; since they are symmetric around  $x = 0$  and  $k = 0$ , we must have  $x_{max} = -x_0 = x_{N-1}$  and  $k_{max} = -k_0 = k_{N-1}$ . Using these relationships, we can determine an exact expression for  $x_0$  and  $k_0$  in terms of  $N$ :

- The term  $a = \frac{1}{2}(N - 1)$  occurs regularly when considering discretised Fourier transforms!

$$\begin{aligned} x_{N-1} &= (N - 1)\Delta x + x_0 = -x_0 \Rightarrow x_0 = -\frac{1}{2}(N - 1)\Delta x = -a\Delta x, \\ k_{N-1} &= (N - 1)\Delta k + k_0 = -k_0 \Rightarrow k_0 = -\frac{1}{2}(N - 1)\Delta k = -a\Delta k. \end{aligned} \quad (8.12)$$

Substituting these into Eq. 8.10, we get

$$\psi_\ell = \frac{\Delta x \Delta k}{2\pi} \sum_{n=0}^{N-1} \psi_n \left( \sum_{m=0}^{N-1} e^{i\Delta x \Delta k(m-a)(\ell-n)} \right). \quad (8.13)$$

We require the bracketed sum of exponentials to be orthogonal — comparing this to Eq. 8.11, this is the case if and only if

$$\boxed{\Delta x \Delta k = \frac{2\pi}{N}} \quad (8.14)$$

Substituting this back in Eq. 8.13, we find that the right hand side simply reduced to  $\psi_\ell$  (try this yourself). Success! Our discretised Fourier transform is now fully consistent.

### What does this mean?

It's important to stop and pause here, and really think about this additional constraint. By requiring the  $\Delta x \Delta k = \frac{2\pi}{N}$ , we are discarding the ability to independently choose our grid spacings  $\Delta x$  and  $\Delta k$  — they are now explicitly intertwined.

For example, if we discretise our wavefunction into  $N = 100$  points, each separated by  $\Delta x = 0.01$  in position space, then the resulting momentum wavefunction  $\phi_m$  will also comprise  $N = 100$  points, separated in momentum space by

$$\Delta k = \frac{2\pi}{N \Delta x} \approx 6.28 \text{ m}^{-1}.$$

In effect, **the more closely we sample our wavefunction in position space, the more distant our sampling is in momentum space**. Choosing  $\Delta x$  is therefore a careful balancing act — we want it to be small enough to accurately represent our wavefunction, but large enough so that we also accurately sample the resulting discretised Fourier transform. How do we do this?

### 8.1.2 The Nyquist Theorem

Recall that the value  $k_{max} = \frac{1}{2}(N - 1)\Delta k$  is uniquely determined by the discretised momentum grid. Using the results from the previous section, we can instead write this in terms of the position spacing  $\Delta x$ :

$$k_{max} = \frac{1}{2}(N - 1)\Delta k = \frac{(N - 1)\pi}{N\Delta x} \approx \frac{\pi}{\Delta x}, \quad N \gg 1. \quad (8.15)$$

Thus, not only does  $\Delta x$  determine the momentum-space grid spacing,  $\Delta k$ , it also determines the maximum possible value of  $k$  in the resulting transform! This is known as the **Nyquist momentum**.

We can use this insight to approach this problem in reverse. What if we know, in advance, a particular value of  $k_{max}$  such that, for all  $|k_m| > |k_{max}|$ ,  $|\phi(k_m)| = 0$ ? In such a case, we can see from the Nyquist momentum that we simply need to choose

$$\Delta x \leq \frac{\pi}{k_{max}} \quad (8.16)$$

to ensure that we recover *all* the momentum-space information of the system when performing the discretised Fourier transform. This is referred to as the **Nyquist theorem** or the **Nyquist condition**, and is the crux of the Nyquist–Shannon sampling theorem.

**Theorem 8.1.** *If a function  $f(x)$  contains no wavenumber components higher than  $k_{max}$ , then its Fourier transform can be completely determined by sampling it in discrete steps of  $\Delta x \leq \pi/k_{max}$ .*

► In signal processing and electrical engineering, the Fourier transform is commonly viewed as a transform between the time and frequency domain, and so this is instead known as the Nyquist frequency.

### Dimensionless Form

Putting everything together, we can write the discretised Fourier transform such that the exponential has a purely dimensionless argument in terms of integers  $n$  and  $m$ .

Recalling that  $a = (N - 1)/2$  and  $\Delta k = 2\pi/N\Delta x$ ,

$$x_n = n\Delta x + x_0 = \Delta x(n - a), \quad (8.17)$$

$$k_m = m\Delta k + k_0 = 2\pi(m - a)/N\Delta x, \quad (8.18)$$

and the dimensionless form is given by

$$\begin{aligned} \psi_n &= \frac{\Delta x}{\sqrt{2\pi}} \sum_{m=0}^{N-1} \phi_m e^{2\pi i(m-a)(n-a)/N} \\ \phi_m &= \frac{\Delta k}{\sqrt{2\pi}} \sum_{n=0}^{N-1} \psi_n e^{-2\pi i(m-a)(n-a)/N}. \end{aligned} \quad (8.19)$$

**Example 8.1** Discretised Fourier transform (Fortran)

Use Fourier differentiation to find the momentum-space representation  $\phi(k)$  of the wavefunction  $\psi(x) = e^{-x^2/2} \sin(4x)$  over the domain  $-5 \leq x \leq 5$  with  $N = 201$  discretised points.

**Solution:**

```

program fourier
    implicit none
    complex(8), parameter :: j=(0.d0,1.d0)
    real(8), parameter :: pi=4.d0*atan(1.d0)
    integer :: n, m, NN, a
    real(8) :: x0, dx, k0, dk, psi(201)
    complex(8) :: sum, phi(201)

    NN = 201
    a = (NN-1)/2

        ! x grid
    x0 = -5.
    dx = abs(2*x0)/(NN-1)

        ! k grid
    k0 = -pi/dx
    dk = 2.*pi/(NN*dx)

    ! create the discrete wavefunction
    psi = [(exp(-(x0+n*dx)**2/2.)*sin(4.*(x0+n*dx)), n=0, NN-1)]
    phi = 0.d0

    ! perform the discretised Fourier transform algorithm
    do m=0, NN-1 ! loop over m
        sum = 0 ! sum over n
        do n=0, NN-1
            sum = sum + psi(n+1)*exp(-2.*pi*j*(m-a)*(n-a)/NN)
        end do
        ! assign the mth element of phi
        phi(m+1) = sum*dk/sqrt(2.*pi)
    end do
end program fourier

```

**Example 8.2** Discretised Fourier transform (Python)

Use Fourier differentiation to find the momentum-space representation  $\phi(k)$  of the wavefunction  $\psi(x) = e^{-x^2/2} \sin(4x)$  over the domain  $-5 \leq x \leq 5$  with  $N = 201$  discretised points.

**Solution:**

```
# define the properties
N = 201
a = (N-1)/2
dx = (5-(-5))/(N-1)

# define the x-grid and discretised wavefunction
x = np.arange(-5, 5+dx, dx)
psi = np.exp(-(x**2)/2)*np.sin(4*x)

# define the k-grid
dk = 2*np.pi/(N*dx)
kmax = np.pi/dx
k = np.arange(-kmax, kmax, dk)

# the 2-dimensional Fourier matrix using broadcasting
n = np.arange(N).reshape((-1, 1))
m = np.arange(N).reshape((1, -1))
W = np.exp(-2j*np.pi*(m-a)*(n-a)/N)

# perform the discretised Fourier algorithm
# psi is indexed by n
psi = psi.reshape(-1, 1)
# sum over n axis
phi = np.sum(psi * W, axis=0)*dk/np.sqrt(2*np.pi)
```

## 8.2 Fourier Differentiation

One of the advantages of transformations such as the Fourier transform or the Laplace transform is that they allow us to transform differential operators in position space to simple arithmetic operations in momentum-space. For example, consider the following wavefunction in position space, decomposed as an inverse Fourier transform over wavenumbers in momentum-space:

$$\psi(x, t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \phi(k, t) e^{ikx} dk. \quad (8.20)$$

Taking the derivative with respect to  $x$  of both sides,

$$\begin{aligned} \frac{\partial}{\partial x} \psi(x, t) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \phi(k, t) \frac{\partial}{\partial x} e^{ikx} dk \\ &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \phi(k, t) ike^{ikx} dk. \end{aligned} \quad (8.21)$$

That is, differentiation with respect to  $x$  in position space is simply multiplication by  $ik$  in momentum space! We can even go one step further, and replace  $\phi(k, t)$  with the Fourier transform of  $\psi(x', t)$  in position space:

$$\frac{\partial}{\partial x} \psi(x, t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \psi(x', t) ike^{ik(x-x')} dk dx'. \quad (8.22)$$

Does this result still follow for the discretised Fourier transform? Indeed, it does:

$$\frac{\partial \psi_\ell}{\partial x} = \frac{\Delta k}{\sqrt{2\pi}} \sum_{m=0}^{N-1} \phi_m \frac{\partial}{\partial x_\ell} e^{ik_m x_\ell} = \frac{\Delta k}{\sqrt{2\pi}} \sum_{m=0}^{N-1} \phi(k_m) ik_m e^{ik_m x_\ell}. \quad (8.23)$$

By replacing  $\phi_m$  by the inverse DFT of  $\psi_n$ , we can put this in a similar form to our expression using the Fourier transform:

$$\psi'_\ell = \frac{1}{2\pi} \Delta x \Delta k \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} \psi_n ik_m e^{ik_m (x_\ell - x_n)}. \quad (8.24)$$

### Dimensionless Form

Substituting in  $x_n = \Delta x(n - a)$ ,  $k_m = \Delta k(m - a)$ ,  $\Delta k = 2\pi/N\Delta x$ , and recalling that  $a = (N - 1)/2$ , we can write this in terms of the dimensionless integers  $n$  and  $m$ :

$$\psi'_\ell = \frac{2\pi i}{N^2 \Delta x} \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} (m - a) \psi_n e^{2\pi i (m-a)(\ell-n)/N}. \quad (8.25)$$

**Example 8.3** Fourier differentiation (Fortran)

Use Fourier differentiation to differentiate the wavefunction  $\psi(x) = e^{-x^2/2} \sin(4x)$  over the domain  $-5 \leq x \leq 5$  with  $N = 201$  discretised points.

**Solution:**

```
program fourierdiff
    implicit none
    complex(8), parameter :: j=(0.d0,1.d0)
    real(8), parameter :: pi=4.d0*atan(1.d0)
    integer :: i, m, l, N, a
    real(8) :: x0, dx, psi(201), dpsi(201)
    complex(8) :: sum

    N = 201
    a = (N-1)/2

    x0 = -5.
    dx = abs(2*x0)/(N-1)

    ! create the discrete wavefunction
    psi = [(exp(-(x0+i*dx)**2)/2.*sin(4.*(x0+i*dx)), i=0, N-1)]
    dpsi = 0.d0

    ! loop over all xl values in the domain
    do l=0, N-1
        sum = 0
        ! perform the Fourier diff algorithm for each xl
        do i=0, N-1
            do m=0, N-1
                sum = sum + psi(i+1)*(m-a) &
                    * exp(2.*pi*j*(m-a)*(l-i)/N)
            end do
        end do
        dpsi(l+1) = sum*2*pi*j/(dx*N**2)
    end do
end program fourierdiff
```

**Example 8.4** Fourier differentiation (Python)

Use Fourier differentiation to differentiate the wavefunction  $\psi(x) = e^{-x^2/2} \sin(4x)$  over the domain  $-5 \leq x \leq 5$  with  $N = 201$  discretised points.

**Solution:**

```
# define the properties
N = 201
a = (N-1)/2
dx = (5-(-5))/(N-1)

# define the x-grid and discretised wavefunction
x = np.arange(-5, 5+dx, dx)
psi = np.exp(-(x**2)/2)*np.sin(4*x)

# create the 3-dimensional Fourier diff matrix
l = np.arange(N).reshape((-1,1,1))
n = np.arange(N).reshape((1,-1,1))
m = np.arange(N).reshape((1,1,-1))

W = np.exp(2j*np.pi*(m-a)*(l-n)/N)

# perform the Fourier differentiation
psi = psi.reshape(1,-1,1)
dpsi = np.sum((2j*np.pi)/(dx*N**2) * psi * (m-a) * W, axis=(1,2))
```

## 8.3 The Discrete Fourier Transform

While the technique we explored above works reasonably well for approximating the Fourier transform between position and momentum space, in practice there are much faster and more efficient algorithms we can harness to calculate the same discretisation. Chief among those is the Fast Fourier Transform, or the FFT as it is frequently referred to. Before we can start adapting our approximation for the FFT, however, we need to understand the transform it is based on — the discrete Fourier transform.

Before we begin, we must stress an important point. The discrete Fourier transform (which we will frequently refer to as the DFT) is **not** the same as the approximate discretised Fourier transform we calculated in the previous section! While the discretised Fourier transform is a discrete approximation to the Fourier transform — in the same way that the finite difference method is a discrete approximation to the derivative — the DFT is instead a transform in its own right, that happens to act on and return discrete data. To summarise:

- **Approximate Fourier transform:** discrete *approximation* to the continuous Fourier transform, considered in previous sections;
- **Discrete Fourier transform or DFT:** an important and well-defined discrete transform that can transform *exactly* between two sets of discrete and periodic data.

Thus, the DFT is an important tool in numerical Fourier analysis in its own right. We'll get into more of the details of the DFT shortly, and see why it is so useful in calculating the discretised Fourier transform. First of all, let's have a quick reminder on the mathematical details of the Fourier transform.

### 8.3.1 Fourier Series and Transforms

In the chapters on differentiation and integration, we spent quite some time making use of the Taylor series, which allows us to expand a function as an infinite summation of polynomials. The **Fourier series**, in contrast, is the expansion of a **periodic function**  $f(x) \in \mathbb{R}$  in terms of an infinite sum of sines and cosines:

$$f(x) = c + \sum_{k=1}^{\infty} (a_k \cos kx + b_k \sin kx). \quad (8.26)$$

Here,  $c$  is a constant contribution to  $f(x)$ , while  $\cos(kx)$  and  $\sin(kx)$  represent the even and odd periodic components respectively. Each of these sine or cosine terms are characterised by their **wavelength**  $\lambda$ :

$$\lambda = \frac{2\pi}{k}, \quad (8.27)$$

where  $k$  is referred to as the **angular wavenumber**, the number of wavelengths per  $2\pi$  radians. Note that, since  $\cos kx$  and  $\sin kx$  are both periodic over the region  $[-\pi, \pi]$ , as a result, so is  $f(x)$ .

Using Euler's formula  $e^{ix} = \cos x + i \sin x$ , the Fourier series expansion is commonly written as the sum over complex coefficients,

$$f(x) = \sum_{k=-\infty}^{\infty} F_k e^{ikx} \quad \text{where} \quad F_k = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) e^{-ikx} dx. \quad (8.28)$$

The **Fourier coefficients**,  $F_k$ , tell us the contribution of the periodic term with wavelength  $\lambda = 2\pi/k$  to the function  $f(x)$ . The Fourier coefficients can therefore be thought of as providing a **wavelength analysis** of the input periodic function  $f(x)$ , by indicating the amount that periodic functions of differing wavelength contribute.

What if the function  $f(x)$  is *not* periodic over the interval  $[-\pi, \pi]$ , but is instead periodic over the interval  $[-L/2, L/2]$  for some positive  $L$ ? In this case, we require the series expansion functions to also be periodic within this interval. For this to be the case, the allowed wavelengths need to be multiples of the interval length; thus, only wavelengths

$$\lambda = \frac{2\pi}{k} = \frac{L}{m}, \quad m = 0, 1, 2, \dots$$

are allowed. Letting  $k_m = 2\pi m/L$ , the Fourier series is now written

$$f(x) = \sum_{m=-\infty}^{\infty} F_m e^{2\pi imx/L} \quad \text{where} \quad F_m = \frac{1}{L} \int_{-L/2}^{L/2} f(x) e^{-2\pi imx/L} dx. \quad (8.29)$$

Since consecutive values of  $k_m$  are equally spaced by  $\Delta k = 2\pi/L$ , we say that the spacing between Fourier coefficients  $F_m$  in the **wavenumber domain** is  $\Delta k$ .

What happens if we let  $L \rightarrow \infty$ ? In other words, this is the limit in which  $f(x)$  is no longer periodic — have a look at Fig. 8.1. As a start, we can see that the Fourier coefficients are no longer discrete; since  $\Delta k \rightarrow 0$  in this limit, the Fourier coefficients become a continuous function.

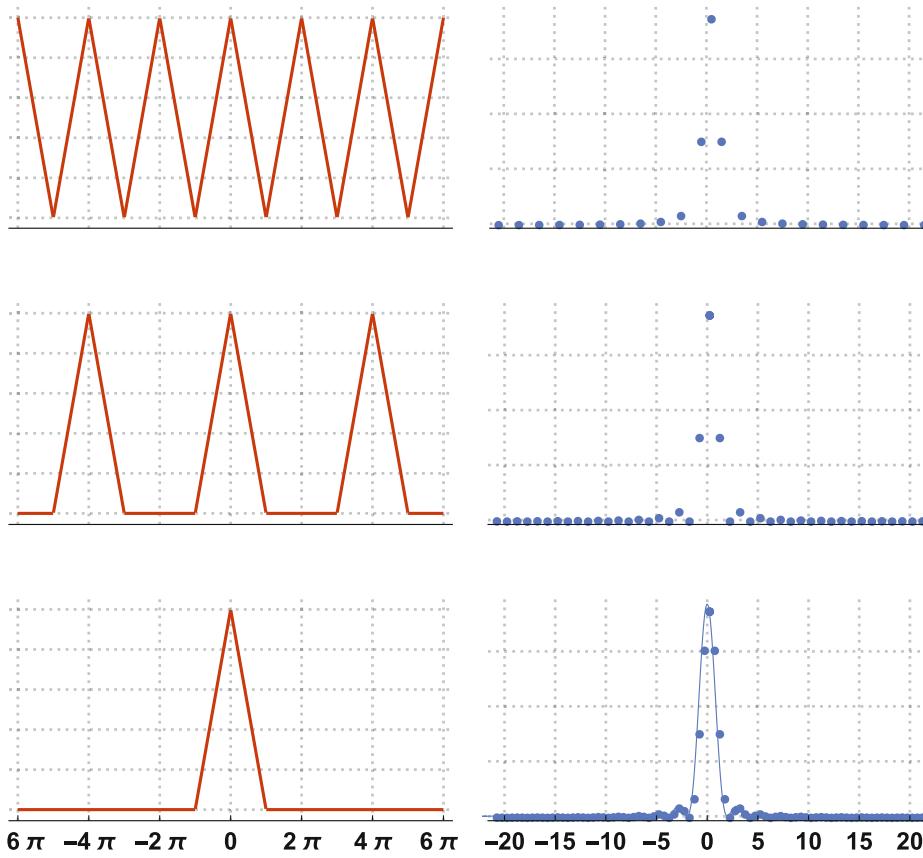
Letting  $k = 2\pi m/L$  be our new continuous variable, in this limit the Fourier series becomes an integral:

$$f(x) = \int_{-\infty}^{\infty} F(k) e^{ikx} dk \iff F(k) = \frac{1}{2\pi} \int_{-\infty}^{\infty} f(x) e^{-ikx} dx. \quad (8.30a)$$

This is the **Fourier transform**, and allows us to perform a frequency or wavenumber analysis on non-periodic functions.

Compare this to the quantum physics Fourier transform between the position and momentum-space in the previous section (Eq. 8.3); you might notice a major difference in their **normalisation** factors.

- We refer to  $f(x)$  and  $F(k)$ , two functions related via a Fourier transform, as **Fourier pairs**.



**Figure 8.1** **Left:** A unit triangle function  $f(x)$ , with period, from top to bottom, of  $L = 2\pi, 4\pi, 8\pi$ . **Right:** The resulting Fourier coefficients for  $-21 \leq k_m \leq 21$ . As  $L$  grows, the spacing between the Fourier coefficients reduces until, when  $\Delta k \ll 1$ , we approximate the continuous Fourier transform  $F(k)$  (bottom right)

### Important: Fourier normalisation

All Fourier transforms have a normalisation factor, but all that matters is that it is included *somewhere* — it is up to convention where it is placed. Some disciplines, such as signal processing, keep the normalisation factor ( $1/L$  for the Fourier series and  $1/2\pi$  for the Fourier transform) as part of the *inverse transformation*, as we did above.

In quantum physics, however, we saw that it is more common to share the normalisation factor across both the Fourier transform *and* the inverse transform, leading to a **symmetric** normalisation factor of  $1/\sqrt{2\pi}$  on both. We will see the reason for this later! However, this is something that must be kept in mind when performing Fourier analyses.

- The **Kronecker delta** is defined by

$$\delta_{mm'} = \begin{cases} 1, & m = m' \\ 0, & m \neq m' \end{cases}$$

and can reduce summations:

$$\sum_n f(n) \delta_{nm} = f(m).$$

Similarly, the **Dirac delta function** can be used to reduce integrals:

$$\int_{-\infty}^{\infty} f(x) \delta(x-x') dx = f(x').$$

### Orthogonality of the complex exponential

Why can we always expand periodic functions by a Fourier series? This follows from the fact that the complex exponential  $w_m(x) = e^{-2\pi imx/L}$  forms a **complete orthogonal set** of functions when integrating  $x$  over the period, and when summing over all  $m$ :

$$\int_{-L/2}^{L/2} w_m^*(x) w_{m'}(x) dx = L \delta_{mm'},$$

$$\sum_{m=-\infty}^{\infty} w_m^*(x) w_m(x') = L \delta(x - x'),$$

where  $\delta(x - x')$  is the Dirac delta function, and  $\delta_{nn'}$  is the Kronecker delta function.

In fact, you can use this orthogonality to derive the expression for the Fourier coefficients  $F_k$  from the Fourier series of  $f(x)$ . Try applying the integral  $\int_{-L/2}^{L/2} dx e^{-2\pi imx/L}$  to both sides of

$$f(x) = \sum_{m=-\infty}^{\infty} F_m e^{2\pi imx/L},$$

and utilise the orthogonality condition. What do you find?

### 8.3.2 The DFT

To recap the last two pages, we have seen two separate use-cases of Fourier analysis:

- **Fourier series:** used when we have a continuous, periodic function  $f(x)$ , resulting in discrete Fourier series coefficients  $F_m$ .
- **Fourier transform:** used when we have a continuous but *non-periodic* function  $f(x)$ , resulting in a continuous Fourier transform  $F(k)$ .

The discrete Fourier transform, on the other hand, is used when we have a discrete, periodic function  $f_n$ , resulting in a discrete, periodic transform  $F_m$ . As a result, it perfectly complements the Fourier series and Fourier transform. The DFT and the inverse DFT are given by

$$F_m = \sum_{n=0}^{N-1} f_n e^{-2\pi i m n / N} \Leftrightarrow f_n = \frac{1}{N} \sum_{m=0}^{N-1} F_m e^{2\pi i m n / N}. \quad (8.31)$$

A couple of things to note about the DFT:

- The DFT contains two finite summations; it accepts an input function with  $N$  discrete points, and its output also contains  $N$  discrete points.
- Like the Fourier series and Fourier transform, we can associate both the DFT and inverse DFT with ‘position’ and ‘wavenumber’ variables respectively; in this case, they are both discrete variables:

$$\begin{aligned} x_n &= n, \quad n = 0, 1, \dots, N - 1 \\ k_m &= m\Delta k, \quad \Delta k = \frac{2\pi}{N}, \quad m = 0, 1, \dots, N - 1. \end{aligned} \quad (8.32)$$

- Because both summations are truncated to  $0 \leq n \leq N - 1$ , this places a periodicity of  $N$  on both the function  $f_n$  and its Fourier transform  $F_m$ :

$$f_{n+aN} = f_n \quad \text{and} \quad F_{m+aN} = F_m$$

for all integers  $a$ . This can be shown by considering the DFT definition:

$$F_{m+aN} = \sum_{n=0}^{N-1} f_n e^{-2\pi i(m+aN)n/N} = \sum_{n=0}^{N-1} f_n e^{-2\pi imn/N} e^{-2\pi ian} = F_m,$$

where  $e^{-2\pi ian} = 1$  for all integers  $a$  and  $n$ .

**Example 8.5** Discrete Fourier transform (Fortran)

Write a program that performs the discrete Fourier transform of the unit triangle function, defined by

$$u(x) = \begin{cases} x + 1, & -1 \leq x < 0 \\ 1 - x, & 0 \leq x < 1 \\ 0, & \text{otherwise} \end{cases}$$

and compare the results to the continuous Fourier transform of the unit triangle function in Fig. 8.1.

**Solution:** First of all, let's discretise the unit triangle function, using

$$f_n = u(x_n), \quad f_0 = -5, \quad f_{N-1} = 5, \quad N = 201.$$

As a result, the discrete Fourier transform  $F_n$  is

$$F_m = \sum_{n=0}^{N-1} f_n e^{-2\pi i m n / 200}.$$

- We use the trig identity  $\tan^{-1}(1) = \pi/4$  to easily create a pi parameter, as well as defining a parameter for the complex number j.

```
program dft
    implicit none
    complex(8), parameter :: j=(0.d0,1.d0)
    real(8), parameter :: pi=4.d0*atan(1.d0)
    integer :: n, m, NN
    real(8) :: dx, x, f(200)
    complex(8) :: Fdft(200)

    NN = 200
    dx = (5.d0-(-5.d0))/(NN-1)

    ! define the discretised unit triangle function
    f = 0.d0; x = -5.d0
    do n=1,NN
        if (-1<x .and. x<0) then
            f(n) = x + 1
        else if (0<=x .and. x<1) then
            f(n) = 1 - x
        end if
        x = x + dx
    end do

    ! perform the DFT
    Fdft = 0.d0
    do m=0,NN-1      ! calculate the mth component
        do n=0, NN-1 ! sum over all n
            Fdft(m+1) = Fdft(m+1) + f(n+1)*exp(-2*j*pi*m*n/NN)
        end do
    end do
end program dft
```

**Example 8.6** Discrete Fourier transform (Python)

Write a program that performs the discrete Fourier transform of the unit triangle function, defined by

$$u(x) = \begin{cases} x + 1, & -1 \leq x < 0 \\ 1 - x, & 0 \leq x < 1 \\ 0, & \text{otherwise} \end{cases}$$

and compare the results to the continuous Fourier transform of the unit triangle function in Fig. 8.1.

**Solution:** First of all, let's discretise the unit triangle function, using

$$f_n = u(x_n), \quad f_0 = -5, \quad f_{N-1} = 5, \quad N = 201.$$

As a result, the discrete Fourier transform  $F_n$  is

$$F_m = \sum_{n=0}^{N-1} f_n e^{-2\pi i m n / 200}.$$

Alternatively, we may write this using matrix-vector multiplication,  $\mathbf{F} = W\mathbf{f}$ , where matrix  $W$  has elements  $W_{mn} = e^{-2\pi i m n / 200}$ . Using NumPy, and writing a Python program to calculate this:

```
#!/usr/bin/env python3
import numpy as np

# define the x grid
N = 200
dx = (5 - (-5)) / (N - 1)
xgrid = np.arange(-5, 5 + dx, dx)

# define the discretised unit triangle function
f = np.zeros(xgrid.shape)
for i, x in enumerate(xgrid):
    if -1 < x < 0:
        f[i] = x + 1
    elif 0 <= x < 1:
        f[i] = 1 - x

# create the matrix W using broadcasting
m = np.arange(N)[:, None]      # this is the first index
n = np.arange(N)[None, :]       # this is the second index
W = np.exp(-2j * np.pi * m * n / N)

# Perform the DFT
F = np.dot(W, f)
```

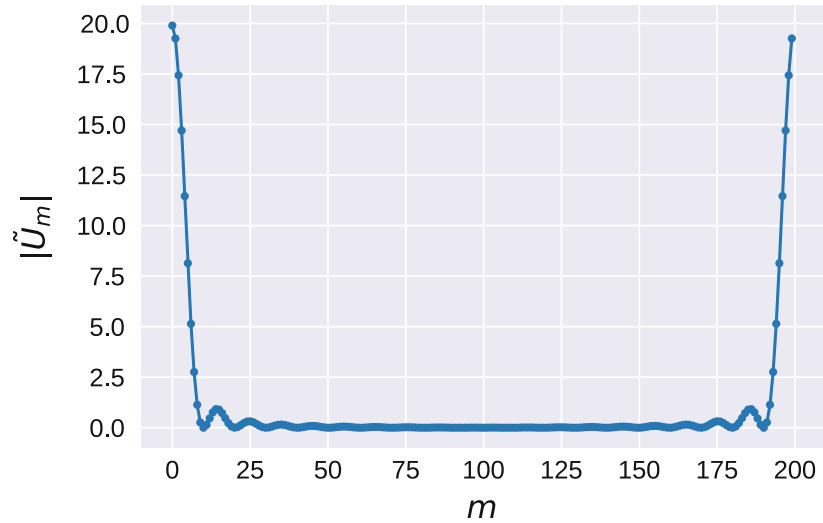
Plotting the magnitude of the output discrete Fourier transform  $|F_m|$  vs.  $m$ , we get the following plot:

## ► The matrix

$$W_{mn}^{(N)} = e^{-2\pi i m n / N}$$

is known as the **Fourier matrix**.

Why did we choose to implement this using NumPy broadcasting and matrix-vector multiplication, instead of using a for loop? As NumPy operations tend to be faster than the equivalent Python iteration, this could save significant computational time when working with large arrays.



**Figure 8.2** Magnitude of the output discrete Fourier transform

Comparing this to Fig. 8.1, we see an obvious problem — the wavenumber ‘peak’ is shifted from the center of the domain to the left and right boundaries! What is going on here? This is a major roadblock to using the DFT to approximate the Fourier transforms used in quantum physics. To find out why, let’s consider something we saw back in the very first section — the Nyquist theorem.

### 8.3.3 One-Sidedness

When we introduced the Fourier series, the infinite summation was centered around  $k = 0$  (the **zero momentum term**), and the Fourier coefficient integral was centered around  $x = 0$ . Likewise, the Fourier transform integrals were centered around the points  $x = 0$ ,  $k = 0$ . We call these **two-sided** or **centered**.

However, the DFT differs in this regard; the summation is only over positive values of  $n$  and  $m$ . The DFT is known as a **one-sided** Fourier transform. When calculating the DFT, it is the *first term* (the  $k_0$  term) which corresponds to the *zero wavenumber* term, not the central term! Subsequent terms represent the contributions of higher momentum components — with a significant caveat, as we'll see below.

#### Interpreting the DFT Momentum Terms When $f_n$ is Real

If we assume that  $f_n$  is a real function, then straightaway we can see that

$$F_0 = \sum_{n=0}^{N-1} f_n \in \mathbb{R}. \quad (8.33)$$

The remaining terms  $1 \leq m \leq N - 1$  can be analysed by considering  $F_{N-m}$ :

$$\begin{aligned} F_{N-m} &= \sum_{n=0}^{N-1} f_n e^{-2\pi i(N-m)n/N} = \sum_{n=0}^{N-1} f_n e^{2\pi imn/N} \overline{e^{-2\pi in}} \\ &= \left( \sum_{n=0}^{N-1} f_n e^{-2\pi imn/N} \right)^* = F_m^*. \end{aligned} \quad (8.34)$$

That is,  $F_{N-m}$  and  $F_m$  are *complex conjugates*. From here, we must consider separately the cases where  $N$  is even and  $N$  is odd.

- If  $N$  is **even**, then  $N/2$  is an integer and the midpoint of the DFT:

$$F_{N/2} = \sum_{n=0}^{N-1} f_n e^{-i\pi n} = \sum_{n=0}^{N-1} f_n (-1)^n \in \mathbb{R}.$$

- We have  $N/2 + 1$  unique DFT terms:  $0 \leq m \leq N/2$ .
- The remaining  $N/2 - 1$  terms are the conjugates of this range, in reverse order and excluding  $m = 0$ .
- If  $N$  is **odd**, then  $N/2$  is not an integer and is an invalid value for  $m$ . Instead, the two ‘midpoint’ coefficients are  $m = (N-1)/2$  and  $m = (N-1)/2 + 1 = (N+1)/2$ .
  - We have  $(N+1)/2$  unique DFT terms:  $0 \leq m \leq (N-1)/2$ .
  - The remaining  $(N-1)/2$  terms are the conjugates of these unique values, in reverse order and excluding  $m = 0$ .

► The Fourier transform used in quantum physics integrates from  $k \in [-\infty, \infty]$ , centered around  $k = 0$ , so it is also **two-sided**.

► In terms of the discrete position and wavenumber variables  $x_n = n$  and  $k_m = m\Delta k$ , this is a bit clearer to see:

The summations takes place over  $0 \leq x_n \leq (N-1)\Delta x$  and  $0 \leq k_m \leq (N-1)\Delta k$ .

Since approximately half of the DFT output is simply the complex conjugate of the other half, this provides us with no additional information about the wavelength components of the system under analysis! However, this does give us insight into an important result regarding discrete Fourier transforms.

Depending on whether  $N$  is even or odd, the above analysis allows us to determine the **maximum wavenumber component** that the DFT provides information for:

$$\begin{aligned} N \text{ even : } & \frac{N}{2} \\ N \text{ odd : } & \frac{N-1}{2} \approx \frac{N}{2} \text{ for large } N. \end{aligned} \quad (8.35)$$

This is the **Nyquist wavenumber** for the DFT, and generalises to cases where  $f_n$  is also complex.

What if we have some prior knowledge that the system under investigation is such that, for all  $m > m_c$ ,  $F_m = 0$ ? In addition, suppose that we know the value of  $m_c$ . In such a case, we can see from the above that we simply need to choose

$$N \geq \frac{1}{2m_c} \quad (8.36)$$

to ensure that we recover *all* the momentum-space information of the system when performing the DFT.

**Theorem 8.2.** *If a discrete function  $f_n$  contains no wavenumber components higher than  $m_c$ , then its DFT is completely determined by sampling using exactly  $N = 1/2m_c$  samples.*

Does this sound a bit too familiar? This is simply the DFT equivalent of the Nyquist condition  $\Delta x \leq \pi/k_{max}$  we derived back when we discretised the quantum Fourier transform integral. However, rather than simply denoting the *boundaries* of the two-sided Fourier domain, the Nyquist wavenumber in the one-sided case occurs in the *center* of the domain, reducing by half the information we have.

### 8.3.4 The Centered DFT

Recall our discretised quantum Fourier transform integrals:

$$\phi_m = \frac{\Delta x}{\sqrt{2\pi}} \sum_{n=0}^{N-1} \psi_n e^{-ik_m x_n} \Leftrightarrow \psi_n = \frac{\Delta k}{\sqrt{2\pi}} \sum_{m=0}^{N-1} \phi_m e^{ik_m x_n}, \quad (8.37)$$

where the discretised position and momentum grids are given by

$$\begin{aligned} x_n &= n\Delta x + x_0, & x_{N-1} &= -x_0 & \Rightarrow \psi(x_n, t) &= \psi_n \\ k_m &= m\Delta k + k_0, & k_{N-1} &= -k_0 & \Rightarrow \phi(k_m, t) &= \phi_m \end{aligned} \quad (8.38)$$

with  $\Delta k = 2\pi/N\Delta x$ . Using the Nyquist condition, we also find that

$$\begin{aligned} x_{N-1} &= (N-1)\Delta x + x_0 = -x_0 & \Rightarrow x_0 &= -\frac{1}{2}(N-1)\Delta x \\ k_{N-1} &= (N-1)\Delta k + k_0 = -k_0 & \Rightarrow k_0 &= -\frac{(N-1)\pi}{N\Delta x} \end{aligned} \quad (8.39)$$

allowing us to write the discretised Fourier transform in dimensionless form, purely in terms of the indices  $n$  and  $m$ ,

$$\begin{aligned} \phi_m &= \sum_{n=0}^{N-1} \phi_n e^{-2\pi i(m-a)(n-a)/N} \\ \psi_n &= \frac{1}{N} \sum_{m=0}^{N-1} \phi_m e^{2\pi i(m-a)(n-a)/N} \end{aligned} \quad (8.40)$$

where  $a = -x_0/\Delta x = -k_0/\Delta k = (N-1)/2$ .

While this is two-sided, and not one-sided like the DFT — resulting in a distinctly different function discretisation — can we write this in terms of the DFT due to its similarity? Let's try that.

Expanding out the bracketed terms, and extracting the term that is not dependent on the summation variable  $n$ , we get

$$\phi_m = e^{2\pi i(m-a)a/N} \sum_{n=0}^{N-1} [\psi_n e^{2\pi ian/N}] e^{-2\pi imn/N}. \quad (8.41)$$

This looks very similar to the one-sided DFT, with the presence of two phase factors! In fact, this is known as the **centered DFT**. We can use this result to approximate the Fourier transform used in quantum physics, using the more common one-sided DFT, by applying the two ‘phase-shifts’ we determined above:

► While this looks similar to the one-sided DFT, remember that we distinguish between them as  $\psi_n$  is discretised differently from  $f_n$ .

In the one-sided DFT,  $f_0$  corresponds to the point  $x = 0$  and  $F_0$  corresponds to the zero-wavenumber term.

In the centered or two-sided discretisation, however, the midpoint of  $\psi_n$  corresponds to the point  $x = 0$ , and the midpoint of  $\phi_m$  corresponds to the point  $k = 0$ .

### The centered DFT

1. Discretise  $\psi(x)$  symmetrically around  $x = 0$  to generate  $\psi_n$
2. Let  $f_n = e^{\pi i(N-1)n/N} \psi_n$
3. Calculate  $F_m$ , the one-sided DFT of  $f_n$
4. The centered discretised Fourier transform is then given by  $\phi_m = e^{\pi i[m-(N-1)/2](N-1)/N} F_m$

To perform the inverse centered DFT, these steps can simply be performed in reverse order.

---

### Shifting

While the approach above is a much more reasonable approximation to the continuous Fourier transform than the one-sided DFT, in practice the one-sided DFT is so common that it is generally a reasonable approximation to simply calculate the one-sided DFT, and then ‘circularly shift’ its values  $(N - 1)/2$  values to the right. The zero-momentum term is now approximately centered in the domain, and

$$|\phi_m| \approx |F_m^{(shift)}| = \begin{cases} |F_{N/2+m}|, & 0 \leq m < N/2 \\ |F_{m-N/2}|, & N/2 \leq m \leq N-1 \end{cases}$$

Note that, as we are not performing the appropriate complex phase shifts, just shifting the values of DFT, this will result in a different complex phase as

$$\phi_m \approx e^{i\theta_m} \tilde{F}_m^{(shift)}.$$


---

**Example 8.7** Centered Fourier Transform (Fortran)

Modify the previous program to calculate the *centered* DFT of the unit triangle function. Compare this to the shifted one-sided DFT.

**Solution:** For the shifted DFT, we can simply shift each element of the one-sided DFT from the previous example using the `cshift` function. To center the momentum as best as we can, we shift each element to the right by  $N/2$ . For the centered DFT, we can implement Eq. 8.40 more or less directly.

```
program centeredddft
    implicit none
    complex(8), parameter :: j=(0.d0,1.d0)
    real(8), parameter   :: pi=4.d0*atan(1.d0)
    integer    :: n, m, NN
    real(8)    :: dx, x, f(200), a
    complex(8) :: Fs(200), Fc(200)

    NN = 200
    dx = (5.d0-(-5.d0))/(NN-1)

    ! define the discretised unit triangle function
    f = 0.d0; x = -5.d0
    do n=1,NN
        if (-1<x .and. x<0) then
            f(n) = x + 1
        else if (0<=x .and. x<1) then
            f(n) = 1 - x
        end if
        x = x + dx
    end do

    ! perform the DFT
    Us = 0.d0
    do m=0,NN-1           ! calculate the mth component
        do n=0, NN-1       ! sum over all n
            Fs(m+1) = Fs(m+1) + f(n+1)*exp(-2*j*pi*m*n/NN)
        end do
    end do

    ! shift the Nyquist momentum to the center
    Us = cshift(Us, int(NN/2))

    ! perform the centered DFT
    a = (NN-1)/2; Uc = 0.d0
    do m=0,NN-1           ! calculate the mth component
        do n=0, NN-1       ! sum over all n
            Fc(m+1) = Fc(m+1) + f(n+1)*exp(-2*j*pi*(m-a)*(n-a)/NN)
        end do
    end do
end program centeredddft
```

**Example 8.8** Centered Fourier Transform (Python)

Modify the previous program to calculate the *centered* DFT of the unit triangle function. Compare this to the shifted one-sided DFT.

**Solution:** For the shifted DFT, we can simply shift each element of the one-sided DFT from the previous example using the `numpy.roll` function. To center the momentum as best as we can, we shift each element to the right by  $N/2$ . For the centered DFT, we can implement Eq. 8.40 more or less directly.

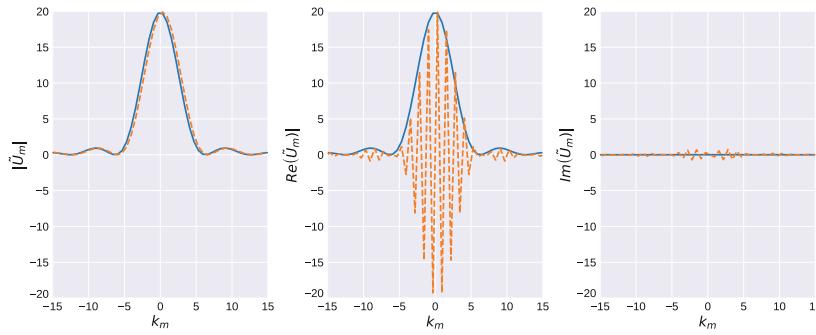
```
# define the x grid and n values
N = 200
dx = (5-(-5))/(N-1)
xgrid = np.arange(-5, 5+dx, dx)

# define the discretised unit triangle function
f = np.zeros(xgrid.shape)
for i, x in enumerate(xgrid):
    if -1 < x < 0:
        f[i] = x + 1
    elif 0 <= x < 1:
        f[i] = 1 - x

n = np.arange(N)[:, None]
m = np.arange(N)[None, :]

# Perform the one-sided DFT using NumPy broadcasting
# and shift the Nyquist momentum to the center
W = np.exp(-2j*np.pi*m*n/N)
Fs = np.roll(np.dot(W, f), N//2)

# Perform the centered DFT using NumPy broadcasting
a = (N-1)/2
W = np.exp(-2j*np.pi*(m-a)*(n-a)/N)
Fc = np.dot(W, f)
```



**Figure 8.3** Blue: the centered DFT. Orange: the shifted one-sided DFT

Plotting the absolute value, real part, and imaginary part of  $F_m^{(centered)}$  and  $F_m^{(shifted)}$ , we can see that the absolute values almost agree — while the centered DFT is centered exactly at the origin (as we expect from the continuous Fourier transform), the shifted one-sided DFT is slightly offset. Comparing the real and imaginary values, however, we see a major difference! The centered DFT is purely real and positive, whereas the shifted one-sided DFT oscillates wildly between negative and positive real values, and has a non-zero complex phase.

## 8.4 Errors: Aliasing and Leaking

We mentioned earlier that the DFT is surprisingly unique compared to other numerical methods we have covered so far — rather than existing merely as a tool to numerically approximate an unknown quantity (such as the eigenvalue, differentiation, or integration techniques of previous chapters), the DFT is an important transform in its own right, and sits right alongside the continuous Fourier transform and Fourier series. In some real-world applications, for example signal analysis and processing, we run into problems where discrete periodic signals need to be analysed — and the DFT is perfect for the job.

Because of this, we haven't really delved into error analysis as we did in previous chapters; in a lot of cases, the DFT provides what is essentially an exact result, with no parameters to modify that can increase the ‘accuracy’ per se. However, there are several cases where we can make some qualitative comments regarding the accuracy of the DFT.

### 8.4.1 Aliasing

We haven't seen the term aliasing before, however we have already explored the underlying cause. Aliasing error occurs when  $\Delta x$  is not sufficiently small such that the Nyquist theorem is not satisfied:  $\Delta x > \pi/k_{max}$ . Fourier coefficients corresponding to wavenumbers larger than the Nyquist wavenumber therefore won't be adequately sampled.

However, the under-sampled Fourier components will not simply be missing from the resulting DFT. Instead, they will show up as ‘incorrect’ lower wavenumber components! This is what's known as aliasing — echoes of coefficients from above the Nyquist wavenumber showing up at lower wavenumbers. To avoid the effects of aliasing, a simple fix is to decrease  $\Delta x$ , in other words, increase the sampling frequency.

### 8.4.2 Leakage

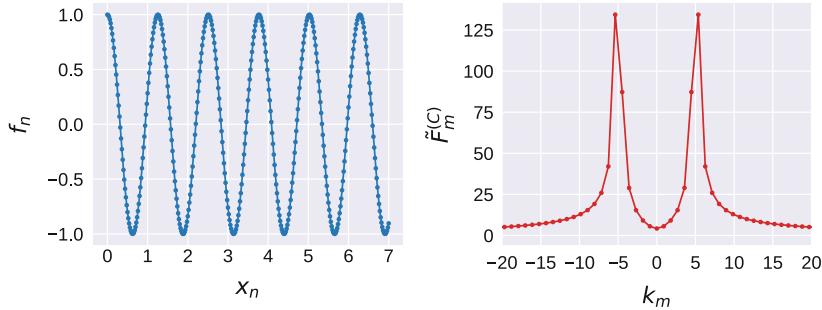
Leakage occurs when the discrete function  $f_n$  is discontinuous at the sampled boundary,

$$|f_n - f_{N-1}| \gg 0, \quad (8.42)$$

which the DFT then treats as a discontinuous  $N$ -periodic function. This can occur in cases where  $f_n$  is not periodic, has been sampled/discretised over a domain smaller than a full period, or the discretisation domain is over a non-integer number of periods. This periodic discontinuity causes non-zero Fourier components corresponding to particular momentum values  $k_m$  to ‘leak’ their amplitude into neighbouring coefficients — resulting in a **smearing** of the momentum-space Fourier components.

To get a better idea of what happens in the case of leakage, consider the function  $f_n = \cos(5x_n)$ , which has been discretised over  $0 \leq x_n \leq 7$  with

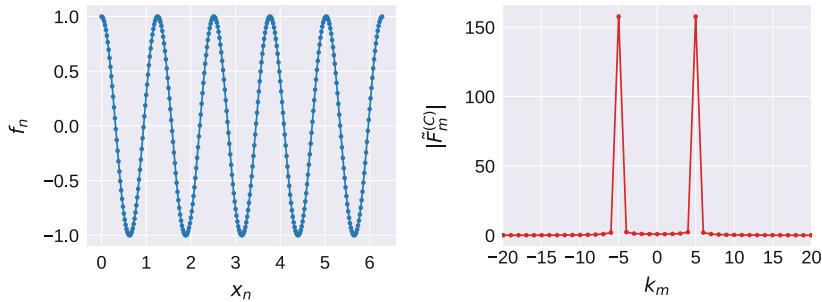
$\Delta x = x_{n+1} - x_n = 0.1$ . Since  $f_n$  is a periodic function with wavenumber  $k = 2\pi/\lambda = 5$ , we would expect it to have non-zero Fourier components where  $k_m = \pm 5$ , and zero elsewhere. Taking the centered DFT of  $f_n$ , let's see what happens.



**Figure 8.4**

While we can definitely identify the peaks corresponding to  $k_m = \pm 5$ , we can also spot some leaking occurring in the neighbourhood of the peaks — the discontinuity in the period of  $f_n$  is causing additional momentum values to appear in the DFT.

To try and reduce the leaking error, let's attempt a new discretisation,  $0 \leq x_n < 2\pi$ :



**Figure 8.5**

Since we are performing the centered DFT over an entire period,  $f_n$  is now continuous across the boundary;  $f_0 \approx f_{N-1}$ . As a result, the leaking is completely eliminated, and we now have two perfectly defined peaks at  $k_m = \pm 5$ , with  $k_m = 0$  for all other values of  $m$ .

However, there may be cases when we don't know the period of  $f_n$  in advance, or perhaps we don't know if  $f_n$  is periodic at all. In these cases, we can perform a small manipulation to 'enforce' periodicity in the DFT domain. By multiplying  $f_n$  by a so-called **window function**, we cause the value of the function near the boundaries to rapidly approach zero, and thus  $f_0 = f_{N-1} = 0$ .

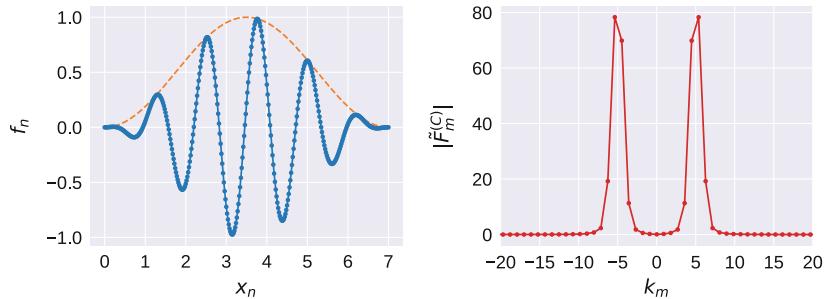


Figure 8.6

In the above example, we have returned to the discretisation domain  $0 \leq x \leq 7$ , but this time multiplied  $f_n$  by the *Hann window function*,

$$W(n) = \frac{1}{2} - \frac{1}{2} \cos\left(\frac{2\pi}{L}(x + x_0)\right), \quad x_0 \leq x \leq x_N \quad (8.43)$$

in order to smooth the boundary discontinuities. Compare the result to Figs. 8.4 and 8.5 — although the peaks in the wavenumber domain aren't as clear and distinct as when we chose the domain such that  $f_n$  was continuous across the boundary, the leakage has been significantly reduced compared to the case with no windowing.

Note that there are many possible choices for window functions, with the ultimate choice depending on the system under study.

### Quantum wavefunctions and leaking

We will see this in more detail later, but it turns out that bound solutions to the Schrödinger equation (i.e. those in the presence of a *potential*) must be **square integrable** — the Hilbert space in which they reside does not allow such solutions to blow up to infinity. Instead, they must decay to zero as  $x \rightarrow \pm\infty$ .

As a result, provided we ensure that we choose a large enough domain for the Fourier transform, we do not need to worry about leaking and windowing; the wavefunction will be ‘naturally’ windowed.

## 8.5 The Fast Fourier Transform

While we definitely require accurate numerical techniques when solving problems in quantum physics, it is important to remember that we would also like them to be relatively *efficient*. An accurate algorithm that takes exponentially long to return its result becomes exponentially less useful! So how does the DFT perform?

When considering the *scaling* of a numerical algorithm, it is often useful to consider directly what your computer is processing. By breaking an algorithm into a list of elementary operations such as addition and multiplication, we can get an idea of how the algorithm scales as the algorithm input increases. Consider an  $N$ -point DFT, acting on an input  $f_n$  of size  $N$  (that is,  $n = 0, \dots, N - 1$ ):

$$F_m = \sum_{n=0}^{N-1} f_n e^{-2\pi i m n / N}. \quad (8.44)$$

For each value of  $m$ , we perform  $N$  multiplications of  $f_n$  with  $e^{-2\pi i m n / N}$ . We need to repeat this another  $N$  times, for each value of  $m = 0, \dots, N - 1$ , so in total we are performing  $N^2$  multiplications. As a result, we say that the DFT scales like  $\sim N^2$  with input of size  $N$  — this is commonly written in Big-O notation as  $\mathcal{O}(N^2)$ .

However, there is a huge number of repeated calculations here that are redundant, for instance, cases where the product of the summation integers  $m$  and  $n$  repeat — and that's even before taking into account the periodicity of the complex exponential. Taking advantage of such recurring symmetries, we can manipulate the DFT to remove those redundant calculations, resulting in an algorithm that is significantly more efficient — the Fast Fourier Transform or FFT.

▶ See Sect. 4.1 for more details on Big-O notation. In this section, we will primarily use it to describe computational complexity.

### History of the Fast Fourier Transform

The most common implementation of the FFT is known almost synonymously as the Cooley–Tukey algorithm, named after the American mathematicians J. W. Cooley and John Tukey. Tasked in 1965 with analysing discrete time-series data of seismological events in order to detect the use of nuclear weapons, but limited by the efficiency of the DFT, Cooley and Tukey developed and published the first modern implementation of the FFT.

Strangely enough, though, this was not the first time the FFT had been discovered! In 1805, a young Carl Friedrich Gauss — the renowned German polymath who has lent his name to concepts in astronomy, electromagnetism, and mathematics<sup>a</sup> — was attempting to approximate the orbit of the dwarf planet Ceres, discovered only that year. Using

the tools available at the time, his approach involved approximating the orbital mechanics via ‘trigonometric interpolation’; what we now know as the DFT<sup>b</sup>. Attempting to efficiently apply it to his data set, he describes in detail the same approach as Cooley and Tukey — only 160 years prior.

Unfortunately, Gauss never explored the computational efficiency of his algorithm further, and the work was only published posthumously in 1866<sup>c</sup>. By then, Newtonian solutions to orbital mechanics problems were the preferred approach, and the publication of his work from 60 years earlier garnered little interest. It wasn’t even until 20 years *after* the publication of the Cooley–Tukey paper that Gauss’s contribution was finally noticed.

<sup>a</sup>Some of Gauss’s more well-known contributions to the fields of mathematics and physics have since taken his name, including Gaussian elimination, Gaussian or normal distributions, and the Gaussian-Legendre quadratures we examined back in Chap. 6.

<sup>b</sup>This was two years *before* Joseph Fourier published his seminal work on Fourier analysis, which introduced the Fourier series!

<sup>c</sup>Gauss was a known perfectionist, and his personal motto regarding publications was *pauca sed matura* (“few, but ripe”).

### 8.5.1 The Cooley–Tukey Algorithm

Let’s use a couple of the symmetries we’ve already explored in the discrete Fourier transform to see if we can reduce the number of computational operations required to calculate the DFT. We’ll start with the DFT as defined previously, but with the additional constraint that the input  $f_n$  has size  $n = 0, \dots, N - 1 = 2^b$ . That is, the DFT is performed on a **power-of-two** number of data points.

To start with, let’s separate the DFT into two separate summations — one over the even values  $n = 2\ell$ , and one over the odd values  $n = 2\ell + 1$ :

$$\begin{aligned} F_m &= \sum_{\ell=0}^{N/2-1} f_{2\ell} e^{-2\pi i(2\ell)m/N} + \sum_{\ell=0}^{N/2-1} f_{2\ell+1} e^{-2\pi i(2\ell+1)m/N}, \\ &= \sum_{\ell=0}^{N/2-1} f_{2\ell} e^{-2\pi i\ell m/(N/2)} + e^{-2\pi im/N} \sum_{\ell=0}^{N/2-1} f_{2\ell+1} e^{-2\pi i\ell m/(N/2)}, \\ &= A_m + e^{-2\pi im/N} B_m. \end{aligned} \tag{8.45}$$

Here,  $A_m$  and  $B_m$  are two DFTs of length  $N/2$  — so we have reduced the problem of calculating an  $N$ -point DFT down to solving two  $N/2$ -point transforms. However, we don’t need to stop here; this exposes even more symmetries that we can exploit. For instance, recall the periodicity condition of the DFT coefficients,  $F_{m\pm N} = F_m$ . Thus, we must have  $A_{m\pm N/2} = A_m$  and

$B_{m \pm N/2} = B_m$ , and we can rewrite the above expression as

$$F_m = \begin{cases} A_m + e^{-2\pi im/N} B_m, & 0 \leq m < N/2 \\ A_{m-N/2} + e^{-2\pi im/N} B_{m-N/2}, & N/2 \leq m \leq N-1 \end{cases}. \quad (8.46)$$

(Apply the periodicity condition to convince yourself this is the case.) This is quite a remarkable result, as we can now see clearly that we are simply repeating the same transform twice, just with a slight change in phase! To see this more clearly, we can do a variable rescaling  $m \rightarrow m + N/2$  on the bottom transform:

$$\begin{cases} F_m &= A_m + e^{-2\pi im/N} B_m, & 0 \leq m \leq N/2 - 1, \\ F_{m+N/2} &= A_m - e^{-2\pi im/N} B_m \end{cases}, \quad (8.47)$$

where we have used  $e^{-2\pi i(m+N/2)/N} = e^{-i\pi} e^{-2\pi im/N} = -e^{-2\pi im/N}$ . This might not be completely intuitive, so let's have a look at a quick example.

### Decomposing the DFT

Let's consider as input, a discretised function  $f_n$  with  $N = 8$  data points, denoted by  $f_0, f_1, \dots, f_7$ . To start with, we'll calculate the DFTs of the even and odd data points separately:

$$\begin{aligned} A_m &= f_0 + f_2 e^{-2\pi m/4} + f_4 e^{-4\pi m/4} + f_6 e^{-6\pi m/4} \\ B_m &= f_1 + f_3 e^{-2\pi m/4} + f_5 e^{-4\pi m/4} + f_7 e^{-6\pi m/4} \end{aligned} \quad (8.48)$$

Next, we need to sum the components of the two DFTs, making sure to use the correct phase as per Eq. 8.47:

$$\begin{aligned} F_0 &= A_0 + B_0 & F_4 &= A_0 - B_0 \\ F_1 &= A_1 + e^{-2\pi i/8} B_1 & F_5 &= A_1 - e^{-2\pi i/8} B_1 \\ F_2 &= A_2 + e^{-4\pi i/8} B_2 & F_6 &= A_2 - e^{-4\pi i/8} B_2 \\ F_3 &= A_3 + e^{-6\pi i/8} B_3 & F_7 &= A_3 - e^{-6\pi i/8} B_3 \end{aligned} \quad (8.49)$$

Let's see if we can determine the efficiency of this approach. We are calculating two DFTs using half the input data, so  $2 \times (N/2)^2 = N^2/2$  multiplications. In order to combine them to determine all the discrete Fourier transform coefficients  $F_m$ , we then perform another  $N$  multiplications, bringing the total number of multiplications to  $N + N^2/2$ . Compared to applying the DFT equation directly, we have reduced the number of operations by  $N^2 - (N + N^2/2) = \frac{1}{2}N(N-2)$ , or 24 in this  $N = 8$  case.

A useful tool for visualising this process is the **butterfly diagram**, shown below. The butterfly diagram shows how the input data points  $f_n$  are transformed to give the transformed terms  $F_m$ .

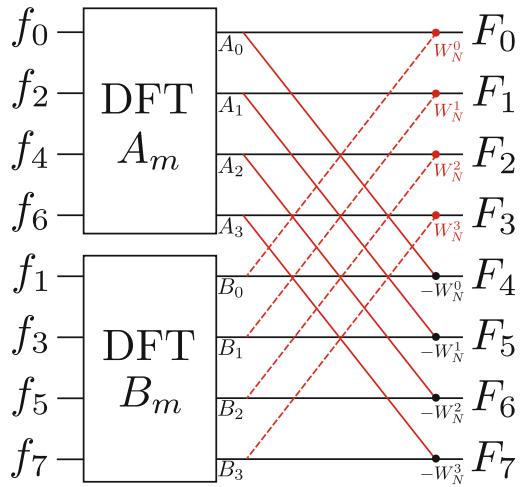


Figure 8.7

There are two simple rules to reading the butterfly diagram:

- the summation of terms is denoted by two lines joining together, and
- multiplication of a term by a constant is denoted by a filled circle, with the constant written below. In our example, colors are used to distinguish which term is multiplied before the summation.

Have a go comparing the butterfly diagram to Eq. 8.49, and try matching the equations; for example, by following the downward red arrow from  $A_0$ , we can see that  $F_4 = A_0 - W_N^0 B_0$ , where we have used the shorthand

$$W_N^m = e^{2\pi im/N}. \quad (8.50)$$

► The shorthand  $W_N^m$  are sometimes referred to as **twiddle factors**.

While we have reduced the number of multiplications, this slightly more efficient algorithm still scales similarly to  $\mathcal{O}(N^2)$ . This is due to the presence of the two stock-standard DFTs required to calculate  $A_m$  and  $B_m$ , which still scale quadratically in  $N$ . Can we make this process even more efficient?

We can! The trick is simply to continue applying this process recursively, until there are no longer any DFTs that we can divide. In this case, we can apply this algorithm to the 4-point DFTs  $A_m$  and  $B_m$  — dividing each into *two* 2-point DFTs, acting on the even and odd coefficients as before:

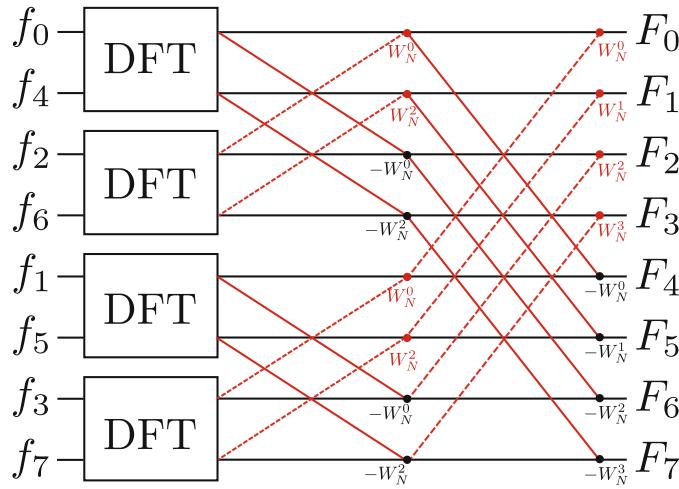


Figure 8.8

Following the same formulation as above, since we are now starting with a  $N/2$ -point DFT, you might expect twiddle factors of the form  $W_{N/2}^m$  to appear. However, we are able to use the property  $W_{N/2}^m = W_N^{2m}$  so that all twiddle factors have the same subscript — this is a useful trick, as the fewer distinct twiddle factors that appear, the less computation we need to do.

We now have reduced the computation down to 4 separate 2-point DFTs — however, 2-point DFTs are simple enough in form that they can be written directly into the butterfly diagram:

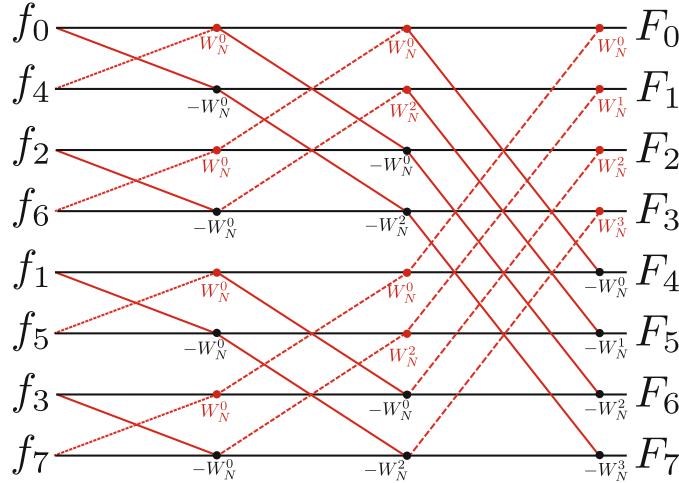


Figure 8.9

- ▶ Note that the order of the input terms  $f_n$  have been rearranged from the previous butterfly diagram, to group the new ‘odd’ and ‘even’ terms.

This is as far as we can go with this approach; so let’s see if the effort has afforded us any additional increase in efficiency. Recall that we restricted our initial DFT to size  $N = 2^b$  — this can be repeatedly divided into two equal

DFTs a total of  $\log_2(N) = b$  times (in this example, we had  $N = 8 = 2^3$ , and managed to divide it 3 times). Once the DFT has been divided into  $\log_2(N)$  ‘stages’, from looking at the final butterfly diagram, we can see that each stage has  $N$  multiplications. Putting this altogether, the Cooley–Tukey algorithm therefore scales like

$$\mathcal{O}(N \log_2(N)). \quad (8.51)$$

This is a *momentous* result! To get an idea of just how momentous, consider a DFT on  $N = 2^{10} = 1024$  data points. Using the DFT formula directly, this requires  $N^2 = 1048576$  operations to compute. Alternatively, using the Cooley–Tukey FFT, we only need  $N \log_2(N) = 10240$  operations, a reduction of more than two orders of magnitude; this corresponds to an over 100 times speedup.

### FFT: Beyond powers of two

In this section, we considered the case where the FFT is constrained to an input size  $N = 2^b$ , with the resulting FFT known as the **radix-2 Cooley–Tukey decimation-in-time** algorithm<sup>a</sup>. In general, if we are generating the samples  $f_n$  from a continuous function, this is fine — we can just ensure that we sample at a number of data points satisfying this constraint. But what if we have a fixed set of data points not of this form?

One approach is to allow the radix to vary at each stage. For example, consider  $N = 96$ ; this has a prime number factorisation  $96 = 2^5 \times 3^1$ , allowing 5 radix-2 stages or decimations in time, before finishing with a final stage of 3-point DFTs (i.e. radix-3). In fact, there is no constraint on *how* we must perform the decimation in time — if the  $N$ -point DFT is such that  $N = N_1 N_2$ , where  $N_1$  and  $N_2$  are not necessarily prime, then it is possible to use radix- $N_1$  to divide the DFT into  $N_1$  DFTs each of size  $N_2$ .

The optimisation doesn’t end there. Depending on the programming language used, there are a host of other steps that can be taken — such as performing the FFT decimation steps ‘in-place’ in the array of the original input data, to tricks for quickly sorting to reorder the data points, designed to achieve the maximum possible efficiency from the FFT.

---

<sup>a</sup>radix-2 since we divide each stage into 2, and decimation in time since it is the input ‘time-domain’ data points that end up getting reordered.

**Example 8.9** Recursive Fast Fourier Transform (Fortran)

Write a function that calculates the Fast Fourier Transform. Then, use it to find the momentum-space representation  $\phi(k)$  of the wavefunction  $\psi(x) = e^{-x^2/2} \sin(4x)$  over the domain  $-5 \leq x \leq 5$ .

**Solution:** We can use the radix-2 Cooley–Tukey algorithm to define our FFT function. Then, discretising the  $x$ -grid into  $N = 2^8 = 256$  discrete points, we can use the FFT to calculate the approximate Fourier transform of  $\psi(x)$  by calculating the centered DFT described Sect. 8.3.4.

```
module fft
    implicit none
    complex(8), parameter :: j=(0.d0, 1.d0)
    real(8), parameter :: pi=4.d0*atan(1.d0)

    contains

        recursive function FFT2(fn, NN) result(Fm)
            integer, intent(in) :: NN
            complex(8), intent(in) :: fn(NN)
            complex(8) :: Fm(NN)

            ! local variables
            integer :: n, m
            complex(8) :: Am, Bm, W

            if (NN == 1) then
                ! DFT is length 1
                Fm(1) = fn(1)
            else if (mod(NN, 2) /= 0) then
                ! DFT is not divisible by 2,
                ! perform standard DFT algorithm
                do m=0,NN-1 ! calculate the mth component
                    do n=0, NN-1 ! sum over all n
                        Fm(m+1) = Fm(m+1)+fn(n+1)*exp(-2*j*pi*m*n/NN)
                    end do
                end do
            else
                ! Divide the DFT using radix-2 Cooley-Tukey
                Fm(:NN/2) = FFT2(fn(:2), NN/2) ! Even terms DFT
                Fm(NN/2+1:) = FFT2(fn(2::2), NN/2) ! Odd terms DFT
                ! combine the two DFTs using twiddle factors
                do m=0, NN/2-1
                    Am = Fm(m+1)
                    Bm = Fm(NN/2+m+1)
                    W = exp(-2*j*pi*m/NN)
                    Fm(m+1) = Am + Bm*W
                    Fm(NN/2+m+1) = Am - Bm*W
                end do
            end if
        end function FFT2
end module fft
```

- A recursive function or subroutine suits the Cooley–Tukey algorithm. In this example to the left, our approach above has been directly translated into Fortran code.

Firstly, if the DFT is of length one, we simply return the input value, since the output of a 1-point DFT is simply the input:  $F_0 = e^{-2\pi i 0/1} f_0 = f_0$ .

Next, if the DFT is not divisible by two, we simply perform the standard DFT algorithm. This could be modified to use radix-3 if the DFT length is a factor of 3.

Finally, if the length of the DFT is divisible by two, we recursively find the FFT of the odd and even terms, and combine them using twiddle factors.

```
program centeredFFT
  use fft
  implicit none
  integer :: n, m, NN
  real(8) :: a, dx, x, x0, dk, k0
  complex(8) :: psi(256), phi(256)

  ! x grid
  NN = 256
  a = (NN-1)/2.
  x0 = -5.
  dx = abs(2*x0)/(NN-1)

  ! create the discrete wavefunction
  do n=0, NN-1
    psi(n+1) = exp(-(x0+n*dx)**2/2.)*sin(4.*((x0+n*dx)))
    ! centered -> one-sided phase shift
    psi(n+1) = exp(2*j*pi*a*n/NN)*psi(n+1)
  end do

  ! perform the one-sided FFT
  phi = FFT2(psi, NN)

  ! one-sided -> centered phase shift
  do m=0, NN-1
    phi(m+1) = exp(2*j*pi*a*(m-a)/NN)*phi(m+1)
  end do

  ! corresponding k grid
  k0 = -pi/dx
  dk = 2.*pi/(NN*dx)
end program centeredFFT
```

**Example 8.10** Recursive Fast Fourier Transform (Python)

Write a function that calculates the Fast Fourier Transform of a data set. Then, use it to find the momentum-space representation  $\phi(k)$  of the wavefunction  $\psi(x) = e^{-x^2/2} \sin(4x)$  over the domain  $-5 \leq x \leq 5$ .

**Solution:** We can use the radix-2 Cooley–Tukey algorithm to define our FFT function. Then, discretising the  $x$ -grid into  $N = 2^8 = 256$  discrete points, we can use the FFT to calculate the approximate Fourier transform of  $\psi(x)$  by calculating the centered DFT described in Sect. 8.3.4.

**Solution:**

```
# define the FFT function
def FFT2(f):
    N = len(f)
    if N == 1:
        # DFT is length 1
        F = f
    elif N % 2 != 0:
        # DFT is not divisible by 2, perform
        # standard DFT algorithm
        m = np.arange(N)[:, None]
        n = np.arange(N)[None, :]
        F = np.dot(np.exp(-2j*np.pi*m*n/N), f)
    else:
        # divide the DFT into two using radix-2 Cooley-Tukey
        Am = FFT2(f[::2])
        Bm = FFT2(f[1::2])
        # combine using twiddle factors
        m = np.arange(N/2)
        W = np.exp(-2j*np.pi*m/N)
        F = np.concatenate([Am + W*Bm, Am - W*Bm])
    return F

# define the properties
N = 256
a = (N-1)/2
dx = (5-(-5))/(N-1)

# define the x-grid and discretised wavefunction
x = np.arange(-5, 5+dx, dx)
psi = np.exp(-(x**2)/2)*np.sin(4*x)

# perform the centered-sided FFT
n = np.arange(N)
phi = FFT2(np.exp(2j*np.pi*a*n/N) * psi)
# one-sided -> centered phase shift
phi *= np.exp(2j*np.pi*a*(n-a)/N)

# corresponding k-grid
```

► A recursive function or subroutine suits the Cooley–Tukey algorithm. In this example to the left, our approach above has been directly translated into Fortran code.

Firstly, if the DFT is of length one, we simply return the input value, since the output of a 1-point DFT is simply the input:  $F_0 = e^{-2\pi i 0/1} f_0 = f_0$ .

Next, if the DFT is not divisible by two, we simply perform the standard DFT algorithm. This could be modified to use radix-3 if the DFT length is a factor of 3.

Finally, if the length of the DFT is divisible by two, we recursively find the FFT of the odd and even terms, and combine them using twiddle factors.

```
dk = 2*np.pi/(N*dx)
kmax = np.pi/dx
k = np.arange(-kmax, kmax, dk)
```

### NumPy vectorization

While the recursive Cooley–Tukey algorithm in the above example provides a reasonable implementation in Fortran, the equivalent Python example is not as computationally efficient, due to Python being an interpretative — rather than compiled — language. As mentioned in previous sections, one solution to dramatically increase the speed of the Python code is to write it such that all recursive Python function calls are instead replaced by NumPy vector operations and broadcasting. Since NumPy array manipulations are done “under-the-hood” in Fortran, this allows us to combine the speed of a compiled language (like Fortran) with the ease-of-use of Python.

To write the radix-2 Cooley–Tukey algorithm using NumPy vectorization and broadcasting, we need to flip our approach; rather than starting with the full  $N$ -point DFT, and recursively breaking it into smaller and smaller DFTs, we instead begin by performing all 2-point DFTs simultaneously. We do this by reshaping the input array into two rows, and apply the 2-point DFTs via matrix multiplication:

$$\begin{aligned} & [f_0 \ f_1 \ f_2 \ f_3 \ f_4 \ f_5 \ f_6 \ f_7] \\ \rightarrow & \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} f_0 & f_1 & f_2 & f_3 \\ f_4 & f_5 & f_6 & f_7 \end{bmatrix} = \begin{bmatrix} f_0 + f_4 & f_1 + f_5 & f_2 + f_6 & f_3 + f_7 \\ f_0 - f_4 & f_1 - f_5 & f_2 - f_6 & f_3 - f_7 \end{bmatrix} \end{aligned}$$

This result represents all  $N/2$ , 2-point DFTs as a  $2 \times N/2$  matrix. Now, we need to slowly, stage-by-stage, recombine all the terms with the twiddle factors, until we recover the final Fourier result. We do this by splitting the resulting array horizontally into two, multiplying the second part by the twiddle factors vector, and adding/subtracting as required:

$$\begin{aligned} & \left[ \begin{bmatrix} f_0 + f_4 & f_1 + f_5 \\ f_0 - f_4 & f_1 - f_5 \end{bmatrix} + \begin{bmatrix} W_2^0 \\ W_2^1 \end{bmatrix} \begin{bmatrix} f_2 + f_6 & f_3 + f_7 \\ f_2 - f_6 & f_3 - f_7 \end{bmatrix} \right] \\ & \left[ \begin{bmatrix} f_0 + f_4 & f_1 + f_5 \\ f_0 - f_4 & f_1 - f_5 \end{bmatrix} - \begin{bmatrix} W_2^0 \\ W_2^1 \end{bmatrix} \begin{bmatrix} f_2 + f_6 & f_3 + f_7 \\ f_2 - f_6 & f_3 - f_7 \end{bmatrix} \right] \end{aligned}$$

Notice that we have stacked the two results *vertically*. We now repeat the process for all remaining Cooley–Tukey stages until we have an  $N \times 1$  array, and we have recovered  $F_m$ .

The vectorized approach, due to thinking of all operations as multi-dimensional arrays acting on each other, is a bit harder to visualise than

the recursive approach. Compare the above with Fig. 8.8 to see that the terms of  $f_n$  are being operated on correctly.

The following Python and NumPy code implements the vectorised radix-2 Cooley–Tukey algorithm, assuming that the input is of size  $2^b$ :

```
def FFT2_numpy(f):
    # Compute all 2-point DFTs
    N = f.shape[0]
    W = np.array([[1, 1], [1, -1]])
    F = np.dot(W, f.reshape([2, -1]))

    # number of remaining Cooley-Tukey stages
    stages = int(np.log2(N))-1
    for i in range(stages):
        k = F.shape[0]    # size of the DFTs to combine
        n = F.shape[1]    # number of DFTs to combine
        Am = F[:, :n//2] # 'even' terms
        Bm = F[:, n//2:] # 'odd' terms
        twiddle = np.exp(-1.j*np.pi*np.arange(k)/k)[:, None]
        F = np.vstack([Am + twiddle*Bm,
                      Am - twiddle*Bm])

    return F.flatten()
```

While the Cooley–Tukey examples shown above are orders of magnitude faster than the naïve DFT implementations we began with based directly off the summation formula, there remain a multitude of further optimisations we could do to ensure that we are not making redundant calculations. For instance, the examples above are what are known as **out-of-place** FFT algorithms; we take an array input, and use that to construct a new transformed array. More efficient approaches involve **in-place** algorithms, where the elements of the input array are rearranged without needing to create new arrays.

Luckily, for both Fortran and Python, there exist standard libraries for calculating the FFT — with the goal of providing functions and subroutines that are as optimised as can be.

► To do this, these libraries apply almost every trick in the book; from breaking down the algorithms into minute pieces to ensure all loops are optimised and nothing is being recalculated, ensuring the processor cache is being used efficiently, to making sure the array elements are stored in memory in an order that allows for speedier manipulation — nothing is off limits.

## 8.6 Fortran: Using FFTW

In Fortran, there are several libraries that can be used to provide the FFT, and chief among them is FFTPACK — for decades the *de facto* leader of Fast Fourier Transforms in Fortran, and as ubiquitous as LAPACK. In 1997, however, there was a new kid on the block — FFTW, or the Fastest Fourier Transform in the West. Designed by mathematicians at the Massachusetts Institute of Technology, the speed increase over FFTPACK was so impressive, that the creators were awarded the J. H. Wilkinson Prize for Numerical Software. Nowadays, FFTW is the undisputed leader in FFT algorithms, and is used in other numerical computation packages such as MATLAB.

In this chapter, I'll introduce the main FFT subroutine provided in the Fortran interface to FFTW, which we will use later when analyzing the Schrödinger equation.

### 8.6.1 The FFTW Fortran Interface

- ▶ In this section, we will be using FFTW version 3. This is the latest version and the recommended version — version 2 and below use a slightly different interface in Fortran.

Once FFTW is installed, it is relatively easy to link it to your Fortran program. As FFTW is written in C, not Fortran, to ensure your program can access the FFTW functions and subroutines, you will need to load the standard C bindings, as well as the FFTW header file:

```
program main
  use iso_c_binding
  implicit none
  include "fftw3.f03"
  ! program code
end program main
```

Note the order of these three statements; **use iso\_c\_binding** must come first to load the type bindings used by C, followed by **implicit none**, and finally **include "fftw3.f03"**, which includes the FFTW interface in your program.

When compiling your program, you will also need to point the compiler to the location of the `fftw3.f03` file and the FFTW library. The exact locations will depend on your system and how you installed FFTW — for example, if you are using the GCC `gfortran` compiler on a Debian based system, the compilation step will look like this:

```
gfortran example.f90 -o example -I/usr/include -lfftw3
```

Here, the flag `-I/usr/include` tells the compiler to look for the file `fftw3.f03` in the directory `/usr/include`, and `-lfftw3` indicates to link the compiled program against the FFTW version 3 library.

### 8.6.2 One-Dimensional FFT

To achieve the phenomenal levels of optimisation that it does, FFTW approaches the FFT in a slightly more ‘quirky’ fashion than other FFT libraries do. To

get started with calculating an FFT, you first need to specify a **plan**. To generate a **plan**, FFTW requires both the input and output arrays; by studying the structure of the input data, it then formulates a **plan** for calculating the FFT in the most efficient way possible.

```
type(c_ptr) :: plan1
plan1 = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE)
```

In the above, we are using the function `fftw_plan_dft_1d` to generate a **plan** to calculate the one-dimensional discrete Fourier transform of the input complex data `in`, and use this to fill the dummy output complex array `out`. Note that both the input and output arrays need to be complex and one-dimensional, and of size `N`, matching the first argument to the function. There are also two additional options we need to provide:

- **Direction:** this indicates whether we would like to perform the discrete Fourier transform.
  - `FFTW_FORWARD`: the standard discrete Fourier transform.
  - `FFTW_BACKWARD`: the inverse discrete Fourier transform.
- **Planning style:** how FFTW determines the most efficient FFT algorithm to use for the plan.
  - `FFTW_ESTIMATE`: instructs FFTW to use simple heuristics to determine the best plan.
  - `FFTW_MEASURE`: instructs FFTW to compute various FFTs, and pick the optimum based on execution time.

While `FFTW_MEASURE` will often lead to optimised plans, it can take a bit longer, and will also overwrite the output array `out`.

Once the **plan** has been created, the FFT can then be performed by calling the `fftw_execute_dft` subroutine, and passing as arguments the **plan**, as well as the input and output arrays associated with the **plan**:

```
call fftw_execute_dft(plan1, in, out)
```

At this point, the (one-sided) FFT will have been performed, with the result stored in the array `out`. Finally, we can destroy the plan:

```
call fftw_destroy_plan(plan1)
```

Similarly to array deallocation, this helps with memory clean up, and reduces the chances of memory leaks.

- ▶ Since FFTW uses C internally, we need to declare the variable that stores the **plan** as a C pointer using `type(c_ptr)`.
- ▶ FFTW also supports FFTs on 2-dimensional and 3-dimensional arrays, through the **plan** functions `fftw_plan_dft_2d` and `fftw_plan_dft_3d` respectively.

**Example 8.11** Fast Fourier Transform in FFTW (Fortran)

Use FFTW to calculate the Fast Fourier Transform of the discrete function  $f_n = \sin(4n^2) \tan(n + 1)$  for  $n = 0, 1, \dots, 255$ .

**Solution:**

```
program fft
    use iso_c_binding
    implicit none
    include "fftw3.f03"
    integer, parameter :: N = 256
    integer :: i
    complex(8) :: fin(N), Fout(N)
    type(c_ptr) :: plan

    ! create the discrete function
    do i=1, N
        fin(i) = sin(4.d0*(i**2))*tan(i+1.d0)
    end do

    ! one-sided FFT
    plan = fftw_plan_dft_1d(N, fin, Fout, FFTW_FORWARD, FFTW_ESTIMATE)
    call fftw_execute_dft(plan, fin, Fout)
    call fftw_destroy_plan(plan)

    do i=1, N
        write(*,*)Fout(i)
    enddo
end program fft
```

---

**Normalization**


---

One thing to be aware of when using FFTW is that, by default, FFTW does not apply any normalisation when performing the FFT. That is, FFTW calculates the following DFT and inverse DFT respectively:

$$F_m = \sum_{n=0}^{N-1} f_n e^{-2\pi imn/N} \Leftrightarrow Nf_n = \sum_{m=0}^{N-1} F_m e^{2\pi imn/N}.$$

As a result, performing the FFT followed by the inverse FFT will not result in the original input data being returned, but the original input data scaled by  $N$ . This is deliberate — by ignoring the normalisation factor, FFTW is able to perform additional optimisations. So when performing the inverse transform, you must remember to divide the FFTW result by  $N$ .

---

## 8.7 Python: Using SciPy

If you're using Python, the SciPy module provides a quick and easy interface to a huge number of different DFT and FFT-related functions. In the background, SciPy uses the algorithms provided by FFTPACK to calculate highly efficient FFTs.

### **scipy.fftpack versus numpy.fftpack**

A lot of the functions available in the SciPy FFTPACK submodule are *also* available in the NumPy FFTPACK submodule — so why bother using the SciPy version? While a lot of the functions are identical, the SciPy versions are typically more optimised than the NumPy versions.

► SciPy is a good choice of library for accessing the FFT in Python, due to how ubiquitous it is. However, an alternative is FFTW, via their Python module interface [pyFFTW](#).

### 8.7.1 The FFTPACK Submodule

To use the methods available in the FFTPACK submodule, you need to import the SciPy FFTPACK submodule under a separate name:

```
>>> from scipy import fftpack
>>> fftpack.fft(np.arange(4))
array([ 6.+0.j, -2.+2.j, -2.+0.j, -2.-2.j])
```

► In this chapter, we will only be covering a small subset of what SciPy is capable of. For more information and details, see the [scipy.fftpack documentation](#).

### 8.7.2 One-Dimensional FFT

SciPy makes it exceptionally easy to perform a Fast Fourier Transform; simply call the function

```
>>> F = fftpack.fft(f, n=None, axis=-1, overwrite_x=False)
```

This calculates the one-sided discrete Fourier transform of input array `f`, storing the result in array variable `F`. Note that all options after `f` are **optional**:

- `n`: the size of the DFT to be performed. If not provided, it is assumed that `n` is simply the length of `f`; `n = f.shape[axis]`. If `n < f.shape[axis]` then the DFT on `f[:n]` is performed. If `n > f.shape[axis]`, then `f` is padded with zeros to increase its length.
- `axis`: the axis or dimension over which the FFT is performed. If not provided, by default the FFT is performed on the last dimension (`axis = -1`).
- `overwrite_x`: If `True`, the contents of the input array `f` can be overwritten as the FFT is calculated — this may allow for an increase in FFT efficiency.

To perform the inverse FFT transform, the corresponding function is

► Unlike FFTW, SciPy FFTPACK *does* properly normalise the inverse FFT.

```
>>> f = fftpack.ifft(F, n=None, axis=-1, overwrite_x=False)
```

which uses the same optional arguments as `fftpack.fft`.

### Example 8.12 Fast Fourier transform in SciPy (Python)

Use SciPy FFTPACK to calculate the Fast Fourier Transform of the discrete function  $f_n = \sin(4n^2) \tan(n + 1)$  for  $n = 0, 1, \dots, 255$ .

- ▶ As FFTPACK uses the Cooley–Tukey algorithm under the hood, it is most efficient when `f` has a length that is a power of two, and least efficient when the length of `f` cannot be factored, for example when `f.shape[axis]` is a prime number.

#### Solution:

```
import numpy as np
from scipy import fftpack

# create the discrete function
n = np.arange(256)
f = np.sin(4*n**2) * np.tan(n+1)

# perform the one-sided FFT
F = fftpack.fft(f)
```

SciPy FFTPACK also supports the two-dimensional FFT and inverse FFT, via the functions

```
>>> F = fftpack.fft2(f, shape=None, axes=(-2, -1), overwrite_x=False)
>>> f = fftpack.ifft2(F, shape=None, axes=(-2, -1), overwrite_x=False)
```

The optional arguments here are similar to the one-dimensional case, however rather than the length of the FFT you can now, if you would like to, specify the  $n \times n$  `shape` to undergo the 2D FFT. Similarly, you can now specify the two axes/dimensions over which to perform the FFT, using the `axes` argument.

SciPy also allows you to extend this to an arbitrary  $N$ -dimensional FFT and inverse FFT, with the following functions:

```
>>> F = fftpack.fftn(f, shape=None, axes=None, overwrite_x=False)
>>> f = fftpack.ifftn(F, shape=None, axes=None, overwrite_x=False)
```

For this case, if `f` is an  $N$ -dimensional array, then, by default, the  $N$ -dimensional FFT is performed. Alternatively, specifying the `axes` allows you to perform a FFT over only a subset of dimensions.

### 8.7.3 Fourier Differentiation

In addition to the one-sided FFT algorithms provided by SciPy, SciPy FFTPACK also provides a built-in function for performing Fourier differentiation using the FFT:

```
>>> df = fftpack.diff(f, order=1, period=None)
```

where  $f$  is the input data to be differentiated, and  $df$  is the resulting derivative. There are also two optional arguments:

- **order**: by specifying `order=n`, this allows you to calculate the  $n$ th order Fourier derivative.
- **period**: The period of the input data (assumed to be  $2\pi$  by default).

The ability to provide the period is especially useful — recall that when we explored Fourier differentiation, the result was inextricably linked to the choice of grid spacing  $\Delta x$  used to discretise the function  $f_n = f(x_n)$ . However, as we have seen, FFTPACK and the DFT in general assume the grid spacing/sampling rate of all input data is such that  $\Delta x = 1$ , requiring us to multiply by the correct phase to ‘re-introduce’ this information (such as when approximating the Fourier transform via the DFT).

However, by specifying the **period** of the input data, the `diff` function takes into account our grid spacing when calculating the Fourier derivative. So, what is the period of our input function? Since we are approximating an arbitrary, non-periodic continuous function, the period is the maximum size of our discretised grid:

$$\lambda = N\Delta x. \quad (8.52)$$

### Example 8.13 Fourier differentiation (Python)

Use SciPy’s built in Fourier differentiation to differentiate the wavefunction  $\psi(x) = e^{-x^2/2} \sin(4x)$  over the domain  $-5 \leq x \leq 5$  with  $N = 201$  discretised points.

#### Solution:

```
import numpy as np
from scipy import fftpack

# define the properties
N = 201
dx = (5-(-5))/(N-1)

# define the x-grid and discretised wavefunction
x = np.arange(-5, 5+dx, dx)
psi = np.exp(-(x**2)/2)*np.sin(4*x)

# perform the Fourier differentiation
dpsi = fftpack.diff(psi, period=N*dx)
```

## Exercises

**P8.1** The Hamiltonian for the quantum harmonic oscillator is given by

$$\hat{H} = \frac{\hat{p}^2}{2m} + \frac{1}{2}m\omega^2\hat{x}^2$$

Substituting this into the time-independent Schrödinger equation  $\hat{H}\psi(x) = E\psi(x)$  and solving for the first excited state, we find that this corresponds to the wavefunction

$$\psi_1(x) = \left(\frac{4\alpha^3}{\pi}\right)^{1/4} e^{-\alpha x^2/2}$$

where  $\alpha = m\omega/\hbar$ .

- (a) Perform the Fourier transform  $\phi_1(k) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \psi_1(x) e^{-ikx} dx$  to find the wavefunction representation in momentum-space.
- (b) Using the Riemann sum technique to approximate the Fourier transform, let  $\alpha = 1$  and discretise  $\psi(x)$  over a reasonable grid size and use Fortran or Python to numerically calculate the momentum-space wavefunction  $\phi(k)$ . Remember to choose your domain  $-x_{max} < x < x_{max}$  such that  $\psi(\pm x_{max}) \approx 0$ .
- (c) How does your result compare against the exact result? Produce an error plot comparing the exact and numerical results over the domain.
- (d) Repeat part (b) for varying grid discretisations  $\Delta x$ , and plot the resulting *maximum* absolute error vs  $\Delta x$ . What do you find? How does the error in this approximation scale with  $\Delta x$ ?

**P8.2** Show that the complex exponential  $e^{2\pi im(\ell-n)}$  is *orthonormal*; that is,

$$\frac{1}{N} \sum_{m=0}^{N-1} e^{2\pi im(\ell-n)/N} = \delta_{\ell n} = \begin{cases} 0, & \ell \neq n \\ 1, & \ell = n \end{cases}.$$

*Hint: consider the case  $\ell = n$  and  $\ell \neq n$  separately. For the case  $\ell \neq n$ , it helps to rewrite the sum using the formula for a geometric series,*

$$\sum_{m=0}^{N-1} r^m = \frac{r^N - 1}{r - 1}$$

- P8.3**
- (a) Using Fourier differentiation (Eq. 8.19) and a FFT implementation, calculate the first derivative of  $\psi_1(x)$  from problem 8.1 letting  $\alpha = 1$  and using  $N = 301$  discretised grid points.
  - (b) Modify Eq. 8.19 to instead calculate the *second* derivative (hint: apply  $\frac{\partial}{\partial x^2}$  to the discretised Fourier transform expression). Use this expression to calculate the second derivative of  $\psi_1(x)$ .
  - (c) Compare your results to the exact solution to the first derivative  $\psi'_1(x)$  and the second derivative  $\psi''_1(x)$ . What do you find? Repeat your results for various values of  $N$ , and plot the maximum error vs  $N$ .
  - (d) Extend your analysis to include the method of finite differences. What can you say about the accuracy of Fourier differentiation vs the finite-difference method?

- P8.4** Repeat 8.3 for the 6th excited state of the quantum harmonic oscillator:

$$\psi_6(x) = \frac{1}{12} \sqrt[4]{\frac{\alpha}{25\pi}} (8\alpha^3 x^6 - 60\alpha^2 x^4 + 90\alpha x^2 - 15) e^{\frac{1}{2}(-\alpha)x^2}$$

As before, let  $\alpha = 1$ . Plot your result, and compare it to the analytical solution. What can you say about Fourier differentiation as the function to be differentiated becomes increasingly more oscillatory?

- P8.5** Consider the function  $f(x) = e^{\cos(x)}$ ,  $0 \leq x \leq 4\pi$ .

- (a) Write your own function in either Fortran or NumPy to find the discrete Fourier transform of  $f(x)$ .
- (b) Use a FFT implementation (either via SciPy or FFTW) to find the discrete Fourier transform of this function.
- (c) Using several different grid points  $N$  in your discretised function  $f_n$ , compute the time required to calculate the DFT directly compared to the FFT. At what values of  $N$  is the FFT faster than the DFT? At what values does it appear slower?

### P8.6 The multidimensional DFT

The DFT can easily be extended to multidimensional discrete functions  $f_{n_1, n_2, \dots, n_D}$ ; the DFT is now applied sequentially to each dimension  $n_i$ :

$$\tilde{F}_{m_1, m_2, \dots, m_D} = \sum_{n_1=0}^{N-1} \cdots \sum_{n_D=0}^{N-1} e^{-2\pi i n_1 k_1 / N} \cdots e^{-2\pi i n_D k_D / N} f_{n_1, n_2, \dots, n_D}$$

where we have assumed that each dimension is discretised using the same number of points  $N$ . Letting  $\mathbf{n} = (n_1, n_2, \dots, n_D)$  and  $\mathbf{m} = (m_1, m_2, \dots, m_D)$ , we can write this in vector notation:

$$\tilde{F}_{\mathbf{m}} = \sum_{\mathbf{n}=0}^{N-1} e^{-2\pi i \mathbf{n} \cdot \mathbf{m} / N} f_{\mathbf{n}}$$

The two-dimensional quantum harmonic oscillator consists of a superposition of two one-dimensional harmonic oscillators — one in the  $x$  direction, and one in the  $y$  direction. For example, the 2D harmonic oscillator in the first excited state in the  $x$  direction and the ground state in the  $y$  direction is given by

$$\psi_{10}(x, y) = \psi_1(x)\psi_0(y) = \sqrt{\frac{2}{\pi}} \alpha x e^{-\alpha(x^2+y^2)/2}$$

- (a) Use Riemann summation to approximate the two-dimensional Fourier transform

$$\phi_{10}(k_x, k_y) = \frac{1}{2\pi} \iint_{-\infty}^{\infty} \psi_{10}(x, y) e^{-ik_x x} e^{-ik_y y} dx dy$$

- (b) Using the definition above of the multi-dimensional discrete Fourier transform, approximate the Fourier transform using the 2D DFT. Note that you'll need to *center* the one-sided DFT ( $\mathbf{n} \rightarrow \mathbf{n} - a$  where  $a = (N-1)/2$ ) to match the centered Fourier transform from part (a). Compute the numerical result in Python or Fortran, using an appropriate grid discretisation.
- (c) Using SciPy's `scipy.fftpack.fft2` or FFTW's `fftw_plan_dft_2d`, approximate the two-dimensional centered Fourier transform — be sure to take into account the required phase shifts. Compare your results to part (b).

## Part III:

# Solving the Schrödinger Equation



# Chapter 9

## One Dimension

The Schrödinger equation, a linear partial differential equation which describes the time evolution of quantum states, is the corner stone of quantum mechanics. Using the Schrödinger equation as the starting point, we are able to obtain information about the bound states, free states, energy levels, and time-evolution of a quantum system. However, whilst analytical solutions to the Schrödinger equation can be found for a few systems (for example, the quantum harmonic oscillator), in general it can only be solved numerically.

In this chapter, we'll ease slowly into the complexities of quantum dynamical systems, by looking at solving the time-independent Schrödinger equation in one dimension.

### 9.1 The Schrödinger Equation

In its most general form, the Schrödinger equation is

$$i\hbar \frac{\partial}{\partial t} \psi(x, t) = \hat{H} \psi(x, t), \quad (9.1)$$

where  $\hbar$  is Planck's constant,  $\psi(x, t)$  is the **wavefunction** of the quantum system, and  $\hat{H}$  is the **Hamiltonian**.

The Hamiltonian operator represents the total energy of the system, which is the sum of kinetic energy,  $\hat{T} = \hat{p}/2m^2$ , and potential energy  $\hat{V} = V(x, t)$ :

$$\hat{H} = \hat{T} + \hat{V} = \frac{\hat{p}^2}{2m} + V(x, t). \quad (9.2)$$

In position space, the momentum operator is simply  $\hat{p} = -i\hbar \frac{\partial}{\partial x}$ . Putting this all together, we get the 1D Schrödinger equation:

$$i\hbar \frac{\partial}{\partial t} \psi(x, t) = \left( -\frac{\hbar^2}{2m} \frac{\partial^2}{\partial x^2} + V(x, t) \right) \psi(x, t) \quad (9.3)$$

► Taking the absolute value of the wavefunction,  $|\psi(x, t)|^2$ , gives the **probability** of finding the quantum particle at  $x$  at time  $t$ .

► The Schrödinger equation is a **linear** partial differential equation.

► Letting  $\langle x | \psi(t) \rangle = \psi(x, t)$ , the Schrödinger equation can be written in Dirac's bra-ket notation:

$$i\hbar \frac{\partial}{\partial t} |\psi(t)\rangle = \hat{H} |\psi(t)\rangle$$

## 9.2 The Time-Independent Schrödinger Equation

Let's assume that the potential (and thus Hamiltonian) in our quantum system is time-independent; as a result, the solution to the Schrödinger Equation 9.3 is *separable* in time and space,

$$\psi(x, t) = \varphi(x)f(t). \quad (9.4)$$

Substituting this back into the Schrödinger equation gives

$$i\hbar\varphi(x)\frac{df}{dt} = -\frac{\hbar^2}{2m}\frac{d^2\varphi}{dx^2}f(t) + V(x)\varphi(x)f(t), \quad (9.5)$$

and then using separation of variables leads to

$$i\hbar\frac{1}{f(t)}\frac{df}{dt} = \left(-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x)\right)\varphi(x) = E. \quad (9.6)$$

As the left-hand side depends only on  $t$ , and the right-hand side depends only on  $x$ , the fact that they are equal means that **they both must have a constant value**. Here, we have called this constant  $E$ .

We can use this fact to solve the differential equation for  $f(t)$ :

$$\frac{df}{dt} = -\frac{iE}{\hbar}f(t) \Rightarrow f(t) \propto e^{-iEt/\hbar}. \quad (9.7)$$

Therefore, the general solution is given by

$$\psi(x, t) = \varphi(x)e^{-iEt/\hbar}. \quad (9.8)$$

where  $\varphi$  satisfies what we call the **time-independent Schrödinger equation**:

$\hat{H}\varphi(x) = E\varphi(x).$

(9.9)

The solutions  $\varphi_N(x)$  to this equation provide the **bound states** or **stationary states** of the system, as well as their respective energy levels  $E_N$ . Note that this is also an **eigenvalue equation**. Since the Hamiltonian  $\hat{H}$  is by definition an Hermitian operator, from linear algebra, we know that:

- The eigenvalues  $E_N$  are always **real**
- The eigenvectors  $\varphi_N(x)$  are **linearly independent** and **orthogonal** with respect to the Hilbert space inner product,

$$\langle \varphi_i(x) | \varphi_j(x) \rangle = \int_{-\infty}^{\infty} \varphi_i(x)^* \varphi_j(x) dx = \delta_{ij},$$

i.e. the eigenvectors form a **complete set** for the Hilbert space.

- Letting  $\langle x | \varphi \rangle = \varphi(x)$ , the time-independent Schrödinger equation can be written in bra-ket notation:

$$\hat{H}|\varphi\rangle = E|\varphi\rangle.$$

### 9.3 Boundary Value Problems

In Chap. 5 when we first looked at solving differential equations using numerical methods, we were concerned with initial value problems (IVPs) – how a system evolves assuming we know its value and derivative(s) at  $x = x_0$ :

$$y''(x) = f(y(x), y'(x), x), \quad y(x_0) = a, \quad y'(x_0) = \alpha. \quad (9.10)$$

In most cases, an initial value problem has a unique solution for a particular set of initial conditions.

However, the time-independent Schrödinger equation takes the form of a partial differential wave equation – instead of dealing with an initial-value problem, we have a **boundary value problem**:

$$y''(x) = f(y(x), y'(x), x), \quad y(x_0) = a, \quad y(x_N) = b. \quad (9.11)$$

Unlike the initial value problem, the boundary value problem (BVP) doesn't have unique solutions! There will be numerous solutions, each corresponding to a different eigenvalue  $\lambda_i$  of the differential equation. This is why we get multiple stationary wavefunctions and energy levels when solving the time-independent Schrödinger equation. These stationary wavefunctions forms a complete set of mutually orthogonal bases, a linear combination of which can be used to represent an arbitrary wavefunctions using appropriate weighting coefficients.

#### 9.3.1 Shooting Method

One approach to dealing with boundary value problems is to try and convert them to initial value problems, and then solve them like we did previously. Consider the boundary value problem

$$y''(x) = f(y(x), y'(x), x), \quad y(x_0) = a, \quad y(x_N) = b. \quad (9.12)$$

Let  $u_\alpha(x)$  be a solution to the equivalent initial value problem:

$$u''_\alpha(x) = f(u_\alpha(x), u'_\alpha(x), x), \quad u_\alpha(x_0) = a, \quad u'_\alpha(x_0) = \alpha. \quad (9.13)$$

Consider the difference between the two solutions at the end point of the boundary,  $x = x_N$ :

$$g(\alpha) = u_\alpha(x_N) - y(x_N) = u_\alpha(x_N) - b. \quad (9.14)$$

Therefore, in order to solve numerically for  $y(x)$ , we need to solve numerically for  $u_\alpha(x)$  satisfying the following conditions:

$$u''_\alpha(x) = f(u_\alpha(x), u'_\alpha(x), x), \quad u_\alpha(x_0) = a, \quad u'_\alpha(x_0) = \alpha,$$

$$g(\alpha) = u_\alpha(x_N) - b = 0.$$

(9.15)

► *Why is it called the shooting method?*

Because we treat the boundary value problem as an IVP with unknown initial gradient, and are solving or ‘shooting’ to find the value at the other boundary.

If we miss our mark, we then adjust the initial gradient and shoot again.

This is slightly more complicated than just solving an initial value problem! It involves guessing  $\alpha$ , solving the differential equation, and then using a **root-finding** algorithm to determine a better estimate for  $\alpha$ . Then repeat the process, until your estimate of  $\alpha$  starts to converge ( $|\alpha_{j+1} - \alpha_j| \leq \epsilon$ ).

**Example 9.1 Bisection shooting method**

Solve the following differential equation using the bisection shooting method

$$y'' + 5y' = 5x, \quad y(0) = 1, \quad y(1) = 0.$$

**Solution:** To solve this via the shooting method, we consider the initial value problem

$$u''_\alpha + 5u'_\alpha = 5x, \quad u_\alpha(0) = 1, \quad u'_\alpha(0) = \alpha.$$

This can be solved numerically using the leap-frog or Runge-Kutta method.

For our first two estimates of  $\alpha$ , let's pick  $\alpha_1 = 1$  and  $\alpha_2 = 5$ . Solving the initial value problem and looking at the right hand boundary, we have  $u_{\alpha_1}(1) < 0$  and  $u_{\alpha_2}(1) > 0$ . Thus, the correct value of  $\alpha$  must lie somewhere in-between these two starting guesses.

So for the third estimate of  $\alpha$ , let's bisect this interval:

$$\alpha_3 = \frac{1}{2}(1 + 5) = 3.$$

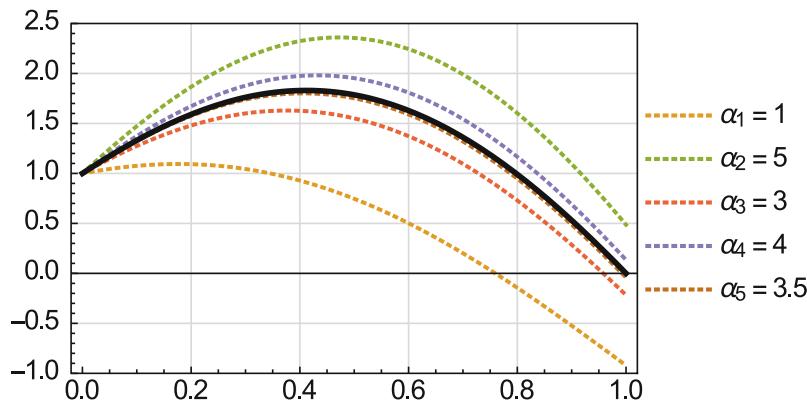
Now, we have  $u_{\alpha_3}(1) < 0$  – so the next estimate must lie in the interval  $\alpha \in (3, 4)$ . Bisecting this interval,

$$\alpha_4 = \frac{1}{2}(3 + 4) = 3.5.$$

See Fig. 9.1 for a graphical interpretation of the bisecting process. We continue bisecting until  $\alpha$  converges to a value. In this case,

$$\alpha = 3.59655,$$

and thus  $u_{3.59655}(x)$  is the solution to the boundary value problem.



**Figure 9.1** The solution  $y(x)$  to the boundary value problem (black) and the solutions  $u_\alpha(x)$  to the initial value problem (dashed) for various values of  $\alpha$

### Newton-Raphson Shooting Method

- The Newton-Raphson shooting method is a **second order** method.

One such root-finding algorithm we've covered is the Newton-Raphson method (Sect. 4.4). Let's use it to determine a better estimate of  $\alpha$ , based on a current estimate  $\alpha_j$ :

$$\alpha_{j+1} = \alpha_j - \frac{g(\alpha_j)}{g'(\alpha_j)} = \alpha_j - \frac{u_{\alpha_j}(x_N) - b}{du_{\alpha_j}(x_N)/d\alpha_j}. \quad (9.16)$$

To calculate  $du_\alpha/d\alpha$ , we need to take the derivative with respect to  $\alpha$  of Eq. 9.13, the ODE which defines  $u_\alpha(x)$ :

$$\frac{d}{d\alpha} u''_\alpha(x) = \frac{d}{d\alpha} f(u_\alpha(x), u'_\alpha(x), x) = \frac{\partial f}{\partial u_\alpha} \frac{du_\alpha}{d\alpha} + \frac{\partial f}{\partial u'_\alpha} \frac{du'_\alpha}{d\alpha}.$$

If we let  $z_\alpha(x) = \frac{d}{d\alpha} u_\alpha(x)$ , then we have the new initial value problem

$$z''_\alpha(x) = \frac{\partial f}{\partial u_\alpha} z_\alpha(x) + \frac{\partial f}{\partial u'_\alpha} z'_\alpha(x), \quad (9.17)$$

with initial conditions

$$z_\alpha(x_0) = \frac{d}{d\alpha} u_\alpha(x_0) = \frac{d}{d\alpha} a = 0 \quad (9.18)$$

$$z'_\alpha(x_0) = \frac{d}{d\alpha} u'_\alpha(x_0) = \frac{d}{d\alpha} \alpha = 1. \quad (9.19)$$

Therefore, the Newton-Raphson iteration becomes

$$\alpha_{j+1} = \alpha_j - \frac{u_{\alpha_j}(x_N) - b}{z_{\alpha_j}(x_N)}. \quad (9.20)$$

**In summary:** To use the Newton-Raphson shooting method to solve the boundary value problem

$$y''(x) = f(y(x), y'(x), x), \quad y(x_0) = a, \quad y(x_N) = b,$$

- (1) Estimate  $\alpha = \alpha_j$
- (2) Solve the following initial value problems (e.g. using Runge-Kutta):
  - (i)  $u''_\alpha(x) = f(u_\alpha(x), u'_\alpha(x), x), \quad u_\alpha(x_0) = a, \quad u'_\alpha(x_0) = \alpha$
  - (ii)  $z''_\alpha(x) = \frac{\partial f}{\partial u_\alpha} z_\alpha(x) + \frac{\partial f}{\partial u'_\alpha} z'_\alpha(x), \quad z_\alpha(x_0) = 0, \quad z'_\alpha(x_0) = 1$
- (3) Use the Newton-Raphson method to better approximate  $\alpha$ :

$$\alpha_{j+1} = \alpha_j - \frac{u_{\alpha_j}(x_N) - b}{z_{\alpha_j}(x_N)}$$

- (4) If  $\alpha$  has converged ( $|\alpha_{j+1} - \alpha_j| \leq \epsilon$ ) then the solution to the boundary value problem is  $y(x) = u_{\alpha_{j+1}}(x)$ . If not,  $\alpha = \alpha_{j+1}$  and return to (2).

Both of these are **first order** methods, however they do not require  $g'(\alpha)$  to be computed, resulting in only one initial value problem to be solved.

As such, they are less computationally intensive than the Newton-Raphson shooting method.

- By replacing  $g'(\alpha_j)$  in the Newton-Raphson shooting method by the backwards finite difference formula, we get the **secant shooting method**:

$$\alpha_{j+1} = \alpha_j - g(\alpha_j) \frac{\alpha_j - \alpha_{j-1}}{g(\alpha_j) - g(\alpha_{j-1})}$$

### 9.3.2 Linear Shooting Method

Linear boundary value problems are such that  $f(y(x), y'(x), x)$  is a *linear* function – that is, we are able to write the differential equation in the following form:

$$y''(x) + P(x)y'(x) + Q(x)y(x) = R(x) \quad (9.21)$$

and they satisfy linearity – if  $u(x)$  and  $v(x)$  are solutions to the ODE, then so is  $w(x) = c_1u(x) + c_2v(x)$ .

Note that the time-independent Schrödinger equation is a linear partial differential equation; writing it in this form it becomes

$$\psi''(x) + \frac{2m}{\hbar^2} [E - V(x)] \psi(x) = 0. \quad (9.22)$$

So, can we use the linearity of these systems to improve on the previous section, where the conversion of a BVP to a IVP resulted in having to guess the initial value  $\alpha$ ? Turns out we can! Walk with me here.

Consider the linear boundary value problem

$$y''(x) + P(x)y'(x) + Q(x)y(x) = R(x), \quad y(x_0) = a, \quad y(x_N) = b, \quad (9.23)$$

as well as the following two initial value problems,

$$\begin{aligned} u''(x) + P(x)u'(x) + Q(x)u(x) &= R(x), & u(x_0) &= a, & u'(x_0) &= 0 \\ v''(x) + P(x)v'(x) + Q(x)v(x) &= 0, & v(x_0) &= 0, & v'(x_0) &= s \end{aligned}$$

where the ODE satisfied by  $v(x)$  is **homogeneous** ( $R(x) = 0$ ), and  $s$  is an arbitrary constant known as the **free shooting parameter**.

Now, let  $w(x) = u(x) + \theta v(x)$ . Due to linearity,  $w(x)$  satisfies both initial value problems, with

$$w(x_0) = u(x_0) + \theta v(x_0) = a \quad (9.24)$$

$$w'(x_0) = u'(x_0) + \theta v'(x_0) = \theta s. \quad (9.25)$$

Furthermore, if  $w(x_N) = u(x_N) + \theta v(x_N) = b$ , then  $w(x)$  also satisfies the boundary value problem! Let's solve for the value of  $\theta$  where this boundary condition is met:

$$\theta = \frac{b - u(x_N)}{v(x_N)}. \quad (9.26)$$

Therefore, the solution to the linear boundary value problem is

$$y(x) = u(x) + \frac{b - u(x_N)}{v(x_N)} v(x). \quad (9.27)$$

This already looks like a better algorithm – simply solve two initial value problems (rather than one) with specified (rather than free) initial conditions,

and a linear superposition provides the solution to the boundary value problem. No more having to iterate the initial conditions until convergence!

However, there is a small caveat (have you spotted it?). In order for the correct value of  $\theta$  to be found, we must have  $v(x_N) \neq 0$ . From Sturm-Liouville theory, this is only guaranteed when  $Q(x) \leq 0$  for  $x \in [x_0, x_N]$ , or in the case of the time-independent Schrödinger equation, when  $E \leq V(x)$ . If this condition is not met, than it is possible (but not *certain*) that no solution to the boundary value problem exists.

**In summary:** To use the linear shooting method to solve the boundary value problem

$$y''(x) + P(x)y'(x) + Q(x)y(x) = R(x), \quad y(x_0) = a, \quad y(x_N) = b,$$

- (1) Solve the initial value problems

$$\begin{aligned} u''(x) + P(x)u'(x) + Q(x)u(x) &= R(x), & u(x_0) &= a, & u'(x_0) &= 0 \\ v''(x) + P(x)v'(x) + Q(x)v(x) &= 0, & v(x_0) &= 0, & v'(x_0) &= s \end{aligned}$$

for any value of  $s \neq 0$ , using a numerical method (Euler/leap-frog/Runge-Kutta etc) of your choice.

- (2) If  $v(x_N) \neq 0$ , then the solution to the boundary value problem is

$$y(x) = u(x) + \frac{b - u(x_N)}{v(x_N)}v(x).$$

Otherwise, no solutions to the boundary value problem may exist.

### 9.3.3 Linear Shooting Method with Homogeneous BVPs

What happens if a system has a linear *and* homogeneous ODE, and left boundary value of zero? For example, consider the boundary value problem

$$y''(x) + P(x)y'(x) + Q(x)y(x) = 0, \quad y(x_0) = 0, \quad y(x_N) = b. \quad (9.28)$$

This problem is both homogeneous ( $R(x) = 0$ ), linear, and the left boundary has value  $y(x_0) = 0$ . Let's proceed as we did above, and see what happens.

Define the two initial value problems

$$\begin{aligned} u''(x) + P(x)u'(x) + Q(x)u(x) &= 0, & u(x_0) &= 0, \quad u'(x_0) = 0, \\ v''(x) + P(x)v'(x) + Q(x)v(x) &= 0, & v(x_0) &= 0, \quad v'(x_0) = s, \end{aligned}$$

where  $s \neq 0$  is the free shooting parameter.

Unlike previously, the differential equation for  $u(x)$  now is *also* homogeneous, with homogeneous initial conditions. Since the trivial solution  $u(x) = 0$  satisfies both the ODE and the initial conditions, from the uniqueness theorem we can deduce that  $u(x) = 0$  is the *only solution*.

Therefore, the solution to the boundary value problem becomes

$$y(x) = u(x) + \frac{b - u(x_N)}{v(x_N)}v(x) = \frac{b}{v(x_N)}v(x). \quad (9.29)$$

The solution  $y(x)$  is simply a constant rescaling of  $v(x)$ , the equivalent initial value problem! And not only that, it doesn't even matter what we choose  $v'(x_0) = s$  to be, as the free shooting parameter  $s$  doesn't feature in Eq. 9.29. How nice.

**In summary:** To use the linear shooting method to solve the *homogeneous* boundary value problem

$$y''(x) + P(x)y'(x) + Q(x)y(x) = 0, \quad y(x_0) = 0, \quad y(x_N) = b,$$

(1) Solve the initial value problem

$$v''(x) + P(x)v'(x) + Q(x)v(x) = 0, \quad v(x_0) = 0, \quad v'(x_0) = s$$

for any value of  $s \neq 0$ , using a numerical method (Euler/leap-frog/Runge-Kutta etc) of your choice.

(2) If  $v(x_N) \neq 0$ , then the solution to the boundary value problem is

$$y(x) = \frac{b}{v(x_N)}v(x).$$

Otherwise, no solutions to the boundary value problem may exist.

► **Why must the wavefunction decay when  $V > E$ ?**

Write the Schrödinger equation in the form

$$\psi''(x) + \lambda^2 \psi(x) = 0$$

$$\lambda^2 \sim [E - V(x)]$$

for  $x \rightarrow \pm\infty$ . In this region,

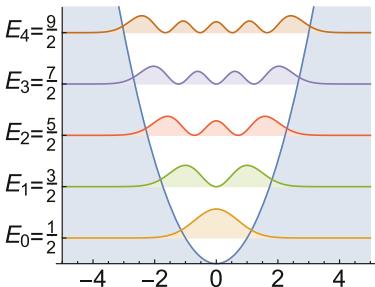
$$V(x) > E \Rightarrow \lambda \leq 0$$

and so the solution is

$$\psi(x) \sim Ae^{-\lambda x} + Be^{\lambda x}.$$

The Hilbert space requires  $\psi(x)$  be **square integrable** – it cannot ‘blow up’ to infinity. So, as  $x \rightarrow \pm\infty$ ,  $V(x) > E$ , and

$$\psi(x) \sim e^{-\lambda|x|} \rightarrow 0$$



**Figure 9.2**

► Always take into account the symmetry/behaviour of the potential  $V(x)$  to ensure your finite boundary conditions are good approximations!

For example, Fig. 9.2 shows a quantum harmonic potential with the first 5 states. To solve for these states  $E \leq 9/2$ , a good choice of domain is  $-5 \leq x \leq 5$ , with  $\psi(-5) \approx 0$  and  $\psi(5) \approx 0$ .

## 9.4 Shooting method for the Schrödinger Equation

Let’s now go back to the time-independent Schrödinger equation, and try and apply the shooting method to find solutions. Recall that the time-independent Schrödinger equation is

$$-\frac{\hbar^2}{2m}\psi''(x) + V(x)\psi(x) = E\psi(x). \quad (9.30)$$

In cases where  $V(x) \rightarrow \infty$  as  $x \rightarrow \pm\infty$ , a reasonable boundary condition to impose is

$$\psi(x) \rightarrow 0 \text{ as } x \rightarrow \pm\infty. \quad (9.31)$$

Although, you might argue that boundary conditions at infinite  $x$  values aren’t *really* that reasonable when we are trying to solve the system numerically. Okay, we hear you. Let’s try and approximate some finite boundary conditions.

Assuming  $V(x)$  is centered on the origin, let’s work on the finite domain  $-a \leq x \leq a$ , with  $a$  chosen such that  $V(\pm a)$  is sufficiently large to ensure that  $\psi(\pm a) \approx 0$ . In this case, we have the finite boundary conditions

$$\psi(-a) \approx 0, \quad \psi(a) \approx 0. \quad (9.32)$$

Since the boundary value problem is linear, homogeneous, and the left boundary value is 0, we know from Sect. 9.3.3 that (up to a constant rescaling factor) this is equivalent to solving the initial value problem

$$-\frac{\hbar^2}{2m}\psi''(x) + V(x)\psi(x) = E\psi(x), \quad \psi(-a) = 0, \quad \psi'(-a) = s, \quad (9.33)$$

where  $s$  is a free shooting parameter and can take any non-zero real value.

### The Discretised Time-Independent Schrödinger Equation

Let’s discretise the  $x$  grid between  $x \in [x_0, x_{N-1}]$  using  $N$  points,

$$x_{j+1} = x_j + \Delta x \text{ where } \Delta x = \frac{x_{N-1} - x_0}{N-1}. \quad (9.34)$$

Using the **second derivative central difference formula** (Sect. 5.1.4) method to approximate the double derivative,

$$f''(x) = \frac{f(x + \Delta x) - 2f(x) + f(x - \Delta x)}{(\Delta x)^2} + O((\Delta x)^2), \quad (9.35)$$

and denoting  $\psi_j = \psi(x_j)$  and  $V_j = V(x_j)$ , we get at a discretised time-dependent Schrödinger equation, accurate to second order:

$$-\frac{\hbar^2}{2m} \frac{\psi_{j+1} - 2\psi_j + \psi_{j-1}}{\Delta x^2} + V_j\psi_j + \mathcal{O}((\Delta x)^2) = E\psi_j. \quad (9.36)$$

Rearranging this into a more iterative form,

$$\psi_{j+1} = 2 \left[ \frac{m(\Delta x)^2}{\hbar^2} (V_j - E) + 1 \right] \psi_j - \psi_{j-1}, \quad j = 1, 2, \dots, N-2 \quad (9.37)$$

### Discretising the Initial Conditions

Let's set the endpoints of our discretised grid to  $x_0 = -a$  and  $x_{N-1} = a$ , ensuring the boundary conditions  $\psi_0 \approx 0$ , and  $\psi_{N-1} \approx 0$ . From the forward Euler approximation to the derivative,

$$\psi'(x_0) \approx \frac{\psi(x_0 + \Delta x) - \psi(x_0)}{\Delta x} = \frac{\psi_1 - \psi_0}{\Delta x}, \quad (9.38)$$

thus the initial condition  $\psi'(x_0)$  provides us with the value of  $\psi_1$ ,

$$\psi_1 = \psi'(x_0)\Delta x + \psi_0. \quad (9.39)$$

Therefore, we can convert Eq. 9.33 into a discretised *initial value problem*,

$$\boxed{\begin{aligned} \psi_{j+1} &= 2 \left[ \frac{m(\Delta x)^2}{\hbar^2} (V_j - E) + 1 \right] \psi_j - \psi_{j-1}, \quad j = 1, 2, \dots \\ \psi_0 &= 0, \quad \psi_1 = s \end{aligned}} \quad (9.40)$$

where  $s \neq 0$  is the free shooting parameter.

Assuming we know the energy  $E$  of the state we wish to find, we now have all we need to implement the shooting method! All that's left to do is to rescale the solution of the initial value problem so that the right hand boundary condition,  $\psi_{N-1} \approx 0$ , is met.

'But wait!' we hear you cry out. 'Isn't there already a scaling condition on wavefunctions? Uh, hello, **normalisation**?'

That's correct, of course. And it turns out the normalisation rescaling is all we need to ensure the initial value solution satisfies (a) the time-independent Schrödinger equation, (b) the boundary conditions, and (c) the normalisation condition.

Hooray!

### Normalisation

Since the wavefunction  $\psi(x)$  represents a probability amplitude, we require it to be normalised:

$$\langle \psi | \psi \rangle = \int_{-\infty}^{\infty} |\psi(x)|^2 dx = 1. \quad (9.41)$$

- A good choice for the numerical integration is Simpson's rule, providing a good balance between accuracy and efficiency.

As we are working with a discretised wavefunction  $\psi_j$ , to perform this integration we must use one of the Newton-Cotes numerical integration methods (see Chap. 6.)

Of course, it is very unlikely that your solution to the IVP will be normalised; instead you'll get

$$\int_{-\infty}^{\infty} |\psi(x)|^2 dx = A. \quad (9.42)$$

Thus, to normalise the wavefunction, it needs to be multiplied by the constant factor  $1/\sqrt{A}$ .

### 9.4.1 The Eigenvalue Problem

So far, we've assumed that we know the energy or eigenvalue  $E$  of the state for which we wish to solve the time independent Schrödinger equation. But unless you're getting inside information from experimentalists, this is not usually the case. To work out the allowed energy values  $E$ , you have to be able to solve the Schrödinger equation in the first place!

Of course, if you're feeling lucky, you could always try and guess  $E$ . But you would have to be feeling *exceptionally* lucky – the possible outcomes are

- (1) You guess the correct eigenvalue  $E$  for the eigenstate you wish to find,
- (2) You guess a correct eigenvalue  $E$ , but it provides the solution for a different state,
- (3) You guess a value of  $E$  that isn't an eigenvalue – no solution satisfying the boundary conditions can be found.

Unless the discrete energy levels turn out to be *incredibly* dense, outcome (3) is probably the likeliest.

So how can we solve for both the eigenstates *and* the eigenvalues? One way is to treat  $E$  as a parameter in the initial value problem:

$$\begin{aligned} \psi_{j+1}^{(E)} &= 2 \left[ \frac{m(\Delta x)^2}{\hbar^2} (V_j - E) + 1 \right] \psi_j^{(E)} - \psi_{j-1}^{(E)}, \quad j = 1, 2, \dots \\ \psi_0^{(E)} &= 0, \quad \psi_1^{(E)} = s \end{aligned} \quad (9.43)$$

Since solutions satisfying the boundary condition  $\psi_{N-1}^{(E)} \approx 0$  only exists when  $E$  is an eigenvalue, we can begin by guessing the eigenvalue  $E^{(0)}$ , shoot to the right-hand boundary, then use a root-finding algorithm to generate a better approximation for the eigenvalue,  $E^{(1)}$ . This process is then repeated until  $E$  has converged.

### Node Counting

What if we would like to solve for a particular state (for example, the ground state, or the first excited state)? Recall that the time-independent Schrödinger equation results in *stationary states*, or *standing waves*. One property of standing waves is that the number of **nodes** (the points where the wavefunction crosses the  $x$ -axis) is related to the energy – the larger the number of nodes, the higher the energy of the state.

So for example, the ground state has zero nodes, the first excited state has one node, the second excited state has two nodes, etc. This provides an easy way of working out whether you are solving for the intended state.

### Bisecting the Energy Eigenvalue

One root finding method we can use is the **bisection method** (Sect. 4.3). In this method, we ‘bracket’ the correct eigenvalue, continually bisecting the upper and lower estimates until it converges to the correct eigenvalue.

To numerically solve for the  $n$ th eigenstate and the eigenvalue of the time dependent Schrödinger equation:

- (1) Estimate lower ( $E_{min}$ ) and upper ( $E_{max}$ ) bounds for the energy
- (2) Bisect the energy range to determine an estimate for the energy

$$E = \frac{1}{2}(E_{min} + E_{max})$$

- (3) Shoot from the left boundary to the right boundary by choosing  $s \sim 10^{-5}$  and solving the initial value problem

$$\psi_{j+1}^{(E)} = 2 \left[ \frac{m(\Delta x)^2}{\hbar^2} (V_j - E) + 1 \right] \psi_j^{(E)} - \psi_{j-1}^{(E)}, \quad \psi_0^{(E)} = 0, \quad \psi_1^{(E)} = s$$

- (4) Count the number of nodes  $n'$  in the solution  $\psi^{(E)}$ 
  - (a) If  $n' < n$ , then set  $E_{min} = E$  and return to step (2)
  - (b) If  $n' > n$ , then set  $E_{max} = E$  and return to step (2)
  - (c) if  $n' = n$ , then the correct eigenstate is being computed; proceed to step (5)
- (5) Bisect the energy eigenvalue bounds to better approximate the boundary condition  $\psi_{N-1}^{(E)} \approx 0$ 
  - (a) If  $\psi_{N-1}^{(E)} \psi_{N-1}^{(E)} > 0$ , then set  $E_{min} = E$
  - (b) If  $\psi_{N-1}^{(E)} \psi_{N-1}^{(E)} < 0$ , then set  $E_{max} = E$

► Whilst  $s$  is a free shooting parameter, in practice it is better to choose

$$|s| \sim 10^{-5} \ll 1$$

This acts to limit the ‘growth’ of the unnormalised shooting, avoiding overflow error, and round-off error caused by subtracting very large floating point numbers.

For example, choosing  $s \sim 1$  can lead to solutions with global maximums of order  $10^6$ !

(6) Check energy eigenvalue convergence:

$$|E_{\max} - E_{\min}| \leq \epsilon \quad \text{or} \quad |\psi_{N-1}^{(E)}| \leq \epsilon \quad \text{for some } \epsilon \ll 1$$

- (a) if the solution is not convergent, return to step (2)
- (b) if the solution is convergent, use numerical integration to normalise the solution  $\psi_{N-1}^{(E)}$ . This then provides the  $n$ th eigenstate  $\psi_n(x)$ , with energy  $E_n = E$ .

### Symmetric potentials

If  $n$  is even, then the wavefunction will be **even** or **symmetric**,

$$\psi(-x) = \psi(x).$$

If  $n$  is odd, then the wavefunction will be **odd** or **antisymmetric**,

$$\psi(-x) = -\psi(x)$$

So if we choose

$$s \sim (-1)^n 10^{-5},$$

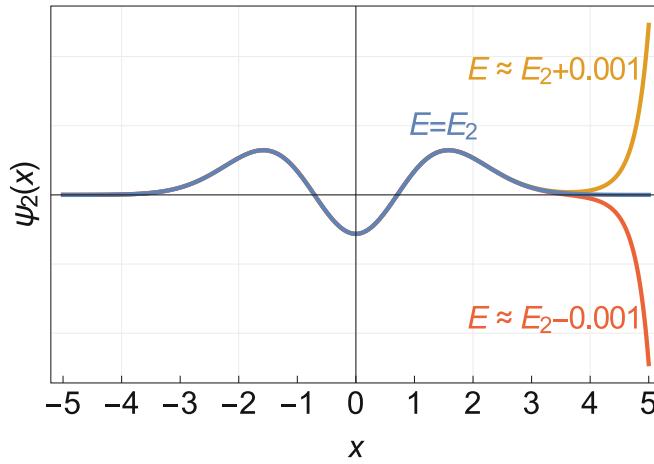
then we will always approach the boundary **from above** ( $\psi_{N-2} > \psi_{N-1} = 0$ ), and the bisection condition simplifies to

$$\psi_{N-1}^{(E)} > 0 \rightarrow E_{\min} = E$$

$$\psi_{N-1}^{(E)} < 0 \rightarrow E_{\max} = E.$$

### 9.4.2 The Matching Method

One problem with the shooting method is that the solution to the initial value problem tends to become **numerically unstable** near the boundary we are shooting towards.



For example, look at the figure above. Performing the shooting method with just a 0.001 error in the value of  $E$  leads to relatively accurate results until we approach the right boundary – here, the shooting solutions tends to grow almost exponentially in magnitude away from the  $x$ -axis!

One way to avoid this issue is by shooting from *both* boundaries simultaneously, and matching them up where they meet. This is the idea behind the **matching method**, also known as the **multiple shooting method**.

#### Multiple Shooting

It is relatively simple to modify our discrete version of the Schrödinger equation to shoot backwards – simply rearrange the equation to make  $\psi_{j-1}$  the subject.

##### Outward Shooting from the Left Boundary:

$$\begin{aligned}\overset{\rightarrow}{\psi}_{j+1}^{(E)} &= 2 \left[ \frac{m(\Delta x)^2}{\hbar^2} (V_j - E) + 1 \right] \overset{\rightarrow}{\psi}_j^{(E)} - \overset{\rightarrow}{\psi}_{j-1}^{(E)}, \quad j = 1, 2, \dots, m-1 \\ \overset{\rightarrow}{\psi}_0^{(E)} &= 0, \quad \overset{\rightarrow}{\psi}_1^{(E)} = s\end{aligned}$$

##### Inward Shooting from the Right Boundary:

$$\begin{aligned}\overset{\leftarrow}{\psi}_{j-1}^{(E)} &= 2 \left[ \frac{m(\Delta x)^2}{\hbar^2} (V_j - E) + 1 \right] \overset{\leftarrow}{\psi}_j^{(E)} - \overset{\leftarrow}{\psi}_{j+1}^{(E)}, \quad j = N-2, N-3, \dots, m+1 \\ \overset{\leftarrow}{\psi}_{N-1}^{(E)} &= 0, \quad \overset{\leftarrow}{\psi}_{N-2}^{(E)} = s'\end{aligned}$$

As before, it is good practice to set the free shooting parameters  $s$  and  $s'$  such that  $s, s' \ll 1$ .

#### ► What causes this instability?

Recall that when  $V > E$ , the possible solutions to the Schrödinger equation are asymptotic to

$$\{e^x, e^{-x}\},$$

however only the decaying solution  $e^{-x}$  is physically allowed.

When solving the system numerically, however, numerical error causes the non-physical solution  $e^x$  to influence the result.

The overall solution must be continuous and differentiable. If  $x_m$  is the matching location, we therefore impose the following **matching conditions** on the two shooting solutions:

1. Continuity:

$$\overset{\rightarrow(E)}{\psi}_m = \overset{\leftarrow(E)}{\psi}_m , \quad (9.44)$$

2. Differentiability:

$$\overset{\rightarrow}{\psi}'(x_m) = \overset{\leftarrow}{\psi}'(x_m) \Rightarrow \frac{\overset{\rightarrow(E)}{\psi}_m - \overset{\rightarrow(E)}{\psi}_{m-1}}{\Delta x} = \frac{\overset{\leftarrow(E)}{\psi}_{m+1} - \overset{\leftarrow(E)}{\psi}_m}{\Delta x}, \quad (9.45)$$

where we have used both the forward and backward first-order finite difference method to approximate the derivative.

Now, getting the inward and outward solutions to be continuous is easy – we simply scale the outward (or inward) solution by a constant value  $C$  so that they meet at point  $x_n$ :

$$\psi^{(E)} = \overset{\rightarrow(E)}{\psi} = C \overset{\leftarrow(E)}{\psi}, \quad \text{where } C = \frac{\overset{\rightarrow(E)}{\psi}_m}{\overset{\leftarrow(E)}{\psi}_m}. \quad (9.46)$$

From here on, we will denote the matched shooting solution as  $\psi_j^{(E)}$  for simplicity.

The differentiability condition is a bit more tricky. We know that a solution to the Schrödinger equation satisfying the boundary conditions only exists for when  $E$  is an energy eigenvalue. Thus, the differentiability condition

$$g(E) = \psi_{m+1}^{(E)} + \psi_{m-1}^{(E)} - 2\psi_m^{(E)} = 0 \quad (9.47)$$

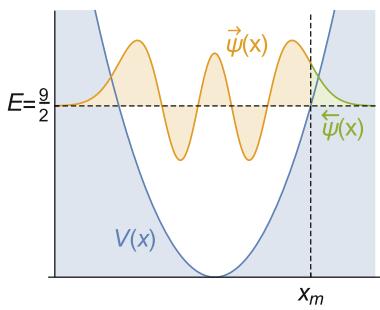
will only be satisfied for the correct values of  $E$ . This provides a method for better approximating  $E$  with each iteration (we can no longer use the right hand boundary condition for this, as we are shooting from there!), by using a root-finding algorithm on  $g(E) = 0$ .

### The Matching Point

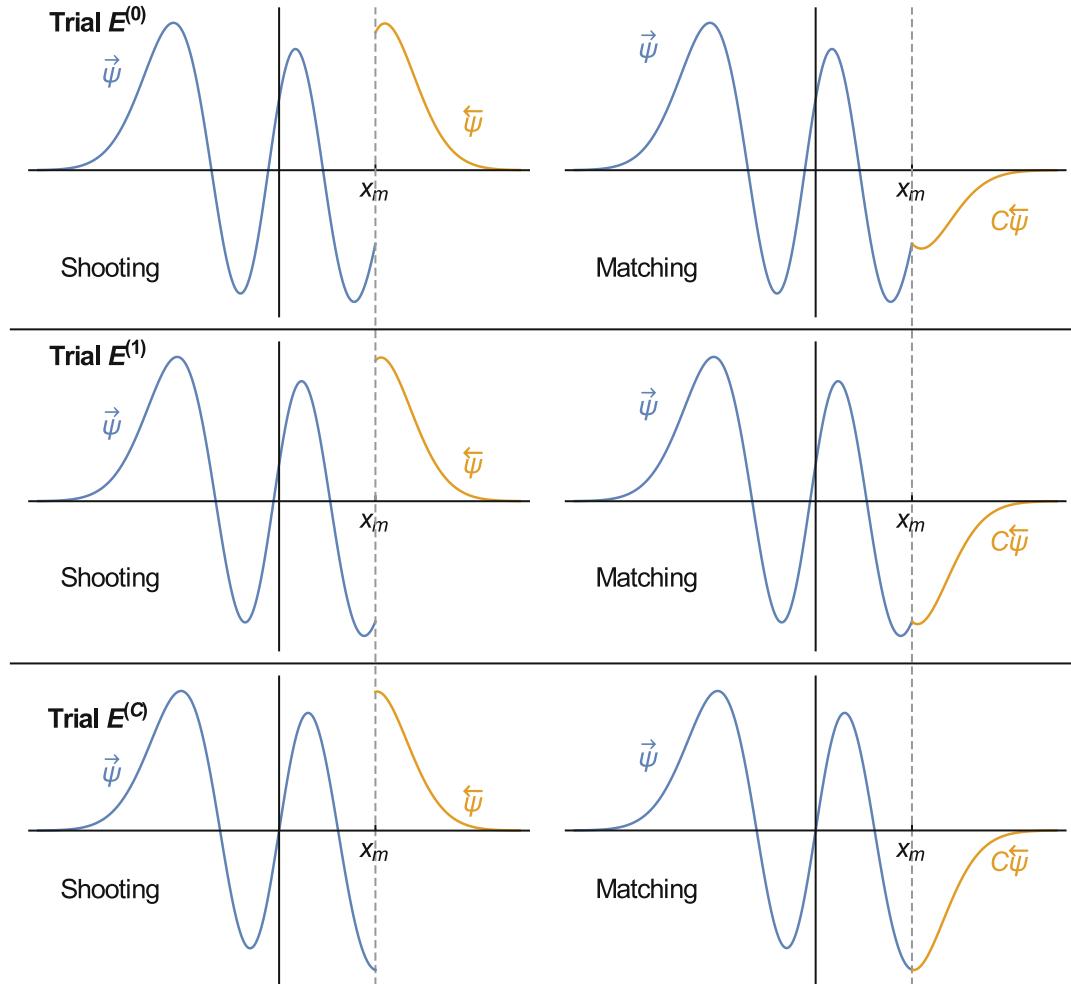
Where you perform the matching is up to you – as long as it is sufficiently far from the boundaries, error due to instability will be avoided. A common choice for the matching point is the classical turning point,

$$V(x) = E. \quad (9.48)$$

This has the advantage that node counting needs to be performed for only *one* of the shooting solutions. The other shooting solution will start from the boundary and terminate near the closest turning point, only slightly entering the region  $E > V(x)$  where oscillatory behaviour occurs.



**Figure 9.3** The matching point in this example is defined by  $V(x_m) = E$ . Note that the inward shooting is mostly contained in the region  $V(x) > E$



**Figure 9.4** An illustration of the matching method

The top row is the initial trial; the value of  $E^{(0)}$  is found by bisecting the chosen energy domain  $[E_{\min}, E_{\max}]$ . Note that after matching the two shooting solutions, the gradient does not match, so  $E^{(0)}$  is not a good estimate of the energy.

The middle row shows a better match using the new trial energy  $E^{(1)}$ . This was found by performing a root-finding algorithm on the differentiability condition Eq. 9.47 to determine a better approximation  $E^{(1)}$ .

The bottom row shows the result of the matching algorithm after the energy has converged. After normalisation, this is the final numerical solution.

### 9.4.3 Symmetric Potentials

When we are solving for the stationary states of a system with a symmetric potential, we can take advantage of the symmetry to reduce the amount of computation we need to do. We mentioned briefly before that, if the potential is symmetric,

$$V(-x) = V(x), \quad (9.49)$$

then the solutions  $\psi_n(x)$  are also symmetric. The exact symmetry, though, differs depending on the energy level/number of nodes  $n$ :

- If  $n$  is even, then  $\psi_n(x)$  is an **even** function:  $\psi_n(-x) = \psi_n(x)$ ,
- If  $n$  is odd, then  $\psi_n(x)$  is an **odd** function:  $\psi_n(-x) = -\psi_n(x)$ .

Putting these together,

$$\psi_n(-x) = (-1)^n \psi_n(x). \quad (9.50)$$

Thus, we need only determine the numerical solution for  $x \geq 0$ ; due to the symmetry of the system, we already have the remaining portion  $x < 0$ .

### Shooting from the Middle

Since we only need to worry about half the domain,  $x \in [0, \infty]$ , when working with symmetric potentials, the point  $x_0 = 0$  makes a good starting point for outward shooting.

The boundary conditions differ depending on whether the function is odd or even; if it is odd,  $x = 0$  corresponds to a **node** or **root**, and if it is even,  $x = 0$  corresponds to a **local maximum**.

- **If  $n$  is odd:**

$$\psi_0^{(E)} = 0, \quad \psi_1^{(E)} = s, \quad (9.51)$$

where  $s \sim 10^{-5}$  is the free shooting parameter

- **If  $n$  is even:**

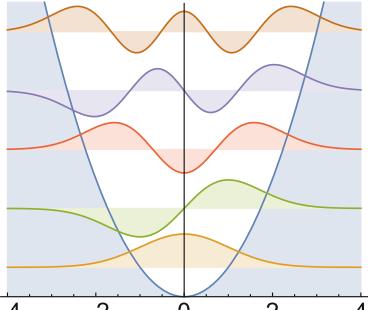
$$\psi_0^{(E)} = 1, \quad \psi_1^{(E)} = \left[ \frac{m(\Delta x)^2}{\hbar^2} (V_0 - E) + 1 \right] \psi_0^{(E)}, \quad (9.52)$$

where  $\psi_1^{(E)}$  is derived from the discretised Schrödinger equation (9.37), since, due to the symmetry,  $\psi_{-1}^{(E)} = \psi_1^{(E)}$

The process for the boundary conditions, as well as the matching point for the inward shooting from the right hand boundary remain unchanged.

Once the shooting method has converged to the correct boundary value on one side of the potential, the solution for the full domain can be easily calculated, and then normalised:

$$\psi_{-j} = (-1)^n \psi_j. \quad (9.53)$$



**Figure 9.5** The ground state and the first four excited states of the quantum harmonic oscillator

Note that the states with an even number of nodes  $n$  are even, whereas those with an odd number of nodes are odd.

► Since we are only using half the domain, make sure to modify your node counting! If  $n'$  is the number of nodes counted:

For even functions, the total number of nodes is

$$n = 2n';$$

For odd functions, we need to take into account the node at  $x_0$ . The total number of nodes is

$$n = 2n' + 1.$$

## 9.5 The Numerov–Cooley Shooting Method

If we would like to achieve higher accuracy in the shooting/matching method of solving the one-dimensional time-independent Schrödinger equation, one approach is simply to discretise the wavefunction using a high order method – for example, a fourth-order Runge-Kutta.

However, it turns out the form of the Schrödinger equation is ideally suited to a fourth-order method known as Numerov's method. Paired with James William Cooley's energy correction formula, the resulting **Numerov–Cooley method** has high accuracy and converges rapidly to the required solution.

### Numerov's Method

If a second-order differential equation can be written in the form

$$y''(x) + P(x)y(x) = R(x) \quad (9.54)$$

(i.e. there is no first derivative term;  $Q(x) = 0$ ) then we can implement the Numerov method. Let's derive it here.

Let  $\Delta x \ll 1$  be a small deviation in  $x$ . Then, from the Taylor series expansion around  $\Delta x$ , we have:

$$\begin{aligned} y(x + \Delta x) &= y(x) + y'(x)\Delta x + \frac{1}{2}(\Delta x)^2 y''(x) + \frac{1}{6}(\Delta x)^3 y'''(x) + \frac{1}{24}(\Delta x)^4 y^{(4)}(x) + \dots \\ y(x - \Delta x) &= y(x) - y'(x)\Delta x + \frac{1}{2}(\Delta x)^2 y''(x) - \frac{1}{6}(\Delta x)^3 y'''(x) + \frac{1}{24}(\Delta x)^4 y^{(4)}(x) + \dots \end{aligned}$$

Summing these two expansions together, all terms with odd powers of  $\Delta x$  cancel out, giving

$$y(x + \Delta x) + y(x - \Delta x) = 2y(x) + (\Delta x)^2 y''(x) + \frac{1}{12}(\Delta x)^4 y^{(4)}(x) + \mathcal{O}(\Delta x^6).$$

We can work out  $y^{(4)}$  simply by taking the second derivative of  $y''(x)$  and applying the method of finite differences:

$$y^{(4)}(x) = \frac{d^2}{dx^2} y''(x) = \frac{y''(x + \Delta x) - 2y''(x) + y''(x - \Delta x)}{(\Delta x)^2} + \mathcal{O}(\Delta x^2).$$

Furthermore, we know that  $y''(x) = R(x) - P(x)y(x)$  from the differential equation. Substituting these into our combined Taylor series expansion and rearranging, we get:

$$\begin{aligned} &y(x + \Delta x) \left(1 + \frac{(\Delta x)^2}{12} P(x + \Delta x)\right) - 2y(x) \left(1 - \frac{5(\Delta x)^2}{12} P(x + \Delta x)\right) \\ &\quad + y(x - \Delta x) \left(1 + \frac{(\Delta x)^2}{12} P(x - \Delta x)\right) \\ &= \frac{(\Delta x)^2}{12} [R(x + \Delta x) + 10R(x) + R(x - \Delta x)] + \mathcal{O}(\Delta x^6). \end{aligned}$$

► If the ODE is **non-linear**,

$$y''(x) = f(y(x), x),$$

then Numerov's method can still be applied.

However, it now leads to an **implicit method** of the form

$$y_{j+1} - 2y_j + y_{j-1} = \frac{\Delta x^2}{12} (f_{j+1} + 10f_j + f_{j-1})$$

- ▶ For a second-order ODE, the **global error** scaling is given by

$$\begin{aligned} \text{global error} = \\ \text{local error} \times \mathcal{O}\left(\frac{1}{\Delta x^2}\right) \end{aligned}$$

- ▶ Note that the accumulation of local error over the entire Numerov iteration process, the **global error**, is proportional to  $\Delta x^4$ .

This is the same order as the 4th order Runge-Kutta, but with significantly less computation per step.

We can apply Numerov's method to the time-independent Schrödinger equation, where  $y(x) = \psi(x)$ ,  $P(x) = -2m[V(x) - E]/\hbar^2$ , and  $R(x) = 0$ . If we now discretise the position space such that  $x_{j+1} - x_j = \Delta x$ , and use the notation  $f(x_j) \equiv f_j$ , this gives the recurrence relation

$$\psi_{j+1}^{(E)} \left(1 + \frac{(\Delta x)^2}{12} P_{j+1}\right) = 2\psi_j^{(E)} \left(1 - \frac{5(\Delta x)^2}{12} P_j\right) - \psi_{j-1}^{(E)} \left(1 + \frac{(\Delta x)^2}{12} P_{j-1}\right)$$

where  $P_j = -\frac{2m}{\hbar^2}(V_j - E)$ . This is accurate to sixth order in  $\Delta x$ . Numerov's iterative method can therefore be used to implement the matching method, with higher accuracy than the previous finite-difference discretisation.

Alternatively, the Numerov discretised form of the time-independent Schrödinger equation  $(\hat{H} - E)\psi_j^{(E)} = 0$  can be written in the more compact form

$$\begin{aligned} & -\frac{\hbar^2}{2m} \frac{Y_{j+1}^{(E)} - 2Y_j^{(E)} + Y_{j-1}^{(E)}}{(\Delta x)^2} + (V_j - E)\psi_j^{(E)} = 0 \\ \text{where } Y_j^{(E)} &= \left[1 + \frac{1}{12}(\Delta x)^2 P_j\right] \psi_j^{(E)} = \left[1 - \frac{m}{6\hbar^2}(\Delta x)^2(V_j - E)\right] \psi_j^{(E)} \end{aligned} \quad (9.56)$$

### 9.5.1 Cooley's Energy Correction Formula

However, Numerov's method by itself doesn't affect the **convergence** of our iterative method, just the accuracy of the shooting – we still need to continually bisect our domain  $[E_{min}, E_{max}]$  until we have accurately determined the correct energy eigenvalue.

Using Cooley's energy correction formula, though, we can use a perturbation theory approach to make a much better approximation to the energy with each iteration:

$$E^{(k+1)} = E^{(k)} + \Delta E \quad (9.57)$$

where

$$\Delta E \approx \frac{\psi_m^{(E_0)*}}{\sum_{j=0}^{N-1} |\psi_j^{(E_0)}|^2} \left[ -\frac{\hbar^2}{2m} \frac{Y_{m+1}^{(E_0)} - 2Y_m^{(E_0)} + Y_{m-1}^{(E_0)}}{(\Delta x)^2} + (V_m - E_0)\psi_m^{(E_0)} \right] \quad (9.58)$$

and  $x_m$  is the matching point of the inward and outward shooting.

Combined with Numerov's method, we have a highly accurate and fast converging iterative process.

- ▶ Cooley's energy correction formula estimates  $\Delta E$  to **first order**.

### Deriving Cooley's Energy Correction Formula

Let  $E^{(0)}$  be our initial guess for the energy of the system. Since the energy of a stationary state can be calculated by  $E = \langle \psi | \hat{H} | \psi \rangle$ , it follows from **perturbation theory** that the *deviation* between our initial estimate and the actual value can be approximated by

$$\Delta E = E - E_0 \approx \frac{\langle \psi^{(E_0)} | \hat{H} - E_0 | \psi^{(E_0)} \rangle}{\langle \psi^{(E_0)} | \psi^{(E_0)} \rangle} + \mathcal{O}(\Delta E^2), \quad (9.59)$$

where  $|\psi^{(E_0)}\rangle$  is the wavefunction solution after applying the Numerov method with  $E = E_0$ , and the denominator is for normalisation.

If  $\Delta x$  is sufficiently small ( $\Delta x \ll 1$ ) then we can approximate the integrals in the previous expression as Riemannian sums,

$$\langle \psi^{(E_0)} | \psi^{(E_0)} \rangle = \int_{-\infty}^{\infty} \psi^{(E_0)}(x)^* \psi^{(E_0)}(x) dx \approx \Delta x \sum_{j=0}^{N-1} |\psi_j^{(E_0)}|^2, \quad (9.60)$$

and similarly

$$\langle \psi^{(E_0)} | \hat{H} - E_0 | \psi^{(E_0)} \rangle \approx \Delta x \sum_{j=0}^{N-1} \psi_j^{(E_0)*} [(\hat{H} - E_0) \psi_j^{(E_0)}]. \quad (9.61)$$

Now, we can replace  $(\hat{H} - E_0) \psi_j^{(E_0)}$  by its Numerov discretisation (Eq. 9.56):

$$\begin{aligned} & \langle \psi^{(E_0)} | \hat{H} - E_0 | \psi^{(E_0)} \rangle \\ & \approx \Delta x \sum_{j=0}^{N-1} \psi_j^{(E_0)*} \left[ -\frac{\hbar^2}{2m} \frac{Y_{j+1}^{(E_0)} - 2Y_j^{(E_0)} + Y_{j-1}^{(E_0)}}{(\Delta x)^2} + (V_j - E_0) \psi_j^{(E_0)} \right]. \end{aligned}$$

We used the Numerov method for the inward and outward shooting, so the term in the square brackets must vanish for  $0 \leq j < m$  and  $m < j \leq N-1$ , where  $x = x_m$  is the matching point. However at the matching point, since our energy  $E^{(0)}$  is simply an estimate, the gradient of the inward and outward solutions will *not* match, leading to a discontinuity in the first derivative – no longer satisfying the Schrödinger equation.

The sum therefore collapses to the single value  $j = m$ :

$$\begin{aligned} & \langle \psi^{(E_0)} | \hat{H} - E_0 | \psi^{(E_0)} \rangle \\ & \approx \psi_m^{(E_0)*} \Delta x \left[ -\frac{\hbar^2}{2m} \frac{Y_{m+1}^{(E_0)} - 2Y_m^{(E_0)} + Y_{m-1}^{(E_0)}}{(\Delta x)^2} + (V_m - E_0) \psi_m^{(E_0)} \right]. \end{aligned}$$

Substituting this back into Eq. 9.59,

$$\Delta E \approx \frac{\psi_m^{(E_0)*}}{\sum_{j=0}^{N-1} |\psi_j^{(E_0)}|^2} \left[ -\frac{\hbar^2}{2m} \frac{Y_{m+1}^{(E_0)} - 2Y_m^{(E_0)} + Y_{m-1}^{(E_0)}}{(\Delta x)^2} + (V_m - E_0) \psi_m^{(E_0)} \right].$$

## 9.6 The Direct Matrix Method

So far, we have examined the shooting, matching, and Numerov–Cooley methods of solving the one-dimensional time-independent Schrödinger’s equation. These methods all utilise the method of finite differences to discretise our position space, followed by a bisection algorithm (shooting, matching, and Cooley’s energy correction formula, respectively) to determine the energy values allowed. An alternative to this approach is the *direct matrix method* — by writing the method of finite differences in matrix form, we can instead use an eigenvalue solver to determine the allowed values of  $E$ .

Let’s return to the time-independent Schrödinger equation

$$\hat{H}\psi(x) = E\psi(x), \quad (9.62)$$

where

$$\hat{H} = -\frac{\hbar^2}{2m} \frac{d^2}{dx^2} + V(x), \quad (9.63)$$

and with boundary conditions  $\psi(x_0) = \alpha$ ,  $\psi(x_{N-1}) = \beta$ , and see if we can derive another numerical method for finding the solution.

### 9.6.1 The Finite Difference Matrix

Recall that by applying the **central difference formula** to approximate the second derivative to second-order, this can be written in the discretised form

$$-\frac{\hbar^2}{2m} \frac{\psi_{j+1} - 2\psi_j + \psi_{j-1}}{\Delta x^2} + V_j \psi_j = E\psi_j + \mathcal{O}(\Delta x^2), \quad j = 1, 2, \dots, N-2$$

where  $x_{j+1} - x_j = \Delta x$  and  $f(x_j) \equiv f_j$ . If we rearrange this equation in the following fashion,

$$-k\psi_{j+1} + (2k + V_j)\psi_j - k\psi_{j-1} = E\psi_j + \mathcal{O}(\Delta x^2), \quad \text{with } k = \frac{\hbar^2}{2m\Delta x^2},$$

then this allows us to write the discretised system as a set of coupled linear equations:

$$\left\{ \begin{array}{llll} j = 1 : & -k\psi_2 + (2k + V_1)\psi_1 & -k\alpha & = E\psi_1 \\ j = 2 : & -k\psi_3 + (2k + V_2)\psi_2 & -k\psi_1 & = E\psi_2 \\ & \vdots & & \vdots \\ j = N-2 : & -k\beta + (2k + V_{N-2})\psi_{N-2} & -k\psi_{N-3} & = E\psi_{N-2}. \end{array} \right.$$

Therefore, by solving this system of linear equations, we can solve for the coefficients  $\psi_j$ ! Moving the  $k\alpha$  and  $k\beta$  terms to the right-hand side, this can

be written as a matrix equation

$$\begin{bmatrix} 2k + V_1 & -k & 0 & \cdots & 0 \\ -k & 2k + V_2 & -k & \ddots & \vdots \\ 0 & -k & 2k + V_3 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -k \\ 0 & \dots & 0 & -k & 2k + V_{N-2} \end{bmatrix} \begin{bmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \vdots \\ \psi_{N-2} \end{bmatrix} = E \begin{bmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \vdots \\ \psi_{N-2} \end{bmatrix} + k \begin{bmatrix} \alpha \\ 0 \\ 0 \\ \vdots \\ \beta \end{bmatrix}$$

or

$$H\psi = E\psi + k\mathbf{b} + \mathcal{O}(\Delta x^2) \quad (9.64)$$

where  $H$  is the **discretised Hamiltonian**;  $\psi$  is a vector representing the discrete wavefunction values; and  $\mathbf{b}$  is a constant vector such that  $b_1 = \psi_0 = \alpha$ ,  $b_{N-2} = \psi_{N-1} = \beta$ , and with all other entries 0.

The solution to this equation is

$$\psi = k(H - EI)^{-1}\mathbf{b} + \mathcal{O}(\Delta x^2). \quad (9.65)$$

Note that  $H - EI$  is a **symmetric tridiagonal matrix** (the only non-zero entries are on the diagonal, sub-diagonal, and super-diagonal), so the solution to  $\psi$  can be found using tridiagonal linear solvers (for example, the LAPACK subroutines ZGTSV or ZPTSV).

Unfortunately, this method requires us to iterate through guesses for the correct energy eigenvalue  $E$  – only the correct value of  $E$  will lead to a differentiable solution.

### Boundary Values and Eigenvalues

When working with bound states or standing waves, however, we'll most likely have the boundary conditions

$$\psi_0 = 0, \quad \psi_{N-1} = 0. \quad (9.66)$$

In this case, the boundary value vector reduces to the zero vector,  $\mathbf{b} = \mathbf{0}$ , and Eq. 9.65 simplifies down to an **eigenvalue equation**

$$H\psi = E\psi + \mathcal{O}(\Delta x^2). \quad (9.67)$$

So simply solving for the eigenvalues and eigenvectors of our discretised Hamiltonian  $H$ , we get the wavefunctions  $\psi_j$  and their associated energy eigenvalues  $E$ .

► This solution is only well-defined if  $H - EI$  is **non-singular**.

► Depending on the numerical linear algebra solver used, additional error is likely to be accumulated.

► Other, higher order discrete approximations to the Schrödinger equation can be used to produce higher order matrix methods. However, this comes at the cost of more linear equations to solve/larger matrices.

Give this a go with the Numerov method in exercise P9.8

### A Note on Accuracy

Like the shooting method, the matrix method will only provide accurate results for reasonably small eigenvalues. In fact, you might even find that for more highly excited states, the resulting error in your calculation of the eigenvalues actually *increases* as  $\Delta x$  decreases!

This could occur if the boundary conditions  $\psi(x_0) = \psi(x_n) \approx 0$  are no longer satisfied by these highly excited states, due to the  $n$ th energy eigenvalue  $E_n$  being less than or equal to the value of the potential  $V(x)$  at the boundaries of the finite grid.

So, keep this in mind when using the direct matrix method.

#### Further reading

The techniques applied to the Schrödinger equation in this chapter can be generalised to other partial differential equations with boundary conditions, and thus can be useful in solid state physics, electromagnetism, condensed matter, and other fields. The following resources are aimed at the graduate level, but present an in-depth study of numerical techniques for solving differential equations.

- Langtangen, H. P., & Linge, S. (2017). Finite Difference Computing with PDEs (Vol. 16). Cham: Springer International Publishing. <https://doi.org/10.1007/978-3-319-55456-3>
- Thomas, J. W. (1995). Numerical Partial Differential Equations: Finite Difference Methods (Vol. 22). New York, NY: Springer New York. ISBN 978-1-4899-7278-1
- Smith, G. D. (1985). Numerical Solutions of Partial Differential Equations: Finite Difference Methods (3rd ed). Oxford [Oxfordshire]: New York: Clarendon Press; Oxford University Press.

## Exercises

### P9.1 The quantum harmonic oscillator

Consider an electron in a quantum harmonic potential. The potential is

$$V(x) = \frac{1}{2}m_e\omega^2x^2$$

where  $m_e$  is the electron mass, and  $\omega$  is the angular frequency of the harmonic potential.

- (a) For  $\hbar = 1$ ,  $m_e = 1$ , and  $\omega = 1$ , use the bisecting shooting method to calculate the wavefunctions  $\psi_n(x)$  and respective energies  $E_n$  of the first four energy levels ( $n = 0, 1, 2, 3$ ) of the quantum harmonic oscillator.
- (b) What is the spacing  $\Delta E = E_{j+1} - E_j$  between successive energy levels?
- (c) Compare your calculated energies to the exact result

$$E_n = \omega \left( n + \frac{1}{2} \right)$$

- (d) The exact solutions for the quantum harmonic wavefunctions are

$$\psi_n(x) = \frac{1}{\sqrt{2^n n!}} \left( \frac{m\omega}{\pi\hbar} \right)^{1/4} e^{-m\omega x^2/2\hbar} H_n \left( \sqrt{\frac{m\omega}{\hbar}} x \right)$$

Compare these to the calculated wavefunctions for  $n = 0, 1, 2, 3$ .

- (e) Plot the absolute and relative error between the numerical and the exact wavefunction solutions. In what regions of the  $x$  domain is the error largest? Why might this be the case?

### P9.2

Modify your code from above to calculate the wavefunctions and energies of the quantum harmonic oscillator using the **method of false positions** (Sect. 4.6). That is, rather than bisecting your upper and lower energy bound,

$$E = \frac{1}{2}(E_{min} + E_{max})$$

instead use the secant method to determine a better approximation to  $E$ :

$$E = E_{max} - \psi_{N-1}^{(E_{max})} \frac{E_{max} - E_{min}}{\psi_{N-1}^{(E_{max})} - \psi_{N-1}^{(E_{min})}}$$

► For simplicity, we will use **atomic units** for problem sets involving numerical solutions to the Schrödinger equation:

$$m_e = q_e = \hbar = 1$$

where  $m_e$  is the electron mass and  $q_e$  is the electron charge.

How does this new algorithm compare to the previous one? Consider both the accuracy, convergence, and computation time.

- P9.3** Solve for the first four energy levels and wavefunctions of the quantum harmonic oscillator, as per **P9.1**, however this time implement it using the matching method.

Make use of the symmetry of the potential to shoot outward from  $x = 0$ , choosing the classical turning point

$$E = V(x_m) = \frac{1}{2}x_m^2$$

as the matching point  $x_m$ .

Compare your results to the analytical solutions. How does the error and convergence of the matching method compare to the shooting method?

#### P9.4 The Morse potential

The one-dimensional Morse potential,

$$V(x) = C_0 + D_e \left(1 - e^{-a(x-x_e)}\right)^2$$

is commonly used to model the potential energy between two atoms of a diatomic molecule. Here,  $x$  is the distance between the atoms,  $x_e$  is the equilibrium bond distance,  $D_e$  is the well depth, and  $a$  is a constant parameter.

► Hint: plot the potential in order to determine the boundary conditions and an appropriate finite domain!

- Use the matching method to obtain the energy and wavefunction for the **ground state** ( $n = 0$ ) of the one-dimensional Morse potential, for the case  $C_0 = 6.4$ ,  $D_e = 0.3$ ,  $a = 1$ , and  $x_e = 1.6$ . Use atomic units ( $\hbar = m = 1$ ).
- The exact solution for the ground state energy is

$$E_0 = \frac{251 + 4\sqrt{15}}{40}.$$

What is the fractional error in your result?

- Compare your result to the exact solution to the ground state wavefunction, given by

$$\psi_0(x) = 0.886463 \exp(-0.774597e^{1.6-x}) (e^{1.6-x})^{0.274597}$$

- P9.5** We saw that the Numerov method for the Schrödinger equation can be written

$$\psi_{j+1}^{(E)} \left[ 1 + \frac{(\Delta x)^2}{12} P_{j+1} \right] = 2\psi_j^{(E)} \left[ 1 - \frac{5(\Delta x)^2}{12} P_j \right] - \psi_{j-1}^{(E)} \left[ 1 + \frac{(\Delta x)^2}{12} P_{j-1} \right]$$

where  $P_j = -\frac{2m}{\hbar^2}(V_j - E)$ . By substituting

$$Y_j^{(E)} = \left[ 1 + \frac{1}{12}(\Delta x)^2 P_j \right] \psi_j^{(E)}$$

into the equation above and rearranging, show that the Numerov approximation to the Schrödinger equation can also be written in the form

$$-\frac{\hbar^2}{2m} \frac{Y_{j+1}^{(E)} - 2Y_j^{(E)} + Y_{j-1}^{(E)}}{(\Delta x)^2} + (V_j - E)\psi_j^{(E)} = 0$$

- P9.6** Use the Numerov–Cooley method to obtain the energies and wavefunctions of the quantum harmonic oscillator, with potential

$$V(x) = \frac{1}{2}m_e\omega^2x^2$$

Let  $m_e = 1$ ,  $\hbar = 1$ ,  $\omega = 1$  for simplicity.

- (a) Demonstrate that the results for your energy are *independent* on the choice of computational parameters such as grid size ( $N$ ) and spacing ( $\Delta x$ )
  - (b) Compare your results to those obtained via the shooting or matching method. How does the accuracy compare for the same values of  $\Delta x$ ?
  - (c) Compare the computational cost of the Numerov–Cooley method to the shooting and/or matching method (for example, by timing the algorithms for the same set of parameters). Which algorithm appears more efficient? Why?
- P9.7**
- (a) Construct the finite difference matrix approximation to the Hamiltonian of the quantum harmonic oscillator

$$V(x) = \frac{1}{2}m_e\omega^2x^2$$

in Fortran, for  $-5 \leq x \leq 5$ , and  $N = 1000$ . Let  $m_e = 1$ ,  $\hbar = 1$ ,  $\omega = 1$  as before. Take advantage of the fact that  $H$  is Hermitian and displays band structure (see Sect. 7.5.2 for details).

- (b) Using the SciPy or the LAPACK subroutine ZHBEV, calculate the eigenvalues and eigenvectors of  $\hat{H}$ .
- (c) Compare the 10 smallest eigenvalues to the theoretical energy levels

$$E_n = \omega \left( n + \frac{1}{2} \right)$$

- (d) Plot the first 5 eigenvectors, and compare them to the analytical solutions.
- (e) For various values of  $\Delta x$ , calculate the ground state energy. Plot the absolute error in your results,  $|E_0 - 1/2|$ , as a function of  $\Delta x$ . How does the error scale with the grid spacing?
- (f) Compare the computational cost of the direct matrix method to the shooting and/or matching method (for example, by timing the algorithms for the same set of parameters). Which algorithm appears more efficient? Why?

**P9.8** Write the iterative form of the Numerov method (Eq. 9.56) as a system of coupled linear equations for a system with boundary conditions  $\psi_0 = \psi_{N-1} = 0$ . Using a similar method to Sect. 9.6.1, rewrite this as a matrix eigenvalue equation in terms of  $\psi = (\psi_0, \psi_1, \dots, \psi_{N-1})$ ,  $H_{ij}$ , and  $E$ .

This is the **Numerov direct matrix method** – it is of higher order than the finite difference matrix method. How does the error of this method scale with  $\Delta x$ ?

**P9.9 Infinite square well**

Consider an infinite square well potential,

$$V(x) = \begin{cases} 0, & -L/2 \leq x \leq L/2 \\ \infty, & \text{otherwise} \end{cases}$$

The resulting wavefunction solutions to the Schrödinger equation have *zero* probability of being located in the region  $|x| \geq L$  (i.e. the particle is fully confined to the square well), with the  $n$ th excited state having energy

$$E_n = \frac{n^2 \hbar^2 \pi^2}{2m_e L^2}$$

- (a) Letting  $m_e = 1$ ,  $\hbar = 1$ , and  $L = 2$ , construct the Numerov approximation to the Hamiltonian. Approximate the infinity in the potential by  $10^{10}$  and discretise the  $x$  coordinate with  $N = 5000$  points for  $-2 \leq x \leq 2$ . Take advantage of the fact that  $H$  is Hermitian and displays band structure (see Sect. 7.5.2 for details).
- (b) Using SciPy or the LAPACK subroutine ZHBEV, calculate the eigenvalues and eigenvectors of  $\hat{H}$ .
- (c) Compare the 5 smallest eigenvalues to the theoretical energy levels.
- (d) Plot the first 5 eigenvectors



## Chapter 10

# Higher Dimensions and Basic Techniques

While the numerical techniques we have introduced so far enable us to solve one-dimensional quantum systems that are intractable analytically, in general most physical systems that we would like to solve are not one-dimensional, but instead two- or three-dimensional. Unfortunately, the shooting or matching method, which we have applied successfully to one-dimensional problems, cannot be generalised to higher dimensions. The shooting method, the first method we introduced to solve the Schrödinger equation, allows us to convert a boundary value problem into a second-order initial value problem, bisecting the initial values until the boundary values are satisfied. In higher dimensions, not only do we have multiple initial value problems that we need to solve concurrently, we need to perform root finding algorithms on the initial values until they are satisfied along a line, or even a surface! As such, shooting-style algorithms are rarely used in dimensions larger than one — instead, techniques based on basis diagonalisation or variational techniques are preferred. In this chapter, we'll explore some common techniques used in higher dimensions, including direct matrix methods, basis diagonalisation, and Monte-Carlo techniques.

### 10.1 The Two-Dimensional Direct Matrix Method

In the previous chapter, we considered the direct matrix method to solve the one-dimensional time-independent Schrödinger equation; this involved applying the method of finite differences directly to the Schrödinger equation, allowing us to write it in matrix form, and utilise eigensolver algorithms. Can we extend this to two dimensions?

In two dimensions, the time-independent Schrödinger equation is

$$\hat{H}\psi(x, y) = E\psi(x, y), \quad (10.1)$$

where

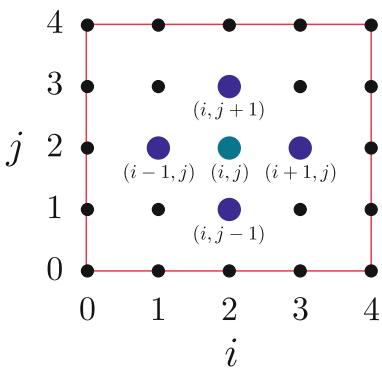
$$\hat{H} = -\frac{\hbar^2}{2m}\nabla^2 + V(x, y), \quad \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}. \quad (10.2)$$

- Note that, since we are in two dimensions, the boundary conditions now must be satisfied along the **lines**  $x = x_0, x = x_{N-1}, y = y_0$ , and  $y = y_{N-1}$  in the 2D plane.

and the boundary conditions are  $\psi(x_0, y) = \psi(x, y_0) = \psi(x_{N-1}, y) = \psi(x, y_{N-1}) = 0$ . To approximate the Laplacian  $\nabla^2$ , we can discretise it using the **central finite-difference formula** to approximate the second derivatives with respect to  $x$  and  $y$  to second order:

$$\begin{aligned} \nabla^2 &= \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \\ &= \frac{\psi_{i+1,j} - 2\psi_{i,j} + \psi_{i-1,j}}{\Delta x^2} + \frac{\psi_{i,j+1} - 2\psi_{i,j} + \psi_{i,j-1}}{\Delta y^2} \\ &\quad + \mathcal{O}(\max(\Delta x^2, \Delta y^2)) \end{aligned} \quad (10.3)$$

where we have discretised the two-dimensional grid such that  $x_{i+1} - x_i = \Delta x$ ,  $y_{j+1} - y_j = \Delta y$ ,  $V(x_i, y_j) \equiv V_{ij}$ , and  $\psi(x_i, y_j) \equiv \psi_{ij}$ .



**Figure 10.1** The action of the central finite-difference formula can be visualised by the above **stencil diagram**.

Here, each circle represents an element of the discretised function  $\psi(x_i, y_j) = \psi_{ij}$ , with the boundary points satisfying the boundary conditions marked by the red line.

To calculate the approximate Laplacian  $\nabla^2\psi_{ij}$  of the center point, we must consider the values of the surrounding points; horizontally for the  $x$  derivative, and vertically for the  $y$  derivative

### Local Truncation Error

What is the local truncation error for the two-dimensional central difference approximation  $\nabla^2 f(x, y)$ ? We can combine the two truncation errors from approximating the *one-dimensional* second derivatives  $\frac{d^2}{dx^2}$  and  $\frac{d^2}{dy^2}$  using the central finite difference method:

$$LE = \frac{1}{12}\Delta x^2 \frac{\partial^4}{\partial x^4} f(x, y) + \frac{1}{12}\Delta y^2 \frac{\partial^4}{\partial y^4} f(x, y) \quad (10.4)$$

Have a go taking the Taylor series expansion around  $\Delta x$  and  $\Delta y$  of the two-dimensional central difference formula, and verify that it agrees with the above. What is the order of the next highest term appearing in the local error?

Applying the **central difference formula** to approximate the second derivative to second order, this can be written in the following discretised form:

$$\begin{aligned} &- \frac{\hbar^2}{2m} \left( \frac{\psi_{i+1,j} - 2\psi_{i,j} + \psi_{i-1,j}}{\Delta x^2} + \frac{\psi_{i,j+1} - 2\psi_{i,j} + \psi_{i,j-1}}{\Delta y^2} \right) + V_{ij}\psi_{ij} \\ &= E\psi_{ij} + \mathcal{O}(\max(\Delta x^2, \Delta y^2)), \end{aligned} \quad (10.5)$$

As before, let's rearrange this to collect all  $\psi_{i,j}$  terms:

$$\begin{aligned} &- \mu\psi_{i,j-1} - \eta\psi_{i-1,j} + v_{ij}\psi_{ij} - \eta\psi_{i+1,j} - \mu\psi_{i,j+1} \\ &= E\psi_{ij} + \mathcal{O}(\max(\Delta x^2, \Delta y^2)), \end{aligned} \quad (10.6)$$

where we have defined  $v_{ij} = 2\mu + 2\eta + V_{ij}$ ,  $\eta = \frac{\hbar^2}{2m\Delta x^2}$ , and  $\mu = \frac{\hbar^2}{2m\Delta y^2}$  for convenience. Like we did with the one-dimensional case, inspecting the indices in the above equation allows us to re-frame this as a matrix-vector equation.

### 10.1.1 Flattening and Natural Ordering

The two-dimensional wavefunction — which we'd like to have in the form of a column vector — is now indexed by two indices, not one! To rewrite the wavefunction as a column vector, a common technique is to ‘flatten’ the wavefunction  $\psi$  into a one-dimensional column vector:

$$\begin{bmatrix} \psi_{11} & \psi_{12} & \psi_{13} \\ \psi_{21} & \psi_{22} & \psi_{23} \\ \psi_{31} & \psi_{32} & \psi_{33} \end{bmatrix} \rightarrow (\psi_{11} \ \psi_{21} \ \psi_{31} \ \psi_{12} \ \psi_{22} \ \psi_{32} \ \psi_{13} \ \psi_{23} \ \psi_{33})^T$$

There is a quite a large degree of freedom when it comes to flattening the wavefunction; how do we decide the order in which to flatten the wavefunction elements? The trick is to remain consistent; if you choose one ordering when flattening your wavefunction, make sure to use the same ordering when considering *all* other vectors and matrices. A common ordering is the so-called **natural ordering**, chosen as it results in a nice symmetric form for our matrix-vector equation. Consider the following stencil diagram:

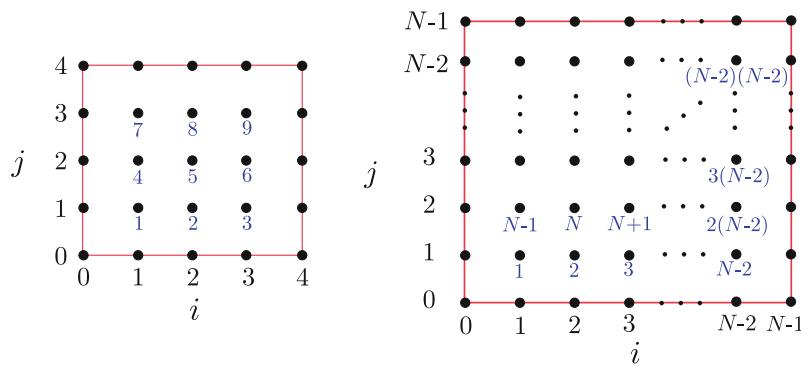


Figure 10.2

Here, each point represents a discrete value of the wavefunction  $\psi_{ij}$  — on the left we have the case  $N = 5$ , and on the right a generalisation to the arbitrary case  $N$ . To flatten the wavefunction with natural ordering, we do the following:

- Begin at the point  $(1, 1)$ ; this is the first element in our flattened vector, and we assign it the index  $k = 1$ .
- Increment  $i$  to traverse the horizontal  $x$ -direction.

► When writing down the error scaling in Big-O notation, we take the **maximum** of  $\Delta x^2$  or  $\Delta y^2$ , we don't include both.

This is because Big-O notation provides an **upper-bound** — we only care about the larger value.

► Diagrams like this of the discretised grid and border conditions are commonly referred to as **stencil diagrams**.

- When you have reached the end of the row, return to  $i = 1$ , and increment  $j$  to move to the start of the next row. Continue this process until the entire wavefunction has been indexed by  $k$ .

The numbers marked in blue are the *new indices*  $k$  of our flattened wavefunction vector  $\psi$ , using the natural ordering. You might have noticed that we are ignoring the elements of  $\psi_{ij}$  located on the boundary — since we already know the values of  $\psi$  at the boundaries from the boundary conditions, we do not need to take them into account when constructing our matrix-vector finite-difference equation. Thus, an  $N \times N$  discretised grid  $\psi_{ij}$  becomes a vector of length  $(N - 2)^2$ - $\psi$ , when flattened.

The natural ordering can also be written in the following vector form:

$$\psi = (\psi_{i,1}, \psi_{i,2}, \dots, \psi_{i,N-2}) \quad (10.7)$$

where  $\psi_{i,\ell} = (\psi_{1,\ell}, \psi_{2,\ell}, \dots, \psi_{N-2,\ell})$ . The  $k$ th element in this vector can then be mapped to the equivalent element of  $\psi_{ij}$  by the index remapping

$$k = (N - 2)(j - 1) + i, \quad \text{where } i, j = 1, 2, \dots, N - 2. \quad (10.8)$$

That is,  $\psi_{ij} = \psi_{(N-2)(j-1)+i}$ . This is a convenient technique, for two reasons; firstly, we are not losing any information about the system — by knowing the location of an element in the flattened array, we can recover  $(i, j)$  by the reverse mapping

$$\begin{cases} i = k \bmod (N - 2) + 1 \\ j = \left\lfloor \frac{k}{N-2} \right\rfloor \end{cases}, \quad (10.9)$$

where  $\lfloor x \rfloor$  is the floor function, returning the largest integer  $m$  such that  $m \leq x$ . Secondly, this is an efficient computational task when using Fortran or NumPy; we can make use of Fortran's intrinsic `reshape()` array function, or NumPy's `numpy.flatten()` and `numpy.reshape()` functions.

### Example 10.1 Natural order flattening (Fortran)

Consider the function  $f(x, y) = e^{-(x^2+y^2)/2}$ , which, along the lines  $x = 0$ ,  $x = 5$ ,  $y = 0$ , and  $y = 5$ , is such that  $f \approx 0$ . Discretise this function to create a 2D array over the region  $-5 \leq x, y \leq 5$ , using  $\Delta x = \Delta y = 0.1$ . Then, use Fortran to flatten this array using natural ordering.

**Solution:**

```
program flatten
    implicit none
    integer :: N, i, j
    real(8) :: dx, x, y
    real(8), allocatable :: f(:, :), fvec(:)
```

```

dx = 0.1d0
N = 10.d0/dx + 1

! allocate the discretised 2D f array and flattened vector
allocate(f(N, N), fvec((N-2)**2))

! Discretise the function over the 2D x and y grid
do i=1, N
    do j=1, N
        x = -5.d0+dx*(i-1)
        y = -5.d0+dx*(j-1)
        f(i,j) = exp(-(x**2 + y**2)/2)
    end do
end do

! Flatten the discrete function using natural ordering
fvec = reshape(f(2:N-1, 2:N-1), [(N-2)**2])
end program flatten

```

### Example 10.2 Natural order flattening (Python)

Consider the function  $f(x, y) = e^{-(x^2+y^2)/2}$ , which, along the lines  $x = -5$ ,  $x = 5$ ,  $y = -5$ , and  $y = 5$ , is such that  $f \approx 0$ . Discretise this function to create a 2D array over the region  $-5 \leq x, y \leq 5$ , using  $\Delta x = \Delta y = 0.1$ . Then, use NumPy to flatten this array using natural ordering.

**Solution:**

```

import numpy as np

# create the discrete x and y grids
dx = 0.1
x = np.arange(-5, 5+dx, dx) # indexed by i
y = np.arange(-5, 5+dx, dx)[:, None] # indexed by j

# Discretise the function over these grids
f = np.exp(-(x**2+y**2)/2)

# Flatten f using natural ordering.
# As we know at the boundary f is approximately 0,
# we only flatten the non-boundary points.
f[1:-1, 1:-1].flatten()

```

#### 10.1.2 The Two-Dimensional Finite Difference Matrix

Now that we have flattened  $\psi_{ij}$  to the vector  $\psi$ , we can write Eq. 10.6 as a set of coupled linear equations. For simplicity, let's consider the case  $N = 5$

to get a feel for this process; once done, it will be easier to visualise how this generalises to arbitrary values of  $N$ .

When  $N = 5$ , discarding the boundary conditions ( $\psi_{0,j} = \psi_{4,j} = \psi_{i,0} = \psi_{i,4} = 0$ ), we have  $k = (N - 2)^2 = 9$  coupled linear equations:

$$\begin{aligned} k = 1 : & + v_{1,1}\psi_1 - \eta\psi_2 - \mu\psi_4 = E\psi_1 \\ k = 2 : & - \eta\psi_1 + v_{2,1}\psi_2 - \eta\psi_3 - \mu\psi_5 = E\psi_2 \\ k = 3 : & - \eta\psi_2 + v_{3,1}\psi_3 - \mu\psi_6 = E\psi_3 \\ k = 4 : & - \mu\psi_1 + v_{1,2}\psi_4 - \eta\psi_5 - \mu\psi_7 = E\psi_4 \\ k = 5 : & - \mu\psi_2 - \eta\psi_4 + v_{2,2}\psi_5 - \eta\psi_6 - \mu\psi_7 = E\psi_5 \\ k = 6 : & - \mu\psi_3 - \eta\psi_5 + v_{3,2}\psi_6 - \mu\psi_9 = E\psi_6 \\ k = 7 : & - \mu\psi_4 + v_{1,3}\psi_7 - \eta\psi_8 = E\psi_7 \\ k = 8 : & - \mu\psi_5 - \eta\psi_7 + v_{2,3}\psi_8 - \eta\psi_9 = E\psi_8 \\ k = 9 : & - \mu\psi_6 - \eta\psi_8 + v_{3,3}\psi_9 = E\psi_9 \end{aligned}$$

This can be written as the following matrix equation,

$$H\psi = E\psi + \mathcal{O}(\max(\Delta x^2, \Delta y^2)), \quad (10.10)$$

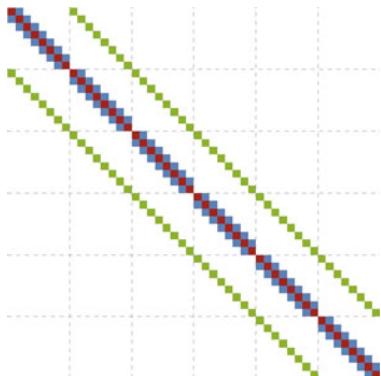
where  $H$  is the **discretised two-dimensional Hamiltonian**:

$$H = \begin{bmatrix} v_{1,1} & -\eta & 0 & -\mu & 0 & 0 & 0 & 0 & 0 \\ -\eta & v_{2,1} & -\eta & 0 & -\mu & 0 & 0 & 0 & 0 \\ 0 & -\eta & v_{3,1} & 0 & 0 & -\mu & 0 & 0 & 0 \\ -\mu & 0 & 0 & v_{1,2} & -\eta & 0 & -\mu & 0 & 0 \\ 0 & -\mu & 0 & -\eta & v_{2,2} & -\eta & 0 & -\mu & 0 \\ 0 & 0 & -\mu & 0 & -\eta & v_{3,2} & 0 & 0 & -\mu \\ 0 & 0 & 0 & -\mu & 0 & 0 & v_{1,3} & -\eta & 0 \\ 0 & 0 & 0 & 0 & -\mu & 0 & -\eta & v_{2,3} & -\eta \\ 0 & 0 & 0 & 0 & 0 & -\mu & 0 & -\eta & v_{3,3} \end{bmatrix} \quad (10.11)$$

The dashed lines in the above discrete 2D Hamiltonian are added to draw attention to the **block-diagonal** structure of the matrix; notice that the diagonal is composed of  $(N - 2) \times (N - 2)$  matrices of the form

$$A_\ell = \begin{bmatrix} v_{1,\ell} & -\eta & & & \\ -\eta & v_{2,\ell} & -\eta & & \\ & \ddots & \ddots & \ddots & \\ & & -\eta & v_{N-3,\ell} & -\eta \\ & & & -\eta & v_{N-2,\ell} \end{bmatrix} \in \mathbb{R}^{(N-2) \times (N-2)}. \quad (10.12)$$

Meanwhile, on the sub- and super- off-diagonals, we have the block matrices  $-\mu I$ , where  $I$  is the  $(N - 2) \times (N - 2)$  identity matrix. Therefore, for the case of an *arbitrary*  $N \times N$  discretised grid, the  $(N - 2)^2 \times (N - 2)^2$  2D



**Figure 10.3** The structure of  $H$  for  $N = 8$ . Here, green is  $-\mu$ , blue is  $-\eta$ , and red is  $v_{ij}$

discrete Hamiltonian has the form

$$H = \begin{bmatrix} A_1 & -\mu I & & & \\ -\mu I & A_2 & -\mu I & & \\ & \ddots & \ddots & \ddots & \\ & & -\mu I & A_{N-3} & -\mu I \\ & & & -\mu I & A_{N-2} \end{bmatrix} \in \mathbb{R}^{(N-2)^2 \times (N-2)^2}. \quad (10.13)$$

Therefore, like the one-dimensional direct matrix method, we have used the finite-difference method to recast the two-dimensional Schrödinger equation into a matrix-vector eigenvalue equation.

**In summary:** To use the finite-difference direct matrix method to solve the two-dimensional time-independent Schrödinger equation

$$\left(-\frac{\hbar^2}{2m}\nabla^2 + V(x, y)\right)\psi(x, y) = E\psi(x, y),$$

$$\psi(x_0, y) = \psi(x, y_0) = \psi(x_{N-1}, y) = \psi(x, y_{N-1}) = 0,$$

- (1) Discretise the two-dimensional grid into  $N \times N$  discrete points, where  $x_{i+1} - x_i = \Delta x$ ,  $y_{i+1} - y_i = \Delta y$ ,  $V(x_i, y_j) \equiv V_{ij}$ , and  $\psi(x_i, y_j) \equiv \psi_{ij}$ . In the general case of a spherically symmetric potential, it suffices to discretise such that  $\Delta x = \Delta y$ .
- (2) Using this discretisation, construct the natural-ordered  $(N - 2)^2 \times (N - 2)^2$  two-dimensional central finite-difference matrix, as given in Eq. 10.13.
- (3) Using an appropriate eigenvector and eigenvalue algorithm, calculate the eigenvectors  $\psi_k$  and eigenvalues  $E_k$  of the finite-difference matrix.
- (4) Reshape the eigenvectors to form  $(N - 2) \times (N - 2)$  matrices, representing the bound wavefunction solutions to the time independent Schrödinger equation over the interior of the discretised grid. Each wavefunction has associated energy  $E_k$ .

► When determining the grid discretisations for a particular potential  $V(x, y)$ , we want the boundaries of our grid to satisfy the boundary conditions  $\psi(x, y)$ . How do we find an appropriate domain so that this is the case?

Like in the 1D case, we can use the classical turning point to estimate — to solve for states with energy  $\leq E$ , for example, we would choose our grid boundaries such that, on the boundaries,  $V(x_i, y_j) \gg E$ .

### 10.1.3 Degeneracy

Let's take a short break from numerical techniques, and consider a topic of vital importance in quantum mechanics (and, as a result, affects our numerical techniques; but more on that later). When solving the time-independent Schrödinger equation, we sometimes stumble across a phenomenon known as **degeneracy**. This occurs when multiple linearly independent solutions to the Schrödinger equation have the same energy eigenvalue  $E$ ; in these cases, the eigenvectors are referred to as **degenerate states**, while the energy level is referred to as **degenerate**.

Hang on, we've already covered the Schrödinger equation in one-dimensions, and didn't have to deal with this issue — why not? Consider two solutions of the one-dimensional Schrödinger equation  $\psi_1(x)$  and  $\psi_2(x)$ , that share the same energy eigenvalue  $E$ :

$$\begin{aligned}\frac{\hbar^2}{2m} \frac{d^2}{dx^2} \psi_1(x) &= [E - V(x)]\psi_1(x), \\ \frac{\hbar^2}{2m} \frac{d^2}{dx^2} \psi_2(x) &= [E - V(x)]\psi_2(x).\end{aligned}\quad (10.14)$$

Multiplying the first equation by  $\psi_2(x)$ , the second by  $\psi_1(x)$ , and subtracting, gives

$$\psi_2(x) \frac{d^2}{dx^2} \psi_1(x) - \psi_1(x) \frac{d^2}{dx^2} \psi_2(x) = 0. \quad (10.15)$$

Now, we can use a bit of mathematical trickery here, and recognise that this is simply an expanded product rule, with cross terms cancelling out; as a result, we can factor out a derivative:

$$\frac{d}{dx} \left[ \psi_2(x) \frac{d}{dx} \psi_1(x) - \psi_1(x) \frac{d}{dx} \psi_2(x) \right] = 0. \quad (10.16)$$

Thus, integrating both sides of the above equation provides

$$\psi_2(x) \frac{d\psi_1}{dx} - \psi_1(x) \frac{d\psi_2}{dx} = a \quad (10.17)$$

for some constant  $a$ . From the boundary conditions  $\psi(x) \rightarrow 0$  as  $x \rightarrow \pm\infty$ , to satisfy this equation for all values of  $x$ , we must have  $a = 0$ . Rearranging both sides and integrating,

$$\int \frac{1}{\psi_1} d\psi_1 = \int \frac{1}{\psi_2} d\psi_2 \Rightarrow \ln \psi_1 = \ln \psi_2 + c \Rightarrow \psi_1 = e^c \psi_2 \quad (10.18)$$

for some complex constant  $c \in \mathbb{C}$ . Thus, in one-dimension, two wavefunction solutions  $\psi_1(x)$  and  $\psi_2(x)$ , with the same eigenvalue, *cannot* be linearly independent, and therefore do not represent two degenerate states. In fact, this is something you might have already come across in the previous chapter, when solving the one-dimensional quantum harmonic oscillator in problems 9.1 or 9.6. The resulting wavefunctions  $\psi(x)$  may have differed from the exact solution by a complex phase factor  $e^{i\phi}$ . In one dimension, this is not usually much cause for concern, since the quantity we are interested in is the probability, or the absolute value squared:  $|e^{i\phi}\psi(x)|^2 = |\psi(x)|^2$ .

Unfortunately, once we move to higher dimensions, this is no longer the case — degenerate states and energy levels can (and do) exist. For example, we can have two distinct quantum states  $\psi_a(x)$  and  $\psi_b(x)$  that are linearly independent, but both satisfy the Schrödinger equation with the

same eigenvalue  $E$ . As they are linearly independent, we can always perform a unitary change of basis operation, to represent the two eigenstates in a new basis. For example,

$$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & i \\ 1 & -i \end{bmatrix} \begin{bmatrix} \psi_a(x) \\ \psi_b(x) \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} \psi_a + i\psi_b \\ \psi_a - i\psi_b \end{bmatrix}. \quad (10.19)$$

Due to linearity, after the change of basis transformation, the eigenvectors  $\psi_a \pm i\psi_b$  continue to have eigenvalue  $E$ .<sup>1</sup>

Can we choose a *unique* eigenbasis to represent these degenerate states? It turns out that we can. Bound analytical solutions to the time-independent Schrödinger equation are generally distinguished by a set of distinct **quantum numbers**, corresponding to conserved quantities of the system. Once we are already familiar with the energy, and its associated operator  $\hat{H}$  (the Hamiltonian) — the corresponding quantum number  $n$  is referred to as the **principal quantum number**, and distinguishes states with different energy levels. Degenerate states, however, must be distinguished by an additional quantum number — implying the presence of another conserved quantity, for example, momentum, angular momentum, or spin.

From the Schrödinger equation, we know that operators corresponding to conserved observables must **commute** with the Hamiltonian. Consider the z-component angular momentum operator  $\hat{L}_z$ ; if it is conserved in a particular quantum system, then

$$[\hat{H}, \hat{L}_z] = \hat{H}\hat{L}_z - \hat{L}_z\hat{H} = 0. \quad (10.20)$$

As a result, it is easy to see that if  $\psi_n(x)$  is an eigenvector of  $\hat{H}$  with eigenvalue  $E_n$ , then  $\hat{L}_z\psi_n(x)$  is *also* an eigenvector with eigenvalue  $E_n$ :

$$\hat{H}\hat{L}_z\psi_n(x) = \hat{L}_z\hat{H}\psi_n(x) = E_n\hat{L}_z\psi_n(x) \quad (10.21)$$

Thus, the eigenspectrum of  $\hat{L}_z$  and  $\hat{H}$  are related by a similarity transform, and it is always possible to choose a basis which diagonalises both operators — it is this basis we can choose to represent the degenerate states. States with degenerate energy can then be labelled by a quantum number associated with their observed z-component angular momentum  $\hat{L}_z$ .

What does this mean in terms of practicalities? When we use an approach such as the finite-difference direct matrix method discussed above, we are solving for the eigenvectors of a real symmetric matrix — from linear algebra, we know that there always exists a choice of basis where the eigenvectors of a real symmetric matrix are purely real. In fact, numerical eigenvalue algorithms such as the Lanczos method will always return a set of real orthogonal eigenvectors for a real symmetric matrix. This is a by-product of how we constructed our discrete Hamiltonian matrix; by choosing Cartesian coordinates, we are implicitly making a choice of basis for the two-dimensional problem.

<sup>1</sup>To see that this is indeed the case, substitute them back into the Schrödinger's equation and expand it out.

► This generalises to an arbitrary size set of degenerate states. For  $N$  degenerate states with the same eigenvalue  $E$ , we can find a new eigenbasis by transforming them using the complex unitary transformation  $U \in \mathbb{C}^{N \times N}$ , where  $UU^\dagger = I$ .

► For example, the energy eigenstates of the Hydrogen atom can be denoted  $|nlm\rangle$  where  $n$  corresponds to the energy,  $\ell$  the angular momentum magnitude  $\hat{L}^2$ , and  $m$  the  $z$ -component of angular momentum  $\hat{L}_z$ .

This is an eigenbasis for the three operators  $\hat{H}$ ,  $\hat{L}^2$ , and  $\hat{L}_z$ :

$$\hat{H}|nlm\rangle = E_n|nlm\rangle$$

$$\hat{L}^2|nlm\rangle = \ell(\ell+1)\hbar^2|nlm\rangle$$

$$\hat{L}_z|nlm\rangle = m\hbar|nlm\rangle$$

Using completeness of the states, however, it is always possible to choose an eigenbasis where this is not the case. For instance, the energy eigenstate

$$|nlj\rangle = \sum_m \alpha_m(j)|nlm\rangle$$

is still an eigenstate of  $\hat{H}$  and  $\hat{L}^2$ , but no longer of  $\hat{L}_z$ .

- ▶ Recall from the linear algebra chapter that the Lanczos iterative method applied to matrix  $A$  begins by choosing an arbitrary real eigenvector  $\mathbf{q}_0$ , and then performing the matrix multiplication  $\mathbf{q}_i^\dagger A \mathbf{q}_i$  iteratively and orthogonalising.

Since  $\mathbf{q}_0$  is real, if  $A$  is purely real and symmetric, we end up with a real orthogonal set of eigenvectors.

Of course, we could always find a different eigenvector basis by beginning the iteration in the complex vector space  $\mathbf{q}_0 \in \mathbb{C}^N$ .

This isn't anything to worry about — indeed, the results will still represent the solution to the Schrödinger equation for the particular energy value  $E$  — however, keep in mind that degenerate states may differ in form from analytical results if a non-Cartesian basis has been chosen by a unitary transformation.

## 10.2 Basis Diagonalization

In the previous section, we considered the direct finite-difference matrix method. This pulled together a number of different topics that we have considered in Part II of this book, and combined them to solve two-dimensional bound quantum states. Most importantly, these included:

- **The finite difference approximation** (Chap. 5), to reduce the time-independent Schrödinger equation — a partial differential equation with boundary conditions — to a system of coupled linear equations that can be written as a matrix eigenvector equation.
- **Linear algebra and the Lanczos algorithm** (Chap. 7), to solve the eigenvalue equation, and subsequently determine the eigenvalues and eigenvectors of the finite-difference real symmetric matrix.

However, one disadvantage to our finite-difference approach so far is the basis in which we choose to do the discretisation. By choosing Cartesian coordinates, the matrix size grows rapidly with the size of the grid, and it becomes computationally demanding and time consuming to calculate all  $(N - 2)^2$  eigenvalues and eigenvectors. This property also affects the accuracy we can achieve — by reducing the values of  $\Delta x$  and  $\Delta y$  to achieve a more accurate finite-difference approximation, we have to increase  $N$ ; severely limiting the size of the 2D grid we can use.

One solution is to choose a less computationally intensive basis to perform the computation. What do we mean by this? By choosing a set of basis functions that are closer in form to the desired solution, perhaps we can use a smaller subspace of the infinite Hilbert space, yet return a more accurate solution.

### 10.2.1 Orthogonal Basis Diagonalization

Consider a particular quantum system with Hamiltonian  $\hat{H}$  for which we would like to solve the time-independent Schrödinger equation

$$\hat{H}\psi(x) = E\psi(x). \quad (10.22)$$

Rather than discretise the Cartesian coordinates  $x$  and  $y$  and use the finite differences approximation, let's assume that there exists another quantum system, described by Hamiltonian  $\hat{H}_0$ , which is similar (but not identical) to the system we want to solve. Furthermore, and most importantly, the energy eigenstates of this Hamiltonian have analytical solutions  $\phi_n(x)$ ,  $n = 0, 1, 2, \dots$ , or  $|\phi_n\rangle$  in Dirac notation.

As the eigenstates of an Hermitian operator form an orthonormal basis over the infinite Hilbert space, we can express any other state as a linear

► In this section, we will start using some Dirac notation for notational simplicity:

$$\langle \phi | \psi \rangle = \int_{-\infty}^{\infty} \phi(x)^* \psi(x) dx$$

$$\langle \phi | \hat{O} | \psi \rangle = \int_{-\infty}^{\infty} \phi(x)^* \hat{O} \psi(x) dx$$

combination of these eigenstates — including the state we would like to calculate:

$$|\psi\rangle = \sum_{n=0}^{\infty} c_n |\phi_n\rangle \quad (10.23)$$

where the set of complex coefficients,  $c_n \in \mathbb{C}$ , uniquely characterise each solution  $|\psi\rangle$ . If we were to substitute this into the unsolved Schrödinger equation Eq. 10.22, we find

$$\hat{H} \sum_{n=0}^{\infty} c_n |\phi_n\rangle = E \sum_{n=0}^{\infty} c_n |\phi_n\rangle, \quad (10.24)$$

and contracting both sides with  $\langle\phi_m|$ ,

$$\sum_{n=0}^{\infty} c_n \langle\phi_m|\hat{H}|\phi_n\rangle = E \sum_{n=0}^{\infty} c_n \langle\phi_m|\phi_n\rangle = E \sum_{n=0}^{\infty} c_n \delta_{mn} = Ec_m. \quad (10.25)$$

By defining the matrix of inner products,

$$H_{mn} = \langle\phi_m|\hat{H}|\phi_n\rangle = \int_{-\infty}^{\infty} \phi_m(x)^* \hat{H} \phi_n(x) dx$$

(10.26)

we can write this as a matrix equation:

$$\sum_{n=0}^{\infty} H_{mn} c_n = Ec_m \Leftrightarrow H\mathbf{c} = E\mathbf{c}.$$

(10.27)

Therefore, in this infinite orthonormal basis, the solution to the Schrödinger equation has been reduced to finding the basis expansion coefficients  $\mathbf{c}$ , corresponding to the eigenvectors of the matrix of inner products  $H_{mn}$ .

However, as we know, we cannot easily compute infinite quantities numerically; instead, we truncate the summation to a specific size  $N$ , providing an approximation of our solution:

$$\sum_{n=0}^{N-1} H_{mn} c_n \approx Ec_m \Leftrightarrow H\mathbf{c} = E\mathbf{c}.$$

(10.28)

$H \in \mathbb{C}^{N \times N}$  is now an  $N \times N$  complex Hermitian matrix, and  $\mathbf{c} \in \mathbb{C}^N$  a complex length- $N$  vector. Note that, while a suitable choice of orthogonal basis functions can provide solutions with high levels of accuracy using a reasonably small matrix size  $N$ , the matrix elements  $H_{mn}$  are now significantly harder to calculate than previously; in  $D$  dimensions, each element is now calculated using a  $D$ -dimensional integral.

Let's begin with a quick one-dimensional example to get a feel of basis diagonalisation.

### Example 10.3 One-dimensional quantum harmonic oscillator

Use basis diagonalisation to numerically determine the energy eigenstates and eigenvalues of the one-dimensional quantum harmonic oscillator  $V(x) = x^2/2$ .

**Solution:** Before we begin, we need to decide on a useful basis set for the basis diagonalisation — the best approach is to consider the symmetry and behaviour of the potential, and consider similar systems and solutions. We can crudely approximate the harmonic potential for  $E \leq 5$  as an infinite square well potential (see Fig. 10.4). We know that, in the case of an infinite potential well, the wavefunction must necessarily be zero beyond the walls. Due to this periodic boundary condition, all solutions to the infinite-potential Schrödinger equation  $\psi^\infty(x)$  can therefore be written in terms of a Fourier series expansion:

$$\psi^\infty(x) = \frac{1}{\sqrt{L}} \sum_{n=-\infty}^{\infty} c_n e^{2\pi i n x / L}, \quad (10.29)$$

where  $L$  is the length of the potential. Thus, the orthogonal basis functions  $\phi_n(x) = e^{2\pi i n x / L} / \sqrt{L}$  seem like a good fit for the quantum harmonic oscillator potential.

Let's begin by calculating the  $H$  matrix elements:

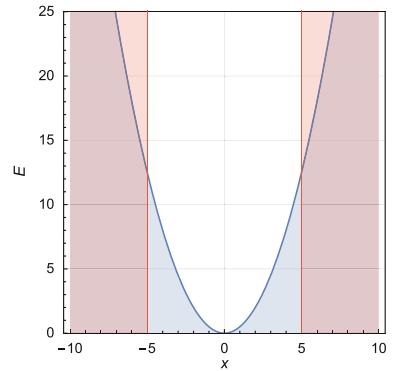
$$\begin{aligned} H_{mn} &= \langle \phi_m | \hat{H} | \phi_n \rangle = \frac{1}{L} \int_{-L/2}^{L/2} e^{-2\pi i m x / L} \hat{H} e^{2\pi i n x / L} dx \\ &= \frac{1}{L} \int_{-L/2}^{L/2} e^{-2\pi i m x / L} \left( -\frac{\hbar^2}{2m_e} \frac{d^2}{dx^2} + \frac{1}{2} x^2 \right) e^{2\pi i n x / L} dx \\ &= \frac{1}{L} \int_{-L/2}^{L/2} e^{-2\pi i m x / L} \left( \frac{2n^2 \hbar^2}{L^2 m_e} + \frac{1}{2} x^2 \right) e^{2\pi i n x / L} dx \\ &= \left( \frac{n^2 \hbar^2}{L^2 m_e} + \frac{L^2}{4\pi^2} \right) \delta_{mn} + \frac{L^2 (-1)^{m+n}}{4\pi^2 (m-n)^2} (1 - \delta_{mn}). \end{aligned}$$

Here,  $\delta_{mn}$  is the Kronecker delta, and indicates the diagonal terms of the matrix, as opposed to the off-diagonal terms ( $1 - \delta_{mn}$ ).

If we now construct the matrix for the case  $L = 10$ , with basis elements  $-5 \leq n, m \leq 5$  (producing an  $11 \times 11$  matrix), and use an eigenvalue algorithm like the Lanczos iteration, we get the following eigenvalues:

$$E = \{0.500005, 1.50012, 2.50133, 3.50907, 4.54128, 5.63491, 6.83562, \dots\}$$

Comparing the eigenvalues  $E \leq 5$  to the analytical solution  $E_n = n + 1/2$ , it doesn't look bad considering we only used an  $11 \times 11$  matrix! In comparison to problem 9.7, we used an  $1000 \times 1000$  finite difference matrix. To increase the accuracy further, we could increase the number of basis functions in our truncated matrix, or find an even better set of basis functions for the quantum harmonic oscillator. However, as you can see, there is now an additional computational load in calculating the matrix elements.



**Figure 10.4** Quantum harmonic potential (blue) overlaid with an infinite square well potential

- ▶ The basis functions  $\phi(x)$  are orthogonal over  $-L/2 \leq x \leq L/2$ , **not** the entire domain. Since the quantum harmonic oscillator has valid solutions  $-\infty \leq x \leq \infty$ , this may produce numerical error for higher energy eigenstates.
- ▶ It is important to note here that we are able to analytically calculate the integral for the elements of  $H_{mn}$ , however, if our basis functions are only known numerically, **or** the integral has no analytical form, we may have no choice but to perform the integral numerically.

### 10.2.2 The Plane-Wave Basis

You may have noticed in the previous example that, using the complex exponential as our set of orthogonal basis functions, results in an integral for  $H_{mn}$  that looks curiously like a Fourier transform. This basis set is referred to as the **plane-wave basis**, as the basis wavefunctions resemble plane-waves travelling in free space. In fact, if you repeat the inner-product calculation with an arbitrary potential  $V(x)$ , you find

$$\begin{aligned} H_{mn} &= \frac{1}{L} \int_{-L/2}^{L/2} e^{-2\pi imx/L} \left( -\frac{\hbar^2}{2m_e} \frac{d^2}{dx^2} + V(x) \right) e^{2\pi inx/L} dx \\ &= \frac{n^2 \hbar^2}{L^2 m_e} \delta_{mn} + \frac{1}{\sqrt{L}} \tilde{V}(2\pi(m-n)/L). \end{aligned}$$

That is, the matrix elements in this basis are composed of a potential-independent diagonal component, as well as the Fourier transform of the potential itself. This is especially useful if the Fourier transform of the potential is such that  $\tilde{V} \approx 0$  for  $|m-n| \geq C$ ; if  $C$  is relatively small, then we can expect a high degree of accuracy for a reasonably small basis truncation.

Note that, while we have considered the case of a one-dimensional plane-wave basis diagonalisation here, the plane-wave basis can also be expanded to arbitrary higher dimension  $D$  as follows:

$$|k_1, k_2, \dots, k_D\rangle = \phi_{\mathbf{k}}(\mathbf{x}) = \frac{1}{\sqrt{L^D}} e^{i\mathbf{k}\cdot\mathbf{x}} \quad (10.30)$$

where  $\mathbf{k} = (k_1, k_2, \dots, k_D)$  and  $\mathbf{x} = (x_1, x_2, \dots, x_D)$ .

### 10.2.3 Numerical Integration

More often than not, the problem you will need to solve will involve (a) numerical orthogonal basis states  $\phi_n \equiv \phi(x_n)$ , and/or (b) numerical potentials. In such a case, the integrations to determine the matrix elements of  $H_{mn}$  will have to be undertaken numerically, *including* the application of the differential operator  $\frac{d^2}{dx^2} \phi(x)$  within the integrand.

How this is performed is up to you, and may depend on the problem under study — a common approach is to apply the method of finite differences to fourth order,

$$\frac{d^2}{dx^2} \phi(x_n) \approx \frac{1}{12\Delta x^2} [16(\phi_{n+1} + \phi_{n-1}) - 30\phi_n - (\phi_{n+2} + \phi_{n-2})], \quad (10.31)$$

and then using Simpson's integration rule to calculate the integral. Both of these approaches generalise to higher dimensions — simply perform the finite-difference approximation multiple times for each spatial variable  $x_i$  and sum the results. However, in cases where the dimensionality becomes a problem, Fourier differentiation techniques and Monte-Carlo integration are alternate approaches.

**In summary:** To use the orthogonal basis diagonalisation method to find the bound states and energies of the time-independent Schrödinger equation

$$\hat{H}\psi(\mathbf{x}) = E\psi(\mathbf{x})$$

where  $\hat{H} = -\frac{\hbar^2}{2m}\nabla^2 + V(\mathbf{x})$ ,

- (1) Choose a set of orthogonal basis functions  $|\phi_n\rangle = \phi_n(\mathbf{x})$ ,  $\langle\phi_m|\phi_n\rangle = \delta_{mn}$ , that approximately satisfy the potential under consideration. This can be done by taking into account the symmetry of the potential, boundary conditions, and known solutions of similar systems.
- (2) Truncate the basis set to size  $N$ , and compute the Hamiltonian matrix in this truncated basis:

$$H_{mn} = \langle\phi_m|\hat{H}|\phi_n\rangle = \int_{-\infty}^{\infty} \phi_m(x)^* \hat{H} \phi_n(x) dx$$

For all allowed values of  $m$  and  $n$ :

- (a) First, calculate  $\phi_m(x)^* \hat{H} \phi_n(x)$  using numerical differentiation techniques.
- (b) Then, compute the integral using numerical integration, such as Simpson's rule.
- (3) Once the matrix is computed, solve the eigenvalue equation  $H\mathbf{c} = E\mathbf{c}$  using the Lanczos algorithm (or LAPACK/SciPy).
  - (a) The eigenvalues  $E$  correspond to the eigenstate energies of the Hamiltonian  $\hat{H}$ .
  - (b) The energy eigenstates are uniquely determined by the eigenvectors, which provide the basis coefficients of the eigenstates:

$$\psi(\mathbf{x}) = \sum_j c_j \phi_j(\mathbf{x}).$$

### 10.3 The Variational Principle

In the previous section we saw that basis diagonalisation, by diagonalising the Hamiltonian in a basis more closely approximating the actual eigenstates, allows us to reduce numerical error while simultaneously decreasing computational load. Substituting the infinite basis expansion into the Schrödinger equation, we arrive at another eigenvalue equation that we can use to solve the system. When we truncated the basis expansion, we assumed that this eigenvalue equation would continue to provide the solutions to the Hamiltonian. But how do we know for sure that this is the case?

It turns out that orthogonal basis diagonalisation is a special case of what is known as **the variational principle**.

#### The variational principle

**Theorem 10.1.** *Consider a system with Hamiltonian  $\hat{H}$ , as well as an arbitrary (possibly non-orthogonal) trial state  $|\psi\rangle$ . The energy expectation value of this state is given by*

$$E = \langle \hat{H} \rangle = \frac{\langle \psi | \hat{H} | \psi \rangle}{\langle \psi | \psi \rangle}. \quad (10.32)$$

*The variational principle states that:*

- $E \geq E_0$ , the ground state energy of the Hamiltonian  $\hat{H}$ ,
- If  $E = E_0$ , then  $|\psi\rangle = |\psi_0\rangle$ , the ground state of the Hamiltonian.

In other words, we can find the ground state of the Hamiltonian by choosing a trial wavefunction, calculating the energy expectation, and varying the wavefunction such that we minimise the expected energy. To see that the expected energy value is always larger than the ground state energy, we can expand the energy expectation expression above in terms of the orthogonal eigenstates  $|\phi_n\rangle$  of  $\hat{H}$ ,

$$|\psi\rangle = \sum_n c_n |\phi_n\rangle, \quad \langle \phi_m | \phi_n \rangle = \delta_{nm} \quad (10.33)$$

for  $c_n \in \mathbb{C}$ . This gives

$$\begin{aligned} E &= \frac{\langle \psi | \hat{H} | \psi \rangle}{\langle \psi | \psi \rangle} = \frac{\sum_n \sum_m c_m^* c_n \langle \phi_m | \hat{H} | \phi_n \rangle}{\sum_n |c_n|^2} = \frac{\sum_n \sum_m c_m^* c_n E_n \langle \phi_m | \phi_n \rangle}{\sum_n |c_n|^2} \\ &= \frac{\sum_n \sum_m c_m^* c_n E_n \delta_{mn}}{\sum_n |c_n|^2} \\ &= \frac{\sum_n |c_n|^2 E_n}{\sum_n |c_n|^2}. \end{aligned} \quad (10.34)$$

Since this sum includes the ground state and ground state energy, we can take it out of the summation:

$$\frac{\langle \psi | \hat{H} | \psi \rangle}{\langle \psi | \psi \rangle} = E_0 + \frac{\sum_n |c_n|^2 (E_n - E_0)}{\sum_n |c_n|^2}. \quad (10.35)$$

Now, the quantity  $E_n - E_0 \leq 0$  for all  $n \leq 0$ , since, by definition, the excited states have energy greater than the ground state. As a result, we can see that the expected energy value of the trial wavefunction can never be less than the ground state energy.

How do we know that the trial wavefunction, with expected energy equal to the ground state energy, *must* necessarily be the ground state wavefunction? Intuitively, it makes sense; for trial wavefunctions close to the ground state, we are very close to a stationary state, so small perturbations of the trial wavefunction should lead to vanishingly small perturbations of energy. We can use this to construct a proof of the second part of the variational principle.

### Formal Proof

Consider a trial wavefunction,  $|\psi\rangle$ , that is perturbed slightly by an infinitesimal amount  $|\delta\psi\rangle$ . This causes a corresponding infinitesimal change in the value of the expected energy  $E + \delta E$ :

$$E + \delta E = \frac{(\langle \psi | + \langle \delta\psi |) \hat{H} (|\psi\rangle + |\delta\psi\rangle)}{(\langle \psi | + \langle \delta\psi |)(|\psi\rangle + |\delta\psi\rangle)}. \quad (10.36)$$

Expanding the right-hand side, and neglecting all terms of order  $\mathcal{O}(\delta\psi^2)$  (such that  $\langle \delta\psi | \hat{H} | \delta\psi \rangle \approx 0$ ), gives

$$\begin{aligned} E + \delta E &= \frac{\langle \psi | \hat{H} | \psi \rangle + \langle \delta\psi | \hat{H} | \psi \rangle + \langle \psi | \hat{H} | \delta\psi \rangle}{\langle \psi | \psi \rangle + \langle \psi | \delta\psi \rangle + \langle \delta\psi | \psi \rangle} \\ &= \frac{\langle \psi | \hat{H} | \psi \rangle + \langle \delta\psi | \hat{H} | \psi \rangle + \langle \psi | \hat{H} | \delta\psi \rangle}{\langle \psi | \psi \rangle} \left( \frac{\langle \psi | \psi \rangle}{\langle \psi | \psi \rangle + \langle \psi | \delta\psi \rangle + \langle \delta\psi | \psi \rangle} \right). \end{aligned}$$

where we have factored out the bracketed term. The bracketed term has the form  $x/(x + \delta x)$ ; for  $\delta x \ll 1$ , we can approximate this to first order using the Taylor series expansion  $x/(x + \delta x) = 1 - \delta x/x + \mathcal{O}(\delta x^2)$ . Applying this to the bracketed term,

$$E + \delta E = \left( E + \frac{\langle \delta\psi | \hat{H} | \psi \rangle + \langle \psi | \hat{H} | \delta\psi \rangle}{\langle \psi | \psi \rangle} \right) \left( 1 - \frac{\langle \psi | \delta\psi \rangle + \langle \delta\psi | \psi \rangle}{\langle \psi | \psi \rangle} \right).$$

Expanding and rearranging, and neglecting terms of order  $\delta\psi^2$  and higher, gives us an expression for  $\Delta E$ :

$$\delta E = \frac{\langle \psi | \hat{H} | \delta\psi \rangle + \langle \delta\psi | \hat{H} | \psi \rangle}{\langle \psi | \psi \rangle} - E \frac{\langle \psi | \delta\psi \rangle + \langle \delta\psi | \psi \rangle}{\langle \psi | \psi \rangle}$$

► This can be shown by writing  $\langle a | \hat{O} | b \rangle$  as an integral over the Hilbert space, and taking the conjugate.

To be in the local region of a stationary state, we require  $\Delta E$  to be zero; this can be done by solving for the numerators to vanish. However, there is an important trick to simplifying this expression; we know from the definition of the inner-product that  $\langle a|\hat{O}|b\rangle$  and  $\langle b|\hat{O}|a\rangle$  are simply *conjugates* of each other — comparing the terms, we see we simply have the same expression which must be satisfied independently by each conjugate term. Thus, for  $\delta E = 0$ , we require

$$\langle \delta\psi | \hat{H} | \psi \rangle - E \langle \delta\psi | \psi \rangle = 0 \Leftrightarrow \langle \delta\psi | (\hat{H} - E) | \psi \rangle = 0$$

This expression is satisfied if and only if the trial wavefunction is an eigenvector of  $\hat{H}$  of energy  $E$ . Therefore, if the trial wavefunction  $|\psi\rangle$  is such that small perturbations do not cause small perturbations in the expected energy — that is, we are in a **local minimum** or **stationary point** for  $E$  in terms of  $\delta\psi$  — then  $|\psi\rangle$  satisfies the time-independent Schrödinger equation and is an eigenstate of  $\hat{H}$ .

The variational principle therefore becomes a minimisation problem — by choosing a trial wavefunction  $|\psi(\alpha)\rangle$  parameterised by the parameters  $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_N)$ , we calculate the energy expectation

$$E(\alpha) = \frac{\langle \psi(\alpha) | \hat{H} | \psi(\alpha) \rangle}{\langle \psi(\alpha) | \psi(\alpha) \rangle} \quad (10.37)$$

and then determine the parameters that **minimise** the expected energy; that is, the parameter values such that

$$\frac{\partial}{\partial \alpha_n} E(\alpha) = 0 \quad (10.38)$$

- ▶ By expanding the trial wavefunction in terms of orthogonal basis functions, and using the Lanczos algorithm to determine energy eigenvalues and eigenstates, we recover the basis diagonalisation method from the previous section — substitute this in for yourself and have a go!

for all parameters  $\alpha_n$ . This can be performed analytically, by choosing a trial wavefunction similar in form to the expected solution, performing the inner-product and differentiating, or numerically, by expanding the trial wavefunction in an orthogonal eigenbasis  $|\psi(\alpha)\rangle = \sum_n \alpha_n |\phi_n\rangle$  and using a numerical optimization technique to determine  $\alpha$ .

**In summary:** To use the variational principle to calculate the ground state energy and wavefunction for a system described by Hamiltonian  $\hat{H}$ :

- (1) Choose a trial wavefunction parametrised by  $\alpha_0$ ; either a parameterised analytical expression, or a numerical expansion in terms of basis functions.

(2) Calculate the energy expectation value

$$E = \frac{\langle \psi | \hat{H} | \psi \rangle}{\langle \psi | \psi \rangle} = \frac{\int \psi(x)^* \hat{H} \psi(x) dx}{\int \psi(x)^* \psi(x) dx}.$$

This can be performed numerically on a discretised grid by approximating  $\hat{H}$  using finite-difference techniques or Fourier differentiation, and then integrating using Simpson's method or Monte-Carlo methods or some other numerical integration method.

- (3) Apply an optimization algorithm to determine a new set of parameters  $\alpha_1$  that reduce the energy expectation value  $E$  of the trial wavefunction.
- (4) Repeat the optimization or minimisation process until the parameters or energy have converged such that  $|\alpha_{n+1} - \alpha_n| \leq \epsilon$  or  $|E_{n+1} - E_n| \leq \epsilon$  for some  $\epsilon \ll 1$ .

### Applications of the variational principle

The variational principle is very powerful, which can be used in conjunction with a huge array of numerical techniques. For example, it is often paired with Monte-Carlo techniques in order to more efficiently compute integrals in high-dimensional spaces.

In addition, it is also widely used in quantum computational chemistry, to calculate the ground states of atomic and molecular orbits. Note that in the above discussion, we have assumed throughout that the trial wavefunction  $|\psi\rangle$  may not be normalised. As a consequence, we could modify the derivations to take into account a **non-orthogonal basis diagonalisation**; that is, a basis set  $|\phi_n\rangle$  such that

$$\langle \phi_m | \phi_n \rangle = \int_{-\infty}^{\infty} \phi_m(x)^* \phi_n(x) dx = S_{mn} \neq \delta_{mn},$$

where  $S_{mn}$  is referred to as the **overlap matrix**, and we'd see that the variational principle continues to hold.

For example, consider the non-orthogonal basis expansion

$$|\psi\rangle = \sum_n c_n |\phi_n\rangle, \quad \langle \psi | \psi \rangle = \sum_m \sum_n c_m^* c_n S_{mn}.$$

Substituting this into the formal proof for the variational principle, we find that energy is minimised if the following expression is satisfied:

$$H\mathbf{c} = E\mathbf{S}\mathbf{c}, \quad \text{where } H_{nm} = \langle \phi_m | \hat{H} | \phi_n \rangle.$$

- ▶ In quantum chemistry, Gaussian distributions are a common non-orthogonal basis set that used to find the ground state of orbitals.
- ▶ The generalised eigenvalue equation can be solved in SciPy using `scipy.linalg.eigh(H, S)`, or in LAPACK via the ZHEGV subroutine.

This is known as a **generalised eigenvalue equation**, as it can be rearranged into the form  $(H - ES)\mathbf{c} = 0$ , which would be the standard eigenvalue equation if we replaced  $S$  with the identity. Thus, in cases where non-orthogonal basis sets more accurately model the system under investigation, solving the generalised eigenvalue equation of  $H$  with respect to the overlap matrix provides numerical solutions to the bound energies and states.

#### Further reading

When it comes to solving the time-independent Schrödinger equation in higher dimensions, numerical techniques tend to become a lot more specialised, making use of the symmetry and underlying properties of the system. A fantastic graduate level textbook, that assumes a familiarity with numerical techniques and programming, is Computational Physics by Thijssen, which focuses on applications of numerical methods to quantum systems. This text is a great resource for more details on the variational method, basis diagonalisation, and Monte Carlo methods to solving the Schrödinger equation.

- Thijssen, J. M. (2007). Computational physics (2nd ed). Cambridge, UK; New York: Cambridge University Press. ISBN 978-0-521-54106-0

## Exercises

### P10.1 Finite-difference matrix construction (Fortran)

Write a function or subroutine that accepts an integer  $N$ , as well as the values  $a$  and  $b$ , and returns the  $(N - 2)^2 \times (N - 2)^2$  finite difference matrix representing the free space ( $V = 0$ ) Hamiltonian  $\hat{H} = -\frac{\hbar}{2m}\nabla^2$  on the discretised grid  $a \leq x, y \leq b$ .

*Hint:* In the construction of the finite difference matrix, a naïve approach is to use two `do` loops to calculate all  $(N - 2)^4$  elements in the matrix. A better approach is to take the symmetry into account, and instead perform one loop down the main diagonal of  $(N - 2)^2$  elements, and setting the off-diagonal elements at the same time due to their (known) placement.

### P10.2 Finite-difference matrix construction (Python)

There are several ways we can approach the construction of the free space ( $V = 0$ ) finite difference matrix using NumPy and SciPy. For each of the following approaches, write a function that accepts an integer  $N$ , as well as the values  $a$  and  $b$ , and returns the  $(N - 2)^2 \times (N - 2)^2$  finite difference matrix representing the free space ( $V = 0$ ) Hamiltonian  $\hat{H} = -\frac{\hbar}{2m}\nabla^2$  on the discretised grid  $a \leq x, y \leq b$ .

- (a) Use list compression and filters to construct lists representing the main diagonal, two off-diagonals, as well as the  $\pm(N - 2)$  diagonals. Then, convert them into diagonal matrices using `np.diag`, and add them together.
- (b) Alternatively, there is a well known relationship connecting the one-dimensional finite difference matrix and the two dimensional finite difference matrix:

$$H_{2D} = H_{1Dx} \otimes I_{N-2} + I_{N-2} \otimes H_{1Dy}.$$

Here,  $I_{N-2}$  is the  $(N - 2) \times (N - 2)$  identity matrix,  $H_{1D}$  is the  $(N - 2) \times (N - 2)$  one-dimensional finite difference matrix (see Eq. 9.64), and  $\otimes$  represents the **Kronecker product**.

In this approach, first construct the one-dimensional finite difference matrix, and then use `scipy.linalg.kron` and `np.identity` to return the two-dimensional finite difference matrix.

Time these two approaches for various values of  $N$ . Which approach seems more efficient?

You may have noticed that a majority of the matrix elements are zero; as a result, we can take advantage of SciPy's **sparse matrix module**. In sparse representation, only the non-zero elements and their positions are stored in memory, leading to more efficient memory usage.

- (c) Construct the 2D finite difference matrix using the `scipy.sparse.diags` function. This function is of the form  
`scipy.sparse.diags(diagonals, offsets)`  
 where `diagonals` is a nested list of diagonal elements, and `offsets` is a list of integer offsets, with 0 corresponding to the main diagonal,  $k > 0$  the  $k$ th upper diagonal, and  $k < 0$  the  $k$ th lower diagonal.
- (d) To retain the advantage of using sparse arrays, we need to use functions within the `scipy.sparse` module. Use the function `scipy.sparse.linalg.eigsh` to calculate the eigenvalues and eigenvectors of the matrix for various values of  $N$ .

### P10.3 The 2D quantum harmonic oscillator

Consider the two-dimensional quantum harmonic oscillator with potential function

$$V(x, y) = \frac{1}{2}m_e\omega^2(x^2 + y^2).$$

- (a) For  $\hbar = 1$ ,  $m_e = 1$ , and  $\omega = 1$ , construct the finite difference matrix approximation to the Hamiltonian for  $x, y \in [-5, 5]$  and  $N = N_x = N_y = 51$  (i.e.  $\Delta x = \Delta y = 0.2$ ). Take advantage of the fact that the matrix  $H$  will be Hermitian and displays banded structure.
- (b) Using LAPACK or SciPy, calculate the eigenvalues and eigenvectors of  $H$ .
  - (i) Compare the 10 smallest eigenvalues to the theoretical energy levels

$$E_{n_x n_y} = \hbar\omega(n_x + n_y + 1).$$

Note that the 2D harmonic oscillator has degenerate eigenvalues; for example  $E_{1,0} = E_{0,1} = 2\hbar\omega$ .

- (ii) Plot the potential function  $V(x, y)$ . Notice that the contours (trajectories of equal value) are circles centered on the origin.
- (c) Reshape the eigenvectors corresponding to eigenvalues  $E_0 = 1$  to form a  $(N - 2) \times (N - 2)$  square array, and plot the results in 3D. This is the ground state of the quantum harmonic oscillator.

► For simplicity, we will use **atomic units** for problem sets involving numerical solutions to the Schrödinger equation:

$$m_e = q_e = \hbar = 1,$$

where  $m_e$  is the electron mass and  $q_e$  is the electron charge.

► Remember, the discretised Hamiltonian will be of dimension  $(N - 2)^2 \times (N - 2)^2$ .

► Using the Cartesian basis, the eigenvalues of the 2D quantum harmonic oscillator are simply the **sum** of two 1D quantum harmonic oscillators.

(d) Compare your result with the analytical solution

$$\psi_0(x, y) = \frac{1}{2\pi} e^{-(x^2+y^2)/2}.$$

What is the magnitude of the error?

#### P10.4 Degeneracy

As we saw in problem 10.3, the two dimensional quantum harmonic oscillator displays degeneracy — there were multiple eigenvectors with the same energy eigenvalue.

(a) Using your results from 10.3, calculate the fractional error

$$\text{error} = \left| \frac{E_{n_x n_y} - \hbar\omega(n_x + n_y + 1)}{\hbar\omega(n_x + n_y + 1)} \right|,$$

and plot the error against the energy eigenvalue  $E_{n_x n_y}$  for all eigenvalues  $\leq 6$ .

You should notice something interesting; for each degenerate energy value  $E_{n_x n_y}$ , the fractional error separates the degenerate states into distinct groups, with each group containing no more than two states due to the Pauli exclusion principle. In fact, this distinguishes degenerate states with different values of angular momentum.

In polar coordinates  $(r, \phi)$ , where  $r^2 = x^2 + y^2$  and  $\tan \phi = y/x$ , the analytical solution for the energy eigenstates of the two-dimensional quantum harmonic oscillator is given by

$$\psi_{n,m}(r, \phi) = \frac{1}{2\pi} r^{|m|} e^{-\hbar\omega r^2/2} \sqrt{\frac{2n!(\hbar\omega)^{|m|+1}}{(|m|+n)!}} L_n^{|m|}(\hbar\omega r^2) e^{im\phi},$$

where  $L_n^m$  are the **generalised Laguerre polynomials**, and we can choose any  $n$  and  $m$  such that  $2n + |m| = 0, 1, 2, 3, \dots$ . This satisfies the Schrödinger equation  $\hat{H}\psi_{n,m}(r, \phi) = E_{nm}\psi_{n,m}(r, \phi)$  with energy eigenvalues

$$E_{nm} = \hbar\omega(2n + |m| + 1).$$

(b) Reshape the two eigenvectors corresponding to eigenvalues  $E = 2$  to form a  $(N-2) \times (N-2)$  square array, and plot the results in 3D. Compare this to plots of the analytical solution — what do you notice?

The energy eigenstates in the polar coordinates are complex, and are simultaneously eigenstates of the  $z$ -component angular momentum operator:

$$\hat{L}_z\psi_{n,m}(r, \phi) = -i\hbar \frac{\partial}{\partial \phi} \psi_{n,m}(r, \phi) = \hbar m \psi_{n,m}(r, \phi).$$

► The generalised Laguerre polynomials are available in SciPy as `scipy.special.genlaguerre`. For use in Fortran, note that  $L_0^1(x) = 1$ .

► Although we are working on a two-dimensional problem, we can view the angular momentum by embedding the system as a 2D  $xy$  plane in three dimensions. The resulting angular momentum vector will be solely in the  $z$ -direction.

- (c) Determine the correct linear combination of the two degenerate  $E = 2$  Cartesian basis eigenstates in order to recover the polar-basis eigenstates corresponding to  $E = 2$ . These states will now be simultaneously eigenstates of the Hamiltonian and the angular momentum.

**Hint:** note that the polar eigenstates are complex and radially symmetric, while the Cartesian energy eigenstates are purely real.

► The inverse Gaussian potential Schrödinger equation has no analytical solution to compare against, however the numerical error should be around the same order of magnitude as for the quantum harmonic potential.

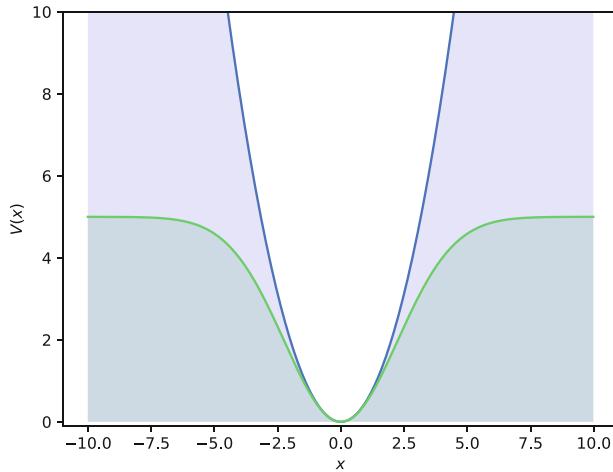
### P10.5 The inverse Gaussian potential

The inverse Gaussian potential in two dimensions is given by

$$V(x, y) = V_0(1 - e^{-\hbar\omega(x^2+y^2)/2V_0}).$$

- (a) Plot the inverse Gaussian potential and the quantum harmonic potential. How do they compare? What is the distinguishing feature of the inverse Gaussian potential, and does this result in different quantum dynamics?
- (b) Let  $V_0 = 5$ . For what range of energies do you expect bound states satisfying the time-independent Schrödinger equation to exist?
- (c) Construct the finite-difference matrix approximation to the Hamiltonian of a particle confined to the inverse Gaussian potential in two dimensions, using  $N = 51$  ( $\Delta x = \Delta y = 0.2$ ),  $-5 \leq x, y \leq 5$ . Using an eigenvalue and eigenvector solver, find the 20 lowest energy eigenvalues.
- (d) Consider the cross-section  $y = 0$ ; plot the quantum harmonic oscillator potential  $V_{QHO}(x, 0)$  and inverse Gaussian potential  $V_{Gaussian}(x, 0)$  along the  $x$  axis. On this plot, overlay the energy eigenvalues for the quantum harmonic oscillators and the inverse Gaussian potential.
- (i) How do the energy eigenvalues compare between the two potentials?
  - (ii) Does the inverse Gaussian potential produce degenerate energy eigenstates?
  - (iii) As  $E \rightarrow 5$ , what do you observe about the energy eigenstates of the inverse Gaussian potential?
- (e) Use Simpson's method to perform the integral  $\iint |\psi(x, y)|^2 dx dy$  of one of the computed eigenstates. Is the state normalised? Comment on your observation.
- Hint:* Recall our discussion on Krylov subspace techniques (Sect. 7.3), used by both LAPACK and SciPy when computing eigenvalues.

**P10.6** Plotting the one-dimensional inverse Gaussian potential  $V(x) = V_0(1 - e^{-\hbar\omega x^2/2V_0})$  for  $V_0 = 5$  alongside the quantum harmonic potential  $\frac{1}{2}m_e\omega^2x^2$ , we can see that for some values of energy, they appear quite similar:



As such, the bound eigenstates of the 2D quantum harmonic oscillator seem like a good choice for basis diagonalisation of the 2D inverse Gaussian Hamiltonian,

$$\hat{H} = -\frac{\hbar^2}{2m_e} \left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) + V_0(1 - e^{-\hbar\omega(x^2+y^2)/2V_0}).$$

- (a) Using the classical turning point for the *quantum harmonic oscillator*, estimate the radius of the classically forbidden potential barrier as a function of  $E$ .  
Using this as a guide, approximately how many eigenstates of the harmonic oscillator can we consider on a discretised grid of size  $-11 \leq x, y \leq 11$ , such that they satisfy the boundary condition  $\phi \approx 0$  at the grid boundaries?
- (b) The exact solution to the 2D quantum harmonic oscillator in **Cartesian coordinates** and using atomic units is given by

$$\phi_E(x, y) = \frac{1}{\sqrt{2^{n+m} n! m! \pi}} e^{-(x^2+y^2)/2} H_n(x) H_m(y),$$

where  $E = n + m + 1/2$ . Choosing the first 50 quantum harmonic eigenstates, use the fourth order finite difference second-derivative approximation to approximate the expression  $\phi_i(x)^* \hat{H} \phi_j(x)$  for all values of  $i$  and  $j$  on the discrete grid.

- (c) Thus, using the above result, use Simpson's integration method to diagonalise the inverse Gaussian Hamiltonian in the harmonic oscillator basis. That is, determine the matrix elements

$$H_{ij} = \langle \phi_j | \hat{H} | \phi_k \rangle.$$

- (d) Using an eigenvalue solving algorithm, find the eigenvalues of  $H_{ij}$ . How do they compare to the eigenvalues calculated using the direct finite-difference matrix method?
- (e) For the 2D quantum harmonic oscillator, it can be shown from studying the degeneracies that the number of states with energy  $E \leq E_{max}$  for some  $E_{max}$  is given by

$$N = \sum_{E=1}^{E_{max}} E = \frac{1}{2} E_{max}(E_{max} + 1).$$

If we wanted to use the quantum harmonic basis to find *all* energy eigenvalues of the inverse Gaussian potential  $E \leq 10$  within reasonable accuracy, what is the minimum number of basis functions we would need? How about  $E \leq 15$ ?  $E \leq 50$ ?

- P10.7** Using the plane wave basis in one-dimension, find the bound states of the potential

$$V(x) = \begin{cases} \infty, & x \leq 1 \\ e^{-100x^2}, & -1 < x < 1 \\ \infty, & x \geq 1 \end{cases}.$$

This is an infinite square potential well, containing a finite potential barrier.

### P10.8 The variational principle

Consider the two-dimensional Pöschl–Teller potential which has the Hamiltonian

$$\hat{H} = -\frac{\hbar^2}{2m_e} \left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) - \frac{1}{\cosh^2(\sqrt{x^2 + y^2})}.$$

- (a) Plot the Pöschl–Teller potential. What do you observe?
- (b) Based on your plot, choose an appropriate grid discretisation. Using this discrete grid, construct a trial wavefunction  $\psi(x, y) = \sum_{n=0}^{10} c_n \phi_n(x)$  where  $\phi_n(x, y)$  are the 10 lowest-energy eigenstates of the 2D quantum harmonic oscillator, and the initial values  $c_n$  are

$$\mathbf{c} = (5, 1, 1, 1, 1, 1, 0.5, 0.5, 0.5, 0.5).$$

Make sure to normalize  $\psi$  after construction.

- (c) Using a finite difference approximation and Simpson's integration method, calculate the energy expectation value  $E^{(0)} = \langle \psi | \hat{H} | \psi \rangle$  of the trial wavefunction.
- (d) Using a random number generator, choose a random coefficient in the basis expansion and alter it by a random small amount. Recalculate the energy expectation value — is it larger or smaller? Renormalise the wavefunction.
- (e) Repeat this process, keeping the random changes that reduce the energy of the trial wavefunction, and discarding those that don't. Stop once the energy value has converged such that  $|E^{(n+1)} - E^{(n)}| \leq 1 \times 10^{-6}$ . What is the converged ground state energy?
- (f) Plot the final trial wavefunction, representing the ground state.
- (g) How long did it take for this crude optimization process to converge? How does that compare to previous basis diagonalisation approaches?
- (h) *Bonus:* repeat the algorithm, this time using Monte-Carlo integration to calculate the two-dimensional energy expectation integral. How does this affect the convergence rate?

**P10.9** Consider a non-orthogonal basis expansion

$$|\psi\rangle = \sum_n c_n |\phi_n\rangle, \quad \langle \psi | \psi \rangle = \sum_m \sum_n c_m^* c_n S_{mn},$$

where  $S$  is the overlap matrix. Substitute this into the variational principle, and introduce an infinitesimal perturbation  $|\psi'\rangle = |\psi\rangle + |\delta\psi\rangle$ .

Solve this for the case where the corresponding energy perturbation  $\delta E = 0$ , and show that this is equivalent to solving the generalised eigenvalue problem  $H\mathbf{c} = E\mathbf{S}\mathbf{c}$ .



## Chapter 11

# Time Propagation

In the previous chapters, we used separation of variables to reduce the Schrödinger's equation to the time-independent Schrödinger's equation, and showcased various techniques and methods to determine the energy eigenstates. This is an extremely useful approach when bound states need to be determined and investigated, and is used to analyse atoms, molecules, and other diverse structures.

When the eigenstates and energy eigenvalues have been calculated, separation of variables also allows us to recover the time evolution of the bound state,

$$\psi_n(x, t) = \psi_n(x)e^{-iE_nt/\hbar}. \quad (11.1)$$

As the energy eigenstates form an orthonormal basis for the Hilbert space, we can then construct any time-dependent solution by taking a linear combination of the eigenstate evolutions,

$$\psi(x, t) = \sum_n c_n \psi_n(x)e^{-iE_nt/\hbar} \quad (11.2)$$

where the coefficients  $c_n \in \mathbb{C}$  uniquely determine the time-dependent wavefunction.

However, this approach often requires a very large number of discrete eigenstates, and may even require integration over the continuous part of the energy spectrum — i.e. the unbound ‘free’ or ‘scattering’ states. In addition, the separation of time and spatial variables implies an explicitly time-independent Hamiltonian, and so the system under investigation can only be analysed under steady-state conditions. This is a hindrance when studying the dynamic response of quantum systems, which is of particular importance for analysing high-speed nano-structured quantum transistors, quantum switches, and quantum logic gates.

In these situations, one needs to solve directly the time-dependent Schrödinger's equation, stated as an initial value problem rather than a

boundary value problem. This time-dependent approach has a more natural correspondence to reality, i.e. starting from an initial state of the system and advancing it in time under the influence of internal and external interaction potentials, which can be either time-independent or time-dependent. In other words, it provides information on transient behaviours, and allows direct visualisation of the transport process, where one can ‘watch’ a system evolve in real time and as a result monitor intermediate stages of the process of interest. As an initial value problem, it is also comparatively easy to implement, flexible, and versatile in treating a large variety of quantum problems. In particular, it removes the need for boundary conditions, which can be difficult to implement for complex shaped potentials using the time-independent methods. On the other hand, you now have an *additional* variable,  $t$ , over which the model must be discretised, slightly increasing the complexity.

In this chapter, we will work through methods and techniques to solving the time-dependent Schrödinger’s equation, and investigate quantum scattering of particles from arbitrarily complex potentials, as well as potentials varying with time.

## 11.1 The Unitary Propagation Operator

Recall from Sect. 9.1 that the time-dependent Schrödinger’s equation is given by

$$i\hbar \frac{\partial}{\partial t} \psi(\mathbf{x}, t) = \hat{H}\psi(\mathbf{x}, t), \quad \hat{H} = -\frac{\hbar^2}{2m} \nabla^2 + V(\mathbf{x}, t) \quad (11.3)$$

where we are now allowing the potential  $V$  to vary in time. We are by now familiar with the process of discretising the position grid; in order to solve this numerically, we need to discretise time  $t$ . To discretise the potential in time, we apply an analogous process to the time grid:

$$V(\mathbf{x}, t) \rightarrow V_n(\mathbf{x}) \equiv V(\mathbf{x}, t_n), \quad t_n = t_0 + n\Delta t. \quad (11.4)$$

This allows the potential to be treated as time independent within each time step  $\Delta t$ . In essence, we are breaking the problem into many small time-independent potentials  $V_n$ . Using this process to ‘remove’ the time-dependence from the potential, it can be seen that by interpreting it as a first order differential equation in time (that is, ignoring any potential differential operators present in  $\hat{H}$ ), there is a formal (albeit, not generally very useful analytically) solution available:

$$\psi(\mathbf{x}, \Delta t) = e^{-i\hat{H}\Delta t/\hbar} \psi(\mathbf{x}, 0) = \sum_{n=0}^{\infty} \frac{(-1)^n}{n!} \left( \frac{i\hat{H}\Delta t}{\hbar} \right)^n \psi(\mathbf{x}, 0) \equiv \hat{U}(\Delta t) \psi(\mathbf{x}, 0), \quad (11.5)$$

- Note that, if the initial state is chosen to be an energy eigenstate  $\psi_n(\mathbf{x})$ , then

$$\hat{U}(\Delta t) \psi_n(\mathbf{x}) = e^{-iE_n t/\hbar} \psi_n(\mathbf{x}),$$

and we remain in this eigenstate for all  $\Delta t$ .

where  $\psi(\mathbf{x}, 0)$  is the initial condition or ‘starting state’, and  $\hat{U}(\Delta t) = e^{-i\hat{H}t/\hbar}$  is the **unitary propagation operator**. It is easy to see that the unitarity

of the propagation operator follows from the fact that the Hamiltonian is Hermitian:

$$\hat{U}\hat{U}^\dagger = e^{-i\hat{H}t/\hbar}e^{-i\hat{H}t/\hbar}^\dagger = e^{-i\hat{H}t/\hbar}e^{i\hat{H}t/\hbar} = I. \quad (11.6)$$

► A unitary operator satisfies the relationship  $UU^\dagger = I$ .

This unitarity is important, as it ensures that the inner-product in the Hilbert space remains invariant under time-propagation,

$$\langle \psi(t) | \psi(t) \rangle = \langle \psi(0) | \hat{U}(t)^\dagger \hat{U}(t) | \psi(0) \rangle = \langle \psi(0) | \psi(0) | \rangle \quad (11.7)$$

as a result, a normalised state acted on by the propagation operator *remains* normalised for the entirety of its time-evolution. Another important property of the time-evolution operator is that it is **reversible** or **symmetric** in time,

$$\hat{U}(-\Delta t)\hat{U}(\Delta t) |\psi(x, t)\rangle = |\psi(x, t)\rangle, \quad (11.8)$$

i.e. reversing the time recovers the initial state of the system. Together with unitarity, these two properties are always satisfied by a quantum dynamical system.

Although this formal solution is well known, computational techniques for treating this exponential propagator are notoriously fickle in ensuring both high accuracy and reasonable convergence. In fact, finding efficient and accurate methods of calculating the matrix exponential has been an open problem for decades, with the tide finally starting to turn due to the development of Krylov-subspace techniques hand-in-hand with more powerful methods of computation.

In the next couple of sections, we will explore some common schemes for numerically calculating the time-evolution or propagation operator  $\hat{U}(\Delta t)$ .

### The power of propagating solutions

Since the wave function  $\psi(\mathbf{r}, t)$  can be expanded in terms of the energy eigenstates, they contain complete quantum-mechanical information about the system under study. As a result, we can derive from the wavefunction all possible observables; these include reflection and transmission coefficients, electric current and current density, lifetime of trapped states, amongst others. For example, the energy eigenvalues can also be obtained from the time propagation of system wavefunctions by performing the time-energy Fourier transform.

## 11.2 Euler Methods

Of the many schemes that have been proposed to tackle the time-dependent Schrödinger's equation, the simplest is the first-order finite-difference approximation to the exponential operator, that is,

$$\psi(\mathbf{x}, t + \Delta t) = \left( 1 - \frac{i\Delta t}{\hbar} \hat{H} + \mathcal{O}(\Delta t^2) \right) \psi(\mathbf{x}, t), \quad (11.9)$$

which is widely known as the Euler method. Since it only retains the first term in the Taylor expansion, it requires a large number of extremely small time steps  $\Delta t$ , sometimes over a few million time steps, to develop a complete time evolution. From this approximation, familiar techniques can then be used to discretize and solve the system. For example, we can couple the Euler method with the fourth-order method of finite differences.

- Here, we are taking the first two terms of the Taylor series expansion of the matrix exponential. This is identical to what we did when we derived the Euler method for differentiation back in Sect. 5.2, hence the same name.
- While the local error of the Euler method is of order  $\mathcal{O}(\Delta t^2)$ , the **global error** is of order  $\mathcal{O}(\Delta t)$  — hence the Euler method is a first order approximation.

### Discretising the Position Space

To use the method of finite difference method to discretise the position space, we first subdivide the position grid into  $N$  discrete points  $x_{j+1} - x_j = \Delta x \ll 1$ , and discretise the wavefunction such that  $\psi(x_j, t) = \psi_j(t)$ . Note that, for simplicity, we are considering only the one-dimensional case here, but this process generalises to higher dimensions trivially. The fourth-order finite difference approximation is

$$\frac{d^2}{dx^2} \psi_j(t) = \frac{16(\psi_{j+1}(t) + \psi_{j-1}(t)) - 30\psi_j(t) - (\psi_{j+2}(t) + \psi_{j-2}(t))}{12\Delta x^2}; \quad (11.10)$$

substituting this into the Euler method, we get

$$\begin{aligned} \psi_j(t + \Delta t) &= \left( 1 - \frac{i\Delta t}{\hbar} V_j(t) \right) \psi_j(t) \\ &+ \frac{i\hbar\Delta t}{24m\Delta x^2} [16(\psi_{j+1}(t) + \psi_{j-1}(t)) - 30\psi_j(t) - (\psi_{j+2}(t) + \psi_{j-2}(t))]. \end{aligned} \quad (11.11)$$

Therefore, for the Euler method, we would proceed as follows:

- Discretise the initial state  $\psi(x, t_0)$  in position space
- Apply the Euler method with finite difference approximation above to determine  $\psi_j(t_0 + \Delta t)$  for all  $j$  in the domain, and for some  $\Delta t \ll 1$ .
- Repeat this process until you have determined the wavefunction at time  $t = t_0 + N\Delta t$ .

The Euler method is a first order expansion of the propagation operator, so we can expect local errors of order  $\mathcal{O}(\Delta t^2)$ . Unfortunately, this is not the

major shortcoming of the Euler method. In fact, it is easy to see that this first order truncated propagation operator is no longer unitary,

$$\left(1 - \frac{i\Delta t}{\hbar} \hat{H}\right) \left(1 - \frac{i\Delta t}{\hbar} \hat{H}\right)^\dagger = 1 + \frac{(\Delta t)^2}{\hbar^2} \hat{H}^2 \neq I \quad (11.12)$$

and as a result the probability amplitude of the propagating wavefunction has the potential to grow or decay exponentially. This can be mitigated to some extent by ensuring the timesteps  $\Delta t$  are exceptionally small, and by renormalising the wavefunction after each time step. It is therefore not a suitable scheme except for very simple cases.

► Furthermore, note also that the Euler method does not preserve time symmetry!

### 11.2.1 The Backwards Euler Method

This non-unitary behaviour is indicative of a wider problem — the Euler method is inherently unstable. If you recall from our investigation of the Euler method back in Sect. 5.4, we found that the backwards Euler method — an *implicit* method — was significantly more stable than the explicit forward Euler method. We can try this approach here as well to approximate the exponential:

$$\psi(\mathbf{x}, t + \Delta t) = \psi(\mathbf{x}, t) - \frac{i\Delta t}{\hbar} \hat{H} \psi(\mathbf{x}, t + \Delta t) + \mathcal{O}(\Delta t^2), \quad (11.13)$$

or, rearranging to bring together both  $\psi(x, t + \Delta t)$  terms,

$$\left(1 + \frac{i\Delta t}{\hbar} \hat{H}\right) \psi(\mathbf{x}, t + \Delta t) = \psi(\mathbf{x}, t) + \mathcal{O}(\Delta t^2). \quad (11.14)$$

As with other implicit methods we have covered, we cannot simply iterate forward to determine  $\psi(\mathbf{x}, t + \Delta t)$ . Instead we need to employ a root finding algorithm to solve the coupled system of linear equations. In this particular case, we could represent  $\hat{H}$  as a matrix (either using the finite difference approximation, or by basis diagonalisation  $H_{ij} = \langle \phi_m | \hat{H} | \phi_n \rangle$ ), and calculate the matrix inverse

$$\psi(\mathbf{x}, t + \Delta t) = \left(1 + \frac{i\Delta t}{\hbar} \hat{H}\right)^{-1} \psi(\mathbf{x}, t) + \mathcal{O}(\Delta t^2). \quad (11.15)$$

However, while significantly more stable than the forwards Euler method, the backwards Euler method (as you can see) is *still* non-unitary and non-reversible.

### 11.2.2 The Crank–Nicolson Approximation

To avoid the numerical instability, one can combine the backwards and forwards Euler method in such a way as to preserve unitarity of the approximated propagation operator — this is the thinking behind the **second-order Crank–Nicolson method**. Instead of performing one time-step of  $\Delta t$  using the Euler method, let's instead perform two time-steps, each of size  $\Delta t/2$ , once using the forward Euler method, and then again with the backwards Euler method:

$$\begin{aligned} t_0 + \Delta t/2 : \quad \psi(\mathbf{x}, t + \Delta t/2) &= \left(1 - \frac{i\Delta t}{2\hbar}\hat{H} + \mathcal{O}(\Delta t^2)\right) \psi(\mathbf{x}, t) \\ t_0 + \Delta t : \quad \left(1 + \frac{i\Delta t}{2\hbar}\hat{H} + \mathcal{O}(\Delta t^2)\right) \psi(\mathbf{x}, t + \Delta t) &= \left(1 - \frac{i\Delta t}{2\hbar}\hat{H} + \mathcal{O}(\Delta t^2)\right) \psi(\mathbf{x}, t) \end{aligned} \quad (11.16)$$

Rearranging this expression, we get the Crank–Nicolson approximation,

$$\boxed{\psi(\mathbf{x}, t + \Delta t) = \frac{2 - i\hat{H}\Delta t/\hbar}{2 + i\hat{H}\Delta t/\hbar} \psi(\mathbf{x}, t)} + \mathcal{O}(\Delta t^2). \quad (11.17)$$

Unlike the Euler method, the Crank–Nicolson method is unitary and unconditionally stable. However, due to the inclusion of an implicit method, the Crank–Nicolson method involves the inversion of potentially large and poorly conditioned matrices, which can be prohibitively expensive in terms of computer memory and CPU time.

## 11.3 Split Operators

- The split operator method is commonly used when solving the **non-linear Schrödinger equation**, in order to separate the linear and non-linear parts of the propagation operator.

A notable scheme for increasing the accuracy of unitary propagation operator approximations is the **split operator method**, so called because it involves ‘splitting’ the unitary operator into three terms,

$$\boxed{\psi(\mathbf{x}, t + \Delta t) \approx e^{-i\hat{H}_0\Delta t/2\hbar} e^{-i\hat{V}(t)\Delta t/2\hbar} e^{-i\hat{H}_0\Delta t/2\hbar} \psi(\mathbf{x}, t)}, \quad (11.18)$$

where  $\hat{H}_0 = -\frac{\hbar^2}{2m}\nabla^2$  is the Hamiltonian of a free particle, and  $\hat{V}(t)$  is the potential operator. That is, we are splitting the propagation of the wavefunction into two ‘free’ propagations and one ‘interacting’ propagation. Straight away, we can see that since  $\hat{V}(t)$  and  $\hat{H}_0$  are Hermitian, the split operators are all unitary and also time-reversible, satisfying two of the important criteria we set for solutions of the time-dependent Schrödinger’s equation. From here, we could apply approximations to the split operators to propagate the solution, for example the Crank–Nicolson method.

However,  $\hat{H}_0$  and  $\hat{V}(t)$  are convenient since we already know their orthogonal eigenstates; and since we know their eigenstates, we therefore know the bases in which their matrix representation are diagonal:

- $\hat{V}(t)$  is diagonal in **position space**, with  $\hat{V}(t)|x\rangle = V(\mathbf{x}, t)|x\rangle$ ,
- $\hat{H}_0 = -\frac{\hbar^2}{2m}\nabla^2 = \frac{\hat{p}^2}{2m}$  is diagonal in **momentum space**, with  $\hat{H}_0|p\rangle = \frac{p^2}{2m}|p\rangle$ .

Furthermore, we know how to convert between the position space and the momentum space, it is simply the Fourier transform! Therefore, this allows us to replace the exponential operators with the exponential of the eigenvalues, as follows:

$$\psi(x, t + \Delta t) \approx \int_{-\infty}^{\infty} dp' e^{-i\rho' x/\hbar} e^{-i\rho'^2 \Delta t / 4m\hbar} \int_{-\infty}^{\infty} d\xi' e^{i\rho' \xi'/\hbar} e^{-iV(\xi', t)\Delta t / 2\hbar} \\ \int_{-\infty}^{\infty} dp e^{i\rho \xi'/\hbar} e^{-i\rho^2 \Delta t / 4m\hbar} \int_{-\infty}^{\infty} d\xi e^{-i\rho \xi/\hbar} \psi(\xi, t), \quad (11.19)$$

where we are performing successive Fourier transforms and inverse Fourier transforms to ensure the acting split operator is diagonal. Using the notation  $F(p) = \mathcal{F}[f(x)]$  to denote the Fourier transform, the nested nature of the Fourier transforms can be made slightly more clear:

$$\begin{aligned} \psi(x, t + \Delta t) &\approx \mathcal{F}^{-1} \left[ e^{-i\mathbf{p}^2 \Delta t / 4m\hbar} \mathcal{F} \left[ e^{-iV(\mathbf{x}, t)\Delta t / 2\hbar} \mathcal{F}^{-1} \left[ e^{-i\mathbf{p}^2 \Delta t / 4m\hbar} \mathcal{F}[\psi(\mathbf{x}, t)] \right] \right] \right]. \end{aligned}$$

Beware, however — the split operator method is an *approximation*, not an exact result. This is because the property of the exponential that we are exploiting,  $e^{A+B} = e^A e^B$ , only applies when the arguments commute, that is,  $[A, B] = 0$ . If this is not the case, then instead we must expand using the **Baker–Campbell–Hausdorff** formula,

$$e^A e^B = \exp \left[ A + B + \frac{1}{2}[A, B] + \frac{1}{12} ([A, [A, B]] + [B, [B, A]]) + \dots \right]$$

In general,  $[\hat{H}_0, \hat{V}] \neq 0$ , since the potential is position-dependent and the free-space Hamiltonian contains a Laplacian operator (double check this result for yourself!). As a result, by truncating the expansion to just the exponentials of single operators, the local error is of order

$$\text{local error} = \frac{i\Delta t^3}{24\hbar^3} ([[\hat{H}_0, \hat{V}(t)], \hat{H}_0] + 2[[\hat{H}_0, \hat{V}(t)], [\hat{V}(t)]]) \quad (11.20)$$

Considering the  $N = (t_1 - t_0)/\Delta t$  time-steps we need to perform in order to calculate the final solution, the global error is therefore of order  $\mathcal{O}(\Delta t^2)$ , and this is a **second-order** method.

For certain potentials  $\hat{V}$ , however, the expansion might terminate for all commutators of a particular order and higher; for those systems, the expansion converges or terminates, and the solution obtained is exact and easily computed numerically. Unfortunately, in general the expansion does not converge or terminate, and is less practical when used in computations for arbitrarily shaped electric and magnetic potentials.

► The FFT algorithm can be used to very efficiently approximate the multi-dimensional Fourier transform.

► For the split operator method, we perform **second-order** symmetric splitting

$$e^{A+B} \approx e^{A/2} e^B e^{A/2},$$

as opposed to first order splitting

$$e^{A+B} \approx e^A e^B$$

to allow increased accuracy.

This is commonly referred to as **Strand splitting**.

Since it is a second-order method, second-order methods (or higher) should be applied to each split operator.

## 11.4 Direct Time Discretisation

In the above approaches, we used techniques to approximate the propagation operator  $\hat{U}(t)$ , based on the formal solution  $\hat{U}(t) = e^{-i\hat{H}t/\hbar}$ . Rather than using the formal solution, we can instead discretise the time-dependent Schrödinger equation directly through the use of the central finite difference approximation. In addition to avoiding the instability issues seen above, we also avoid the use of implicit methods that require root-finding algorithms or matrix inversions.

### Second-Order Finite Differences

To begin with, we discretise both the time and the position grids:

$$\begin{aligned} x_{i+1} - x_i &= \Delta x, & t_{j+1} - t_j &= \Delta t; \\ \psi_{i,j} &\equiv \psi(x_i, t_j), & V_{i,j} &\equiv V(x_i, t_j). \end{aligned} \quad (11.21)$$

Note that in this example we will consider the one-dimensional case; this extends to the two- and three-dimensional case in a similar manner. Apply the centered finite difference approximation to the first-order time derivative in the Schrödinger equation,

$$i\hbar \frac{\psi_{i,j+1} - \psi_{i,j-1}}{2\Delta t} = \hat{H}\psi_{i,j}. \quad (11.22)$$

Next, we can apply a fourth-order finite difference method to approximate the action of  $\hat{H}$  on the wavefunction:

$$\begin{aligned} i\hbar \frac{\psi_{i,j+1} - \psi_{i,j-1}}{2\Delta t} \\ = -\frac{\hbar^2}{24m\Delta x^2} [16(\psi_{i+1,j} + \psi_{i-1,j}) - 30\psi_{i,j} - (\psi_{i+2,j} + \psi_{i-2,j})] + V_{i,j}\psi_{i,j} \end{aligned}$$

We can write this in a more compact form by letting

$$g_{i,j} = -\frac{2i\Delta t}{\hbar} \left( \frac{5\hbar^2}{4m\Delta x^2} + V_{i,j} \right), \quad a = \frac{i\hbar\Delta t}{12m\Delta x^2}; \quad (11.23)$$

substituting this into the approximation gives

$$\boxed{\psi_{i,j+1} = g_{i,j}\psi_{i,j} + 16a(\psi_{i+1,j} + \psi_{i-1,j}) - a(\psi_{i+2,j} + \psi_{i-2,j}) + \psi_{i,j-1}}.$$

Thus, to determine  $\psi_{i,j+1}$ , we need knowledge of the wavefunction  $\psi_{i,j}$ , as well as the previous timestep value  $\psi_{i,j-1}$ .

We can also write this set of coupled linear equations as a matrix equation,

$$\psi_{j+1} = A_j\psi_j + \psi_{j-1}, \quad (11.24)$$

where  $\psi_j = (\psi_{0,j}, \psi_{1,j}, \dots, \psi_{N-1,j})$  is a vector containing the spatial elements of the wavefunction for a particular timestep, and the matrix  $A_j$  is given by

$$(A_j)_{mn} = \begin{cases} g_{m,j}, & m = n \\ 16a, & |m - n| = 1 \\ -a, & |m - n| = 2 \end{cases} \quad (11.25)$$

(i.e. the diagonal, first off-diagonal, and second off-diagonals respectively).

- The centered finite difference equation leads to the leap-frog method.

- This can be extended to the 2 and 3-dimensional case in an analogous fashion as we did in the previous chapter — by writing out the system of linear equations, and determining the matrix structure.

### 11.4.1 Stability

For a general parabolic partial differential equation of the form

$$\frac{\partial u}{\partial t} = \beta \frac{\partial^2 u}{\partial x^2} + f(x, t), \quad (11.26)$$

it turns out that we require  $\Delta t \leq \Delta x^2 / |2\beta|$  to guarantee stability of the second-order finite difference method. By comparing this to the Schrödinger equation in the case of no potential, we can see that

$$\Delta t \lesssim \left| \frac{m\Delta x^2}{i\hbar} \right| = \frac{1}{\hbar} m\Delta x^2. \quad (11.27)$$

If there is a non-zero potential, this is modified slightly:

$$\Delta t \lesssim \frac{m\Delta x^2}{\hbar + m\Delta x^2 V_{max}/2\hbar}, \quad (11.28)$$

where  $V_{max}$  is the maximum value of the potential.

So whilst the second-order finite difference method is unitary and time symmetric, a very small timestep  $\Delta t$  is required to ensure stability, resulting in an increase of computational time required to obtain accurate results.

### 11.4.2 Runge–Kutta method

To increase the accuracy with direct discretisation approaches, we can apply higher order discretisation techniques. One of the most popular for solving the Schrödinger equation is RK4, the fourth-order Runge–Kutta method that is significantly more stable than both the second-order leap-frog method and the Euler methods; so you can see why it remains so popular!

► See Sect. 5.9 to refresh your memories on the details of the RK4 method.

Unlike the second-order leap-frog discretisation we just explored, here we instead *semi*-discretise the Schrödinger equation in space (not time!), producing a set of coupled, linear first-order ODEs in time. These can then be solved by numerically integrating the system via the 4th order Runge–Kutta algorithm.

To begin the (semi) discretisation, we first must choose a discretisation scheme over the spatial coordinates. To ensure the truncation error remains bounded by  $\Delta x^4$ , let's choose the fourth-order finite difference approximation. Substituting this into the Schrödinger equation and rearranging, we get

$$\begin{aligned} \frac{\partial}{\partial t} \psi_i(t) &= \frac{2i\hbar}{3m\Delta x^2} (\psi_{i+1} + \psi_{i-1}) \\ &\quad - \frac{i\hbar}{24m\Delta x^2} (\psi_{i+2} + \psi_{i-2}) - \frac{i}{\hbar} \left[ \frac{5\hbar^2}{4m\Delta x^2} + V_i(t) \right] \psi_i(t), \end{aligned} \quad (11.29)$$

for  $\psi_i(t)$ ,  $i = 0, 1, \dots, N - 1$ . We can write this as a matrix equation,

$$\begin{bmatrix} \psi'_1(t) \\ \psi'_2(t) \\ \psi'_3(t) \\ \vdots \\ \psi'_{N-3}(t) \\ \psi'_{N-2}(t) \end{bmatrix} = \begin{bmatrix} g_1(t) & 16a & -a & 0 & \cdots & 0 \\ 16a & g_2(t) & 16a & -a & \cdots & \vdots \\ -a & 16a & g_3(t) & 16a & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & -a \\ \vdots & \ddots & \ddots & \ddots & g_{N-3} & 16a \\ 0 & \cdots & 0 & -a & 16a & g_{N-2}(t) \end{bmatrix} \begin{bmatrix} \psi_1(t) \\ \psi_2(t) \\ \psi_3(t) \\ \vdots \\ \psi_{N-3}(t) \\ \psi_{N-2}(t) \end{bmatrix}, \quad (11.30)$$

where

- Note that  $g_i(t)$  and  $a$  are defined slightly differently from in the previous section, as we do not take into account time discretisation here.

$$g_{i,j} = -\frac{i}{\hbar} \left( \frac{5\hbar^2}{4m\Delta x^2} + V_{i,j} \right), \quad a = \frac{i\hbar}{24m\Delta x^2}. \quad (11.31)$$

Or, more succinctly,

$$\frac{d}{dt} \psi(t) = \mathbf{f}(\psi, t) = M(t)\psi(t), \quad (11.32)$$

where  $M(t)$  is the  $(N-2) \times (N-2)$  matrix defined above,  $\psi(t) = (\psi_1(t), \dots, \psi_{N-2}(t))$ , and the initial condition is  $\psi(t_0) = \psi_0$ . Compare this to Sect. 5.9.1 — we have a multivariable system of coupled linear ODEs. Discretising the time parameter such that

$$\frac{d}{dt} \psi_n = M(t_n)\psi_n, \quad (11.33)$$

we can now apply the RK4 algorithm to determine the time-evolution of the initial state  $\psi_0$ :

$$\psi_{n+1} = \psi_n + \frac{1}{6}\Delta t(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) + \mathcal{O}(\Delta t^5), \quad (11.34)$$

where the stage vectors are

$$\begin{aligned} \mathbf{k}_1 &= M(t_n)\psi_n \\ \mathbf{k}_2 &= M\left(t_n + \frac{1}{2}\Delta t\right)\left(\psi_n + \frac{1}{2}\mathbf{k}_1\Delta t\right) \\ \mathbf{k}_3 &= M\left(t_n + \frac{1}{2}\Delta t\right)\left(\psi_n + \frac{1}{2}\mathbf{k}_2\Delta t\right) \\ \mathbf{k}_4 &= M(t_n + \Delta t)(\psi_n + \mathbf{k}_3\Delta t) \end{aligned} \quad (11.35)$$

## 11.5 The Chebyshev Expansion

So far, all the techniques we have explored, whether approximating the unitary propagation operator like Crank–Nicolson, directly discretising the time-step like Runge–Kutta, or applying pseudo-spectral methods like the split-operator approach, are all **local approximations** to the propagation operator. That is, due to either instability or inaccuracy for large values of  $\Delta t$ , they are only accurate within a small region of the starting time  $t_0$  — thus they must be performed iteratively over many small values of  $\Delta t \ll 1$ .

Alternatively, the Chebyshev expansion is a **global approximation**; it is valid for *any* value of  $\Delta t$ , allowing us to calculate the final state of the system directly, given the Hamiltonian and initial state — especially useful if we are not interested in any of the intermediate dynamics. It does so by expanding the unitary propagation operator as a series expansion of **Chebyshev polynomials**, unlike the more common power series approach used by the Taylor expansion.

The Chebyshev series expansion of the unitary time propagation operator is given by

$$\begin{aligned} \psi(\mathbf{x}, t + \Delta t) \\ = e^{-i(E_{\max} + E_{\min})\Delta t / 2\hbar} \left[ J_0(\alpha)T_0(-i\tilde{H}) + 2 \sum_{n=1}^{\infty} J_n(\alpha)T_n(-i\tilde{H}) \right] \psi(\mathbf{x}, t), \end{aligned}$$

where

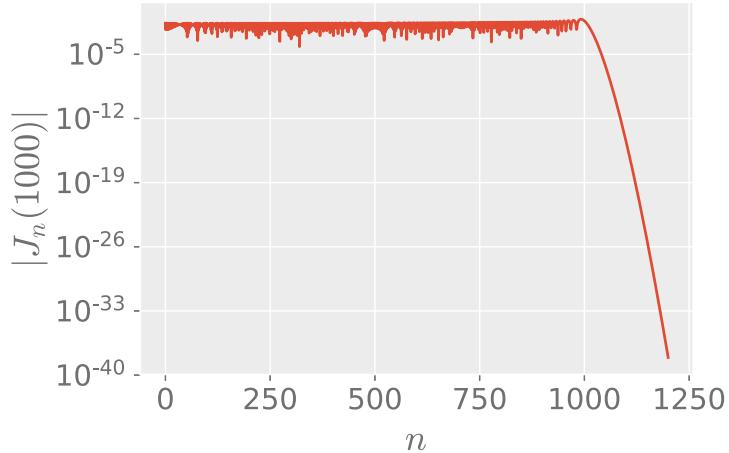
- $E_{\min}, E_{\max} \in \mathbb{R}$  are the minimum and maximum energy eigenvalues of the system Hamiltonian  $\hat{H}$ ,
- $\alpha = (E_{\max} - E_{\min})\Delta t / 2\hbar$ ,
- $J_n(\alpha)$  are the Bessel functions of the first kind,
- $T_n$  are the Chebyshev polynomials of the first kind,
- the **normalised Hamiltonian** is defined as

$$\tilde{H} = \frac{2\hat{H} - E_{\max} - E_{\min}}{E_{\max} - E_{\min}}.$$

The power of the Chebyshev expansion as a global method comes from the Bessel function series coefficients, as it turns out that  $J_n(\alpha) \approx 0$  when  $n > |\alpha|$ , allowing for fast convergence and significantly higher accuracy after only  $|\alpha|$  terms. This is perhaps more evident by plotting the Bessel functions  $|J_n(\alpha)|$  vs.  $n$  for a fixed value of  $\alpha = 1000$ ; see the following plot.

► Pafnuty Chebyshev (1821–1894) was a Russian mathematician, well known for his work in probability theory and mechanics.

► We must normalise the Hamiltonian so that its energy eigenvalues lie in the domain  $E \in [-1, 1]$ ; this allows maximal convergence of the Chebyshev expansion.



- ▶ For simplicity, it is convenient when constructing the Chebyshev expansion, to instruct the series to be truncated when  $|2J_n(\alpha)| \leq \epsilon$  for some  $\epsilon \ll 1$ .

This can be calculated in advance of computing the series expansion, or even during the series expansion with a `while` loop.

Note that the  $y$ -axis is using a logarithmic scale — this should impress just how rapidly the Bessel series coefficients approach zero as  $n \approx \alpha$ . However, recall that  $\alpha$  is proportional to both  $\Delta t$  and  $E_{max} - E_{min}$ ; thus for longer evolution times, and Hamiltonians with larger ranges of energy eigenvalues, an increase in the number of series terms is required to reach this convergence point.

### Time-dependent Hamiltonians

If the Hamiltonian is time-varying, perhaps due to a time-dependent potential  $V(\mathbf{x}, t)$ , then the Chebyshev approximation *cannot* be applied globally. As with the Crank–Nicholson method, we need to discretise the Hamiltonian in time such that  $t_{i+1} - t_i = \Delta t \ll 1$ ,  $\hat{H}_i \equiv \hat{H}(t_i)$ , and then propagate each of these time-steps independently and iteratively.

#### 11.5.1 Implementing Bessel and Chebyshev Functions

As we have seen with the generalised Laguerre and associated Legendre functions, sometimes it can be difficult to utilise special functions numerically. Both SciPy and Fortran provide intrinsic functions for computing the Bessel functions of the first kind, namely

```
scipy.special.jv(n, x)
```

```
bessel_j(n, x)
```

respectively. Unfortunately, while SciPy provides Chebyshev polynomials (`scipy.special.eval_chebyt(n, x)`), we cannot simply make use of the SciPy Chebyshev function, since it only accepts a floating point value  $x$ , whereas our argument is  $\tilde{H}$ , an operator or a matrix!

Thankfully, the Chebyshev polynomials satisfy a very convenient set of recurrence relations,

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x), \end{aligned} \quad (11.36)$$

► For example, applying the recurrence relation,

$$\begin{aligned} T_2(x) &= 2xT_1(x) - T_0(x) \\ &= 2x^2 - 1. \end{aligned}$$

which generalise in the case of operator arguments:

$$\begin{aligned} T_0(-i\tilde{H})\psi(\mathbf{x}, t) &= \psi(\mathbf{x}, t) \\ T_1(-i\tilde{H})\psi(\mathbf{x}, t) &= -i\tilde{H}\psi(\mathbf{x}, t) \\ T_{n+1}(-i\tilde{H})\psi(\mathbf{x}, t) &= -2i\tilde{H}T_n(-i\tilde{H})\psi(\mathbf{x}, t) - T_{n-1}(-i\tilde{H})\psi(\mathbf{x}, t). \end{aligned} \quad (11.37)$$

**In summary:** To use the Chebyshev series expansion to propagate a wavefunction under the influence of the Hamiltonian  $\hat{H}$  for time  $t$ :

- (1) Determine if the system allows global propagation for time  $t$ ; if the Hamiltonian is time-dependent, then you will need to discretise  $\hat{H}$  into discrete time-steps  $\Delta t \ll 1$ , and propagate each of these sequentially.
- (2) Choose an appropriate basis in which to construct the matrix  $H_{ij} = \langle \phi_i | \hat{H} | \phi_j \rangle$ , or use the finite difference approximation to find the matrix representation in Cartesian coordinates.
- (3) Use an eigenvalue solver to determine the minimum and maximum eigenvalues of  $H_{ij}$ ; use these to normalise the Hamiltonian  $\tilde{H}$ , and calculate  $\alpha = (E_{max} - E_{min})\Delta t/2\hbar$ .
- (4) **Construct the series expansion.**

The first two terms are

$$J_0(\alpha)\psi(\mathbf{x}, t) - 2J_1(\alpha)i\tilde{H}\psi(\mathbf{x}, t),$$

with subsequent terms determined using the Chebyshev recurrence relation.

Continue the expansion until  $|J_n(\alpha)| \leq \epsilon \ll 1$ .

- (5) Finally, multiply the expansion by the phase factor  $e^{-i(E_{max} + E_{min})\Delta t/2\hbar}$ .

► Unlike the other methods explored, the Chebyshev expansion requires that the minimum and maximum eigenvalues of the operator  $\hat{H}$  are known for optimum convergence. Depending on the size of the system, this can be the largest computational bottleneck

Possible solutions involve utilizing sparse matrix eigenvalue algorithms that only return the largest and smallest eigenvalues, or even guessing/over-estimating  $E_{max}$  and  $E_{min}$ .

## 11.6 The Nyquist Condition

So far, we have considered approximations to the time-propagation of initial states, without really worrying about what these states *are* or what they look/behave like. How do we choose the initial state in our propagation problem? In general, this is highly dependent on the problem itself; the potential present might suggest a particular state, or you may be working from experimental data, and already know the initial state you would like to investigate.

When working with potential barriers, scattering, and particles travelling in free space, a useful initial state is the **Gaussian wave packet**, in one-dimension given by

$$\psi(x, 0) = \frac{1}{(2\pi\sigma^2)^{1/4}} e^{-(x-x_0)^2/4\sigma^2} e^{ik_0 x}, \quad (11.38)$$

where

- $\sigma$  is the standard deviation of the wave packet; it is related to the uncertainty in the position of the wave packet via  $\sigma = \Delta x$ ,
- $\langle \hat{x} \rangle = x_0$  is the  $x$  expectation value of the wave packet,
- $\langle \hat{k} \rangle = k_0$  is the initial wavenumber of the wave packet.

Thus, the Gaussian wavepacket represents a particle localised in space around the point  $x_0$ , with uncertainty  $\Delta x$ , and momentum centered around  $p_0 = \hbar k_0$ .

We can take the Fourier transform of this wave packet, in order to find its representation in momentum space:

$$\phi(k, 0) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \psi(x, 0) e^{-ikx} dx = \left( \frac{2\sigma^2}{\pi} \right)^{1/4} e^{-(k-k_0)^2\sigma^2} e^{-ix_0(k-k_0)}. \quad (11.39)$$

As the Fourier transform of a Gaussian function is another Gaussian function, this result is no surprise; the probability distribution in momentum space is

$$|\phi(k, 0)|^2 = \sigma \sqrt{\frac{2}{\pi}} e^{-2(k-k_0)^2\sigma^2}. \quad (11.40)$$

We can see that the standard deviation is now  $\sigma_k = 1/2\sigma$ , and thus the uncertainty in the wavenumber is  $\Delta k = 1/2\sigma$ . In fact, if we convert this to momentum uncertainty via the de Broglie relation  $\Delta p = \hbar\Delta k = \hbar/2\sigma$ , we see that

$$\Delta x \Delta p = (\sigma) \left( \frac{\hbar}{2\sigma} \right) = \frac{\hbar}{2}, \quad (11.41)$$

and thus the Gaussian wave packet *saturates* the Heisenberg uncertainty principle — it is a state of minimum allowed uncertainty.

- This wavefunction is referred to as a Gaussian wave packet as its probability distribution  $|\psi(x, 0)|^2$  is a Gaussian function of the form

$$\frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2}$$

where  $\mu$  is the mean, and  $\sigma$  is the standard deviation.

- Note: the quantities  $\Delta x$  and  $\Delta k$  we are introducing here are **not** discretisations, but rather the uncertainties in the expectation values of  $x$  and  $k$ !

$$\Delta x^2 = \langle x^2 \rangle - \langle x \rangle^2$$

### 11.6.1 Sampling the Momentum Space

Back when we explored numerical algorithms for approximating the Fourier transform of quantum states, we introduced the **Nyquist theorem**, which states that if a wave function  $\psi(x)$  contains no wavenumber components higher than  $k_{max}$ , then its momentum components can be completely determined by sampling it in discrete steps of

$$\Delta x = \frac{\pi}{k_{max}}.$$

This is an important result, and provides us with a upper bound on our discretisation to ensure that we are properly sampling the momentum space of our initial state. To start with, we need to define our  $k_{max}$ . Since the Gaussian wave packet is a continuous function of  $k$ , there is no value  $k_{max}$  such that  $|\phi(k, 0)| = 0$  for all  $k > k_{max}$ . However, can choose a  $k_{max}$  such that

$$|\phi(k, 0)|^2 \leq \epsilon \quad \forall k > k_{max} \quad (11.42)$$

for some small value  $\epsilon$ , due to our knowledge of the Gaussian distribution ( $|\phi(k, 0)| \rightarrow 0$  for  $k \rightarrow \pm\infty$ ). Substituting this into the expression for the probability distribution of the particle in the momentum space,

$$\begin{aligned} \sigma \sqrt{\frac{2}{\pi}} e^{-2(k_{max} - k_0)^2 \sigma^2} &= \epsilon \\ \Rightarrow k_{max} &= k_0 + \frac{1}{\sqrt{2}\sigma} \sqrt{\ln\left(\frac{\sigma}{\epsilon} \sqrt{\frac{2}{\pi}}\right)}. \end{aligned} \quad (11.43)$$

Therefore, the Nyquist condition becomes

$$\Delta x \leq \frac{\pi}{k_0 + \frac{1}{\sqrt{2}\sigma} \sqrt{\ln\left(\frac{\sigma}{\epsilon} \sqrt{\frac{2}{\pi}}\right)}} \quad (11.44)$$

### 11.6.2 Non-zero Potentials

In the above, we assumed that the Gaussian wave packet was propagating in free-space; that is, there were no nearby potentials. If this is not the case, and we know in advance there *will* be a non-zero potential  $V(x)$  in the vicinity of the wave packet, we can modify the Nyquist condition to take this into account.

Let's assume that the maximum possible kinetic energy gained by the wave packet due to the potential is equal to the largest value of the potential,  $V_{max} = \max_x V(x)$ . The increase in kinetic energy is given by

$$\Delta KE = V_{max} = \frac{1}{2} p_V^2 = \frac{1}{2} k_V^2 \hbar^2 \quad \Rightarrow \quad k_V = \frac{1}{\hbar} \sqrt{2V_{max}}, \quad (11.45)$$

► See Sect. 8.1.2 for a quick refresher on the derivation of the Nyquist sampling theorem.

► Here, we are back to  $\Delta x$  referring to our discretisation of the position grid. Sorry for the (temporary) confusion, order is now restored!

i.e. we can associate a momentum or wavenumber  $k_V$  with this increase in kinetic energy. This acts to increase the possible value of  $k_{max}$  in the system, and therefore we can estimate the new Nyquist condition as

$$\Delta x \leq \pi \left[ k_0 + \sqrt{2V_{max}/\hbar} + \frac{1}{\sqrt{2}\sigma} \sqrt{\ln\left(\frac{\sigma}{\epsilon}\sqrt{\frac{2}{\pi}}\right)} \right]^{-1}. \quad (11.46)$$

### Example 11.1 Wave packet spreading (Fortran)

Solve the Schrödinger equation for a wave packet with no initial momentum.

**Solution:** Let's use the Runge–Kutta fourth order method for this example, using  $\Delta x = 0.1$ , and choosing  $\Delta t = 0.01 \leq \Delta x^2$  for stability of our finite-difference discretisation.

```
program wavepacket
    implicit none
    real(8), parameter :: pi = 4.d0*atan(1.d0)
    complex(8), parameter :: j = complex(0.d0, 1.d0)

    integer :: i, N, steps
    real(8) :: xmin, xmax, dx, tmin, tmax, dt
    complex(8) :: a, g

    complex(8), allocatable :: psi(:, :), k1(:, ), &
                             k2(:, ), k3(:, ), k4(:, )

    ! set the x-grid size and spacing
    dx = 0.1d0; xmin = -10.d0; xmax = 10.d0
    N = (xmax-xmin)/dx + 1

    ! set the time grid size and spacing
    dt = 0.01d0; tmin = 0.d0; tmax=5.d0
    steps = (tmax-tmin)/dt + 1

    ! create the initial wavepacket
    allocate(psi(steps, N), k1(N), k2(N), k3(N), k4(N))
    psi = 0.d0
    do i=1, N
        psi(1, i) = exp(-(xmin+dx*i)**2/4.d0)/((2.d0*pi)**0.25d0)
    end do

    g = -5.d0*j/(4.d0*dx**2)
    a = j/(24.d0*dx**2)

    ! the RK4 4-step method
    do i=1, steps-1
        k1 = f(psi(i, :), N)
```

```

k2 = f(psi(i, :) + 0.5*k1*dt, N)
k3 = f(psi(i, :) + 0.5*k2*dt, N)
k4 = f(psi(i, :) + k3*dt, N)
psi(i+1, :) = psi(i, :) + dt*(k1 + 2*k2 + 2*k3 + k4)/6.
end do

contains
! this function calculates the fourth-order
! finite difference approximation to the
! second derivative.
function f(y, N)
  complex(8), intent(in) :: y(N)
  integer, intent(in) :: N

  complex(8) :: f(N)
  integer :: i

  f(1) = g*y(1) + 16*a*y(2) - a*y(3)
  f(2) = 16*a*y(1) + g*y(2) + 16*a*y(3) - a*y(4)
  f(3) = -a*y(1) + 16*a*y(2) + g*y(3) &
         + 16*a*y(4) - a*y(5)

  do i=4, N-3
    f(i) = -a*(y(i-2)+y(i+2)) &
           + g*y(i) + 16*a*(y(i+1)+y(i-1))
  end do

  f(N-2) = -a*y(N-4) + 16*a*y(N-3) + g*y(N-2) &
            + 16*a*y(N-1) - a*y(N)
  f(N-1) = -a*y(N-3) + 16*a*y(N-2) &
            + g*y(N-1) + 16*a*y(N)
  f(N) = -a*y(N-2) + 16*a*y(N-1) + g*y(N)
end function
end program wavepacket

```

### Example 11.2 Wave packet spreading (Python)

Solve the Schrödinger equation for a wave packet with no initial momentum.

**Solution:** Let's use the Runge–Kutta fourth order method for this example, using  $\Delta x = 0.1$ , and choosing  $\Delta t = 0.01 \leq \Delta x^2$  for stability of our finite-difference discretisation.

```
import numpy as np

# create the x grid
dx = 0.1
x = np.arange(-10, 10+dx, dx)
N = len(x)

# create the initial wave packet
psi0 = np.exp(-x**2/4)/((2*np.pi)**(1/4))

# the potential is zero in this case
V = np.zeros([N])

# construct the 4th order FD matrix
g = -5j/(4*dx**2) - 1j*V
a = 1j/(24*dx**2)

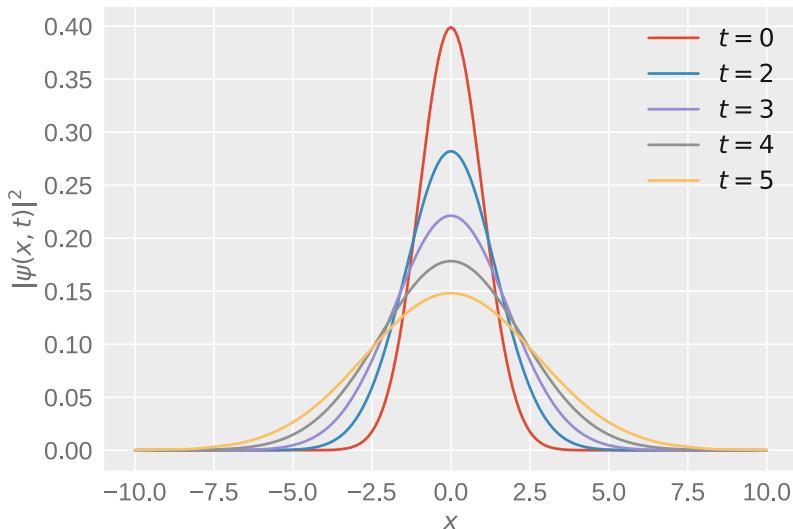
diag = np.diag(g)
off_diag1 = np.diag([16*a]*(N-1), 1) + np.diag([16*a]*(N-1), -1)
off_diag2 = np.diag([-a]*(N-2), 2) + np.diag([-a]*(N-2), -2)

M = diag + off_diag1 + off_diag2

# create the time grid
dt = 0.01
t = np.arange(0, 20+dt, dt)
steps = len(t)

# create an array containing wavefunctions for each step
y = np.zeros([steps, N], dtype=np.complex128)
y[0] = psi0

# the RK4 method
for i in range(0, steps-1):
    k1 = np.dot(M, y[i])
    k2 = np.dot(M, y[i] + k1*dt/2)
    k3 = np.dot(M, y[i] + k2*dt/2)
    k4 = np.dot(M, y[i] + k3*dt)
    y[i+1] = y[i] + dt*(k1 + 2*k2 + 2*k3 + k4)/6
```



Plotting the wavefunction for various values of time  $t$  from the example above, we can see that as time increases, the  $x$  expectation value remains constant, however  $\sigma$  increases and the wave packet spreads in space.

#### Further reading

The Schrödinger equation is generally taught starting with the time-independent case. For an introduction from the perspective of time-dependent systems, Tannor provides a useful textbook targeted at the third year undergraduate/first year graduate level, that introduces concepts such as wave packets and scattering.

- Tannor, D. J. (2007). Introduction to quantum mechanics: a time-dependent perspective. Sausalito, Calif: University Science Books. ISBN 978-1-891389-23-8.

Wang and Midgley solve the time-dependent Schrödinger equation with complex spatial boundary conditions, and they also provide an overview of related computational methods.

- Wang, J. B. and Midgley, S. (1999). Time-dependent approach to electron transport in nano electronic devices, *Physical Review B* 60, 13668-13675.
- Wang, J. B. (2006). Quantum Waveguide Theory, *Handbook of Theoretical and Computational Nanotechnology* (vol 3, chapter 9, American Scientific Publishers, Los Angeles).

A seminal paper in numerical computing, Moler's 'Nineteen Dubious Ways to Compute the Exponential of a Matrix' (and the followup 25 years later) breaks down and compares the various methods of computing the matrix exponential — an indication of just how difficult numerical methods can be! These two papers are an interesting read, although occasionally quite technical. His conclusion 25 years later? The arrival of a new technique that may not be as dubious as the other — Krylov subspace techniques.

- Moler, C., & Van Loan, C. (1978). Nineteen Dubious Ways to Compute the Exponential of a Matrix. *SIAM Review*, 20(4), 801–836.
- Moler, C., & Van Loan, C. (2003). Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later. *SIAM Review*, 45(1), 349.

## Exercises

- P11.1** Using a Taylor series expansion, expand out the following expression

$$e^{-i\hat{H}_0\Delta t/2\hbar}e^{-i\hat{V}(t)\Delta t/2\hbar}e^{-i\hat{H}_0\Delta t/2\hbar} - e^{-i\hat{H}\Delta t/\hbar}$$

to show that the local error of the symmetric Strand split operator method is given by

$$\frac{i\Delta t^3}{24\hbar^3}([[\hat{H}_0, \hat{V}(t)], \hat{H}_0] + 2[[\hat{H}_0, \hat{V}(t)], \hat{V}(t)]) + \mathcal{O}(\Delta t^4).$$

Don't forget to take into account that  $\hat{H}_0$  and  $\hat{V}$  don't necessarily commute!

- P11.2 Free state propagation**

Consider an initial Gaussian wave packet centered at  $x_0 = 25$ , with initial momentum  $k_0 = 2$  and standard deviation  $\sigma = 2$ . Let  $\hbar = m = 1$ .

- (a) Using  $\epsilon = 10^{-10}$ , calculate the Nyquist condition. Use this to choose a reasonable grid spacing  $\Delta x$  for the domain  $0 \leq x \leq 100$ . How many grid points  $N$  are there?
- (b) Show that the boundary conditions on the edge of the grid are satisfied at time  $t = 0$ .
- (c) Using the fourth order finite-difference method in the position space, apply the Euler method in time-steps of  $\Delta t = 0.005$  for  $t = 20$  s. Plot the results at  $t = 0, 7$ , and 20 s — what do you notice?
- (d) Modify part (c) to use an integration algorithm to normalise the wavefunction after each time-step, and plot the results at the same time steps. How do they compare?

- P11.3** Repeat the previous exercise, this time modifying your code to use instead the second-order finite difference discretisation in time.

- (a) Does the value of  $\Delta t$  allow for a stable numerical algorithm? Compare your results to those of the Euler method.
- (b) The exact solution to the Schrödinger equation for the time-evolution of a Gaussian wave packet is

$$\psi(x, t) = \frac{1}{(2\pi)^{1/4}\sqrt{\sigma + it/2\sigma}} \exp\left[-\frac{(x - x_0 - k_0 t)^2}{4\sigma^2 + 2it}\right] e^{ik_0(x - k_0 t/2)}.$$

Use this to produce plots of the absolute and relative error of the second-order finite difference method.

- (c) Note that the standard deviation of the Gaussian wavepacket, and thus the uncertainty in  $x$ , is not independent, but varies with time. Does this correspond to what you see in your numerical simulation? What does this mean for the Nyquist condition?

#### P11.4 Fourier differentiation

Instead of applying the second-order finite difference approximation to calculate the second derivative, we can instead use Fourier differentiation. In this case, we only need to discretise the time-independent Schrödinger equation in time, not space, giving

$$\psi_{j+1}(x) \approx i\Delta t \frac{d^2}{dx^2} \psi_j(x) - 2i\Delta t V_j(x) + \psi_{j-1}(x),$$

where  $\psi_j(x) \equiv \psi(x, t_j)$ .

- (a) At each time-step  $t_j$ , use Fourier differentiation to calculate  $\frac{d^2}{dx^2} \psi_j(x)$ , and then use the centered finite difference approximation to calculate the wavefunction at the next time-step  $\psi_{j+1}(x)$ .
- (b) Time how long it takes this numerical algorithm to calculate the solution  $t = 7$ , and compare it with the result of the second-order finite difference discretisation. How does it compare?

#### P11.5 The potential barrier

A time-independent potential barrier has the form

$$V(x) = \frac{V_0}{\cosh^2[(x - x_0)/\omega]},$$

where  $V_0 = 1.5$ ,  $\omega = 1$ , and  $x_0 = 35$ .

- (a) Plot the potential barrier over the domain  $0 \leq x \leq 100$ .
- (b) Using the Runge–Kutta algorithm, simulate the dynamics of a Gaussian wave packet incident on this potential barrier from the left. Choose the grid discretisation based on the Nyquist sampling theorem, and ensure that at  $t = 0$ , the Gaussian wave packet has no significant overlap with the potential barrier, and that all boundary conditions are met.
- (c) Plot the wavepacket at various times (superimposed on the potential barrier), as it interacts with the potential barrier. How would you describe the dynamics of the system?

- (d) We can define the transmission coefficient as the probability of the incident particle tunnelling through the barrier; likewise, the reflection coefficient is the probability that the particle is reflected. Choosing time  $t$  after the interaction, use numerical integration to calculate the transmission and reflection coefficients, and ensure they add to 1.
- (e) As you increase the height of the barrier  $V_0$ , how do the transmission and reflection coefficients change? What about if you increase the initial momentum  $k_0$ ?

### P11.6 Time-dependent quantum harmonic oscillator

The time-dependent quantum harmonic oscillator is a useful model to describe time-dependent laser sources, and the behaviour of trapped ions. A simple example of a one dimensional time dependent quantum harmonic oscillator is

$$\hat{H}(t) = -\frac{\hbar^2}{2m} \frac{d^2}{dx^2} + \frac{1}{2}m\omega^2(x - \sin t)^2. \quad (11.47)$$

- (a) At time  $t = 0$ , what is the Hamiltonian? Using this as a clue, choose a suitable initial state  $\psi(x, 0)$  for the wavefunction at time  $t = 0$ .
- (b) Using Cartesian coordinates, approximate the Hamiltonian  $\hat{H}(t)$  using the finite-difference approximation to construct the time-dependent matrix  $H_{ij}(t)$  over the grid  $-10 \leq x \leq 10$  for a reasonable value of  $\Delta x$ . Then, using Fortran or Python, write a function that returns this matrix for specified values of  $t$ , in atomic units ( $\hbar = m = \omega = 1$ ).
- (c) Propagate the initial wavefunction under the time-dependent Hamiltonian,

$$\psi(x, t) = \left( \prod_{n=1}^{N-1} e^{-i\hat{H}(n\Delta t)\Delta t/\hbar} \right) \psi(x, 0),$$

using the Chebyshev approximation in small time-steps of  $\Delta t$ . Make sure that you evaluate the current Hamiltonian matrix at each time-step!

- (d) Plot the resulting wavefunction over time. How do the dynamics of the time-dependent quantum harmonic oscillator differ from the time-independent case?

### P11.7 Scattering from a potential wells

In addition to scattering quantum particles from potential barriers, we can also scatter from potential *wells*. For example, consider the

following potential:

$$V(x) = \begin{cases} -1, & 0 \leq x \leq 1 \\ 0, & \text{otherwise} \end{cases}$$

- (a) Choosing a reasonably sized grid and discretisation parameters, use the Runge–Kutta method to propagate an initial Gaussian wave packet towards the potential well. How does the wave packet interact with the potential well?
- (b) Use numerical integration to calculate the transmission  $T$  and reflection  $R$  coefficients at each time step. Plot the  $T$  and  $R$  vs. time; is  $T + R = 1$  satisfied for all values of  $t$ ?
- (c) Use the Chebyshev approximation to propagate the initial wave packet from  $t = 0$  to  $t = t_f$ , where  $t_f$  is such that the wave packet has finished interacting with the potential well. Do this in *one* time-step  $\Delta t = t_f$ . How does this compare to the Runge–Kutta result at time  $t_f$ ?
- (d) *Bonus:* modify the potential to turn it into a **potential cliff**,

$$V(x) = \begin{cases} -1, & 0 \leq x \\ 0, & \text{otherwise} \end{cases}.$$

How does this change the dynamics?



# Chapter 12

## Central Potentials

We tend to be hardwired to think in terms of Cartesian coordinates; this is the coordinate system we are first introduced to in school, and generally our first approach when solving an unknown problem. We can avoid this bias, as we saw earlier, by using basis diagonalisation with a non-Cartesian basis set. However, there are some situations where spherical coordinates are a much better fit, and there is no better example than central potentials — potentials that only depend on radial distance.

An example of a central potential we have already explored is the quantum harmonic oscillator; in spherical coordinates, it is simply  $V(r) = \frac{1}{2}m_e\omega^2r^2$ . While relatively easy to solve in Cartesian coordinates, one complication is that Cartesian coordinates don't take into account spherical symmetry and conservation of angular momentum, producing degenerate states that are difficult to distinguish. By using spherical coordinates, we slightly increase the amount of algebraic manipulation required, with the advantage that it significantly simplifies the final numerical algorithm.

In this chapter, we will explore techniques for solving the time-independent Schrödinger's equation in spherical coordinates, and consider examples such as the Hydrogen atom.

### 12.1 Spherical Coordinates

Spherical coordinates are commonly denoted  $(r, \theta, \phi)$ ; here,  $r$  represents the radial distance from the origin ( $r^2 = x^2 + y^2 + z^2$ ),  $\theta$  is the polar angle (angle between the  $z$ -axis and the position in space), and  $\phi$  is the azimuthal angle (angle between the  $x$  axis and the *projection* of the coordinate in the  $xy$ -plane).

In spherical coordinates, the Laplacian operator takes the following form:

$$\nabla^2 = \frac{1}{r^2} \frac{\partial}{\partial r} \left( r^2 \frac{\partial}{\partial r} \right) + \left[ \frac{1}{r^2 \sin \theta} \frac{\partial}{\partial \theta} \left( \sin \theta \frac{\partial}{\partial \theta} \right) + \frac{1}{r^2 \sin^2 \theta} \frac{\partial^2}{\partial \phi^2} \right], \quad (12.1)$$

and thus the time-independent Schrödinger equation can be written

- To convert Cartesian coordinates to azimuthal and polar angles, we have the following identities:

$$\theta = \cos^{-1} \left( \frac{z}{r} \right)$$

$$\phi = \tan^{-1} \left( \frac{y}{x} \right)$$

It is important to take into account the quadrant when calculating the inverse tangent; to help, NumPy provides the function `np.arctan2(y,x)`, and Fortran provides the intrinsic function `atan2(y,x)`.

- To convert from spherical coordinates to Cartesian coordinates:

$$x = r \sin(\theta) \cos(\phi)$$

$$y = r \sin(\theta) \sin(\phi)$$

$$z = r \cos(\theta)$$

$$\hat{H}\psi_n(r, \theta, \phi) = E_n\psi_n(r, \theta, \phi) \quad (12.2)$$

where

$$\hat{H} = -\frac{\hbar^2}{2m_e r^2} \frac{\partial}{\partial r} \left( r^2 \frac{\partial}{\partial r} \right) - \frac{\hbar^2}{2m_e r^2} \left[ \frac{1}{\sin \theta} \frac{\partial}{\partial \theta} \left( \sin \theta \frac{\partial}{\partial \theta} \right) + \frac{1}{\sin \theta} \frac{\partial^2}{\partial \phi^2} \right] + V(r). \quad (12.3)$$

By using separation of variables, we can see that the solution to the time-independent Schrödinger's equation must be of the form

$$\psi_n(r, \theta, \phi) = R(r)Y(\theta, \phi). \quad (12.4)$$

Since the potential, which completely describes the system under study, is a function only of  $r$ , it follows that the angular part of the solution  $Y(\theta, \phi)$  is independent and identical for all central potentials.

### 12.1.1 Angular Momentum and the Spherical Harmonics

The Hamiltonian above becomes a bit unwieldy to write down repeatedly, but it turns out we can make a convenient substitution. The angular momentum operator in quantum mechanics is defined by  $\hat{L} = \hat{r} \times \hat{p} = -i\hbar\hat{r} \times \nabla$ ; in spherical coordinates, the *square* of this operator is given by

$$\hat{L}^2 = -\frac{\hbar^2}{\sin \theta} \frac{\partial}{\partial \theta} \left( \sin \theta \frac{\partial}{\partial \theta} \right) - \frac{\hbar^2}{\sin^2 \theta} \frac{\partial^2}{\partial \phi^2}. \quad (12.5)$$

Notice that this is simply the term in square brackets in the Laplacian expression — allowing us to write the Laplacian in terms of the operator  $\hat{L}^2$ . The Hamiltonian therefore becomes

$$\hat{H} = -\frac{\hbar^2}{2m_e r^2} \frac{\partial}{\partial r} \left( r^2 \frac{\partial}{\partial r} \right) + \frac{\hat{L}^2}{2m_e r^2} + V(r), \quad (12.6)$$

and its energy eigenstates satisfy the time-independent Schrödinger equation

$$\hat{H} |\psi_n\rangle = E_n |\psi_n\rangle \quad (12.7)$$

with energy eigenvalue  $E_n$ , indexed by the **principal quantum number**  $n$ .

Writing the Hamiltonian in this form also provides some additional insight into the angular momentum of the system. For example, consider the commutator  $[\hat{H}, \hat{L}^2]$ . The Hamiltonian contains three terms — the first (the kinetic energy term) contains an operator dependent on  $r$ , the next contains the operator  $\hat{L}^2$ , and the last,  $V(r)$ , is simply a function of  $r$ . Since  $\hat{L}^2$  contains only operators dependent on the angular components, it follows that  $[\hat{H}, \hat{L}^2] = 0$ . As such, the angular momentum of the system is conserved, and

solutions to the time-independent Schrödinger equation are simultaneously eigenstates of  $\hat{L}^2$ :

$$\hat{L}^2 |\psi_{n\ell}\rangle = \hbar^2 \ell(\ell + 1) |\psi_{n\ell}\rangle, \quad (12.8)$$

where  $\ell$  is the so-called **azimuthal quantum number**.

But this is not all! Quantum theory also tells us that the  $z$ -component of angular momentum,  $\hat{L}_z = -i\hbar \frac{\partial}{\partial \phi}$ , commutes with the angular momentum magnitude;  $[\hat{L}^2, \hat{L}_z] = 0$ . In fact, since  $\hat{L}_z$  contains only an operator dependent on  $\phi$ , it is easy to see that it commutes with the Hamiltonian as well. Therefore, the  $z$ -component of angular momentum is also conserved, and the energy eigenstates are *also* simultaneous eigenstates of  $\hat{L}_z$ :

$$\hat{L}_z |\psi_{n\ell m}\rangle = \hbar m |\psi_{n\ell m}\rangle, \quad (12.9)$$

where  $m$  is the **magnetic quantum number**.

This is a very useful result, as we already have analytical solutions to the eigenstates of the  $\hat{L}^2$  and  $\hat{L}_z$  operators; the **spherical harmonics**:

$$Y_{\ell m}(\theta, \phi) = (-1)^m \sqrt{\frac{(2\ell + 1)(\ell - m)}{4\pi(\ell + m)!}} P_\ell^m(\cos \theta) e^{im\phi}, \quad (12.10)$$

where  $P_\ell^m$  are the **associated Legendre polynomials**, and for a given  $\ell \in \mathbb{N}_0$ ,  $m = -\ell, -\ell + 1, \dots, \ell - 1, \ell$ .

### Coding the spherical harmonics

If you’re using Python, SciPy makes it very easy to use both the associated Legendre polynomials, and even the spherical harmonics, directly in your code; simply import

```
from scipy.special import sph_harm, lpmv
```

and then the spherical harmonics  $Y_{\ell m}$  can be calculated via

```
sph_harm(m, l, theta, phi)
```

and the associated Legendre polynomials  $P_{\ell}^m(x)$  by

```
lpmv(m, l, x)
```

Note that the magnetic quantum number  $m$  (or order) comes *before* the azimuthal quantum number  $\ell$  (the degree) in the argument list.

In Fortran, it is a little more difficult since Fortran does not provide intrinsic functions for calculating the associated Legendre polynomials or the spherical harmonics. While there are external libraries that can fill this gap, such as SHTOOL which provides both a Fortran95 and a Python interface, none are currently as ubiquitous as FFTW or LAPACK. Another alternative is to hard-code a ‘lookup table’ of the spherical harmonic/Legendre functions needed for specific values of  $\ell$ ,  $m$ , and  $\theta$ ; this is an accurate and fast solution for cases when a known small subset of special functions are required. However, this becomes more laborious when the potential values of  $\ell$  and  $m$  grow rapidly.

In such a case, another option is to write a subroutine or function that computes the Legendre polynomials directly, and then uses the expression above to calculate the spherical harmonics. This is non-trivial — if you have access to SHTOOL, we recommend it! It is generally a poor idea to directly code the analytical expression for the Legendre polynomials; they are complicated expressions that involve many function evaluations and potentially even derivatives, and can lead to numerical inaccuracy due to the size of the sums involved. A better approach is to implement the recursion relation satisfied by the Legendre polynomials:

$$(\ell - m)P_{\ell}^m(x) = x(2\ell - 1)P_{\ell-1}^m - (\ell + m - 1)P_{\ell-2}^m(x), \quad (12.11)$$

where the two required starting values are given by

$$P_m^m(x) = (-1)^m (2m - 1)!! (1 - x^2)^{m/2}, \quad P_{m+1}^m(x) = x(2m + 1)P_m^m(x),$$

where  $n!! = n(n - 2)(n - 4) \dots$  denotes the **double factorial**.

## 12.2 The Radial Equation

Now that we have a solution to the angular component of the bound state solution that applies to *every* central potential  $V(r)$ , the problem reduces to determining the radial component of the solution,  $R(r)$ .

Performing separation of variables, and substituting  $\psi_{n\ell m}(r, \theta, \phi) = R(r)Y(\theta, \phi)$  into the Schrödinger equation  $\hat{H}\psi_{n\ell m} = E_n\psi_{n\ell m}$ , we find that the radial equation must satisfy the following second-order differential equation:

$$-\frac{\hbar^2}{2m_e} \frac{1}{r^2} \frac{d}{dr} \left( r^2 \frac{d}{dr} R_{n\ell}(r) \right) + V_{eff}(r)R(r) = E_n R_{n\ell}(r) \quad (12.12)$$

where

$$V_{eff}(r) = V(r) + \frac{\hbar^2 \ell(\ell+1)}{2m_e r^2} \quad (12.13)$$

is the **effective potential**. We have therefore reduced a 3-dimensional partial differential equation with boundary conditions down to a single-dimensional ordinary differential equation with simpler boundary conditions.

As we now have a one-dimensional differential equation with boundary conditions, we can use the techniques from Chap. 9 to solve for the radial solution, significantly reducing the computational complexity of the problem. However, before we can do that, there are still two things we need to address; (a) what are the boundary conditions on  $R(r)$ , and (b) how do we take into account the strange-looking differential operator?

► This is directly analogous to the classical case, where we absorb the centripetal force due to conservation of angular momentum into the central potential.

### 12.2.1 Radial Probability Density

A common approach to simplifying the radial equation is to consider instead the solution for the function  $\mathcal{P}_{n\ell}(r) = rR_{n\ell}(r)$ , so that our solution to the Schrödinger equation is now of the form

$$\psi_{n\ell m}(r, \theta, \phi) = \frac{1}{r} \mathcal{P}_{n\ell}(r) Y_{\ell m}(\theta, \phi). \quad (12.14)$$

If we substitute  $R_{n\ell}(r) = \mathcal{P}_{n\ell}(r)/r$  into the differential operator in the radial equation, we find

$$\begin{aligned} \frac{1}{r^2} \frac{d}{dr} \left( r^2 \frac{d}{dr} R_{n\ell}(r) \right) &= \frac{1}{r^2} \frac{d}{dr} \left( r^2 \frac{d}{dr} \frac{\mathcal{P}_{n\ell}(r)}{r} \right) \\ &= \frac{1}{r^2} \frac{d}{dr} \left( r \frac{d\mathcal{P}_{n\ell}}{dr} - \mathcal{P}_{n\ell}(r) \right) \\ &= \frac{1}{r} \frac{d^2\mathcal{P}_{n\ell}}{dr^2}. \end{aligned} \quad (12.15)$$

Thus the radial equation, in terms of  $\mathcal{P}_{n\ell}(r)$ , becomes

$$\boxed{-\frac{\hbar^2}{2m_e} \frac{d^2\mathcal{P}_{n\ell}}{dr^2} + V_{eff}(r)\mathcal{P}_{n\ell}(r) = E_n \mathcal{P}_{n\ell}(r)} \quad (12.16)$$

### Radial Probability Density

The function  $\mathcal{P}_{n\ell}(r)$ , aside from providing a useful numerical technique to finding a nicer form of the radial equation, also represents the **radial probability density**; that is, the probability distribution describing the probability of finding the particle located at radius  $r$  from the origin. To see that this is indeed the case, consider the probability of finding a particle at location  $[r, r + \delta r]$  for  $\delta r \ll 1$ :

$$\begin{aligned} & \int_r^{r+\delta r} \int_0^\pi \int_0^{2\pi} |\psi_{n\ell m}(r, \theta, \phi)|^2 r^2 \sin \theta \, dr \, d\theta \, d\phi \\ &= \left( \int_0^\pi \int_0^{2\pi} |Y_{\ell m}(\theta, \phi)|^2 \sin \theta \, d\theta \, d\phi \right) \left( \int_r^{r+\delta r} |R(r)|^2 r^2 dr \right) \\ &= \int_r^{r+\delta r} |\mathcal{P}(r)|^2 dr. \end{aligned} \quad (12.17)$$

Note that the angular components integrate to 1 as the spherical harmonic functions are normalised.

Note that  $r$ , the radius is a non-negative quantity, i.e.  $r \geq 0$ . The Hilbert space requires  $\mathcal{P}(r)$  be **square integrable** – it cannot ‘blow up’ to infinity. So, as  $r \rightarrow 0$  and  $r \rightarrow \infty$ ,  $\mathcal{P}(r) \rightarrow 0$ ; this provides us with the boundary conditions on  $\mathcal{P}(r)$ .

## 12.3 The Coulomb Potential

We have already come across one example of a spherically-symmetric central potential; the quantum harmonic oscillator. Another physical system that showcases this same spherical symmetry is the potential in the hydrogen atom. The hydrogen atom contains a single proton in the nucleus; as a result, the bound electron experiences the following **Coulomb** potential:

$$V(r) = k \frac{q_1 q_2}{r} = -\frac{Z q_e^2}{4\pi\epsilon_0 r}, \quad (12.18)$$

where

- $Z$  is the number of protons in the nucleus, for hydrogen,  $Z = 1$ ;
- $q_1 = Z q_e$  is the electric charge of the nucleus; as  $Z = 1$ , so  $q_1 = e$ ;
- $q_2 = -q_e$  is the electric charge of the electron;
- $q_e = 1.602 \times 10^{-19}$  C (Coulombs) is the elementary charge;
- $k = 1/4\pi\epsilon_0$  is the Coulomb constant;
- $\epsilon_0 = 8.85 \times 10^{-12}$  Fm<sup>-1</sup> (farads per metre) is the vacuum permittivity.

Previously in the text, we have used atomic units, where  $\hbar = m_e = q_e = 1$  without much focus on *why* we are allowed to do that. Now that we are working with a physical system, it is important to drill down why exactly we are allowed to ‘set them to one’.

The atomic units we are using in this text are the **Hartree units**, where *all* physical units of measure can be reconstructed in terms of the following fundamental physical quantities: (1) reduced Planck’s constant  $\hbar$ , (2) electron mass  $m_e$ , (3) elementary charge  $q_e$ , (4) Coulomb’s constant  $k$ .

For example, we can refer to the mass of a proton in terms of how many electron masses it contains; a proton contains approximately 1836.15 times more mass than an electron, so  $m_p = 1836.15 m_e$ . In practice, however, it is common to drop the  $m_e$  Hartree unit specifier altogether, and simply say  $m_p = 1836.15$  — in cases like this, it is important to state prior that all subsequent work will use atomic units, and specify the values of the physical quantities used.

We can also express other derived units in terms of the Hartree atomic units; for example, using the four fundamental physical quantities described above, the corresponding unit of length in Hartree atomic units is the **Bohr radius**,

$$a_0 = \frac{\hbar^2}{m_e q_e^2 k} \approx 5.29 \times 10^{-11} \text{ m} \quad (12.19)$$

► You’ll recall from your electromagnetism courses that the Coulomb potential is so-called as particles interacting with one experience a Coulomb force:

$$F = -\frac{dV}{dr} = k \frac{q_1 q_2}{r^2}.$$

► Another common set of atomic units are the Rydberg atomic units, where  $\hbar = 1$ ,  $m_e = 1/2$  and  $q_e = 2$ . In this case, the basic unit of energy is the Rydberg,  $E_r = 13.61 \text{ eV}$ .

and the unit of energy is the Hartree,

$$E_h = \frac{m_e q_e^4}{k^2 \hbar^2} \approx 27.21 \text{eV}. \quad (12.20)$$

So, as a result, in all numerical exercises where atomic units are first assumed, all length scales will be in terms of the Bohr radius, and all energy values will be in terms of the Hartree energy.

Returning to the radial equation, and substituting in the Coulomb potential, we find that

$$-\frac{\hbar^2}{2m_e} \frac{d^2 \mathcal{P}_{n\ell}}{dr^2} + \left[ \frac{\hbar^2 \ell(\ell+1)}{2m_e r^2} - k \frac{Z q_e}{r} - E_n \right] \mathcal{P}_{n\ell}(r) = 0. \quad (12.21)$$

### Asymptotic Results

How would a solution for the radial equation, under the effect of a coulomb potential, behave for large and small  $r$ ? While we could skip ahead and simply attempt a numerical solution, occasionally the numerical solution will fail for reasons that are not quite clear. As a result, it is useful to consider the asymptotic behaviour near the boundary conditions.

- **Case  $r \rightarrow 0$ :** In this regime, the angular momentum term dominates, and as a result,

$$\frac{d^2 \mathcal{P}_{n\ell}}{dr^2} \sim \frac{1}{r^2} \ell(\ell+1) \mathcal{P}_{n\ell}(r). \quad (12.22)$$

If we substitute in a solution of the form  $\mathcal{P}(r) = r^s$ , we find that this admits two solutions,  $\mathcal{P}(r) \sim c_1 r^{\ell+1} + c_2 r^{-\ell}$ . Now, as  $r \rightarrow 0$ ,  $r^{-\ell} \rightarrow 1$  for  $\ell = 0$ , and diverges to infinity for  $\ell \geq 1$ ; therefore we must discard it as unphysical for not satisfying the boundary condition  $\mathcal{P}_{n\ell}(0) = 0$ . Therefore, for  $r \rightarrow 0$ , the asymptotic solution is of the form  $r^{\ell+1}$ . Note that the larger the angular momentum, the larger  $\ell$ , and thus the radial probability density  $\mathcal{P}(r)$  reduces to zero more rapidly.

- **Case  $r \rightarrow \infty$ :** In this regime, it is the constant energy term that will dominate, producing

$$\frac{d^2 \mathcal{P}_{n\ell}}{dr^2} \sim -\frac{2m_e}{\hbar^2} E_n \mathcal{P}_{n\ell}(r), \quad (12.23)$$

with solution  $\mathcal{P}_{n\ell}(r) \sim e^{\pm r \sqrt{-2m_e E_n} / \hbar}$ . To satisfy the other boundary condition, we require  $\mathcal{P}_{n\ell}(r) \rightarrow 0$  as  $r \rightarrow \infty$ , and so we must discard  $e^{r \sqrt{-2m_e E_n} / \hbar}$ , which diverges, as unphysical.

Combining these two cases, we see that the radial equation can be satisfied by an expression of the form

$$\mathcal{P}_{n\ell} \propto r^{\ell+1} e^{-r \sqrt{-2m_e E_n} / \hbar} f_{n\ell}(r), \quad (12.24)$$

where  $f_{n\ell}(r)$  is an unknown polynomial of  $r$ .

- The electron-volt is defined by the change in energy of an electron travelling across a 1V potential ( $1.6 \times 10^{-19} \text{J}$ ). Due to the mass-energy relation  $E = mc^2$ , the electron-volt is commonly used as a unit of mass in high energy physics.

- Even if the numerical solution appears to converge, it is always a good idea to know the asymptotic behaviour of the solution — it could be that your numerical algorithm is converging to an *unphysical* solution, or that there is systematic or large numerical error.

- From the asymptotic solution for  $r \rightarrow \infty$ , we can see that if  $E_n > 0$ , then the quantity under the square root is negative, and

$$\mathcal{P}_{n\ell}(r) \sim e^{\pm i r \sqrt{2m_e E_n} / \hbar},$$

this looks familiar to the plane wave solution of a free particle.

In fact,  $E_n < 0$  for bound states of the Coulomb potential, while  $E_n > 0$  corresponds to free states or scattering states.

## 12.4 The Hydrogen Atom

If we substitute an expression of the form Eq. 12.24 into the radial equation, we find that  $f_{n\ell}(r)$  will satisfy the resulting boundary value problem only for non-negative values of  $n$ , the principal quantum number — producing energy eigenvalue quantization. In fact, it turns out that the differential equation satisfied by  $f_{n\ell}(r)$  is that satisfied by the generalized Laguerre polynomials,  $L_k^a(x)$ , defined by the recurrence relations

$$\begin{aligned} L_0^a(x) &= 1 \\ L_1^a(x) &= 1 + b - x \\ L_{k+1}(x) &= \frac{2k+1+a-x}{k+1}L_k^a(x) - \frac{k+a}{k+1}L_{k-1}^a(x). \end{aligned} \quad (12.25)$$

The full analytical solution for the radial probability density for the orbitals of an atom with  $Z$  charge is given by

$$\mathcal{P}_{n\ell}(r) = \sqrt{\frac{Z(n-l-1)!}{n^2 a_0 [(n+\ell)!]^3}} e^{-Zr/a_0 n} \left(\frac{2Zr}{a_0 n}\right)^{\ell+1} L_{n-\ell-1}^{2\ell+1} \left(\frac{2Zr}{a_0 n}\right), \quad (12.26)$$

where  $n \leq 1$  and  $0 \leq l \leq n-1$ . The energy eigenvalues are then given by

$$E_n = -\frac{Z^2 m_e q_e^4}{2\hbar^2 n^2}, \quad n = 1, 2, 3, \dots \quad (12.27)$$

Straight away, we can note several properties of the eigenvalues of the hydrogen atom;

- While  $\mathcal{P}_{n\ell}(r)$  depends on both  $n$  and  $\ell$ , the energy eigenvalues depend only on  $n$ . As a result, the system displays **degeneracy** — for a given  $n$ , all states  $0 \leq \ell \leq n-1$  have the same energy eigenvalues.
- The **ground state energy** is

$$E_1 = -\frac{m_e q_e^4}{2\hbar^2} = -\frac{1}{2} E_h, \quad (12.28)$$

exactly half the Hartree energy. It follows that the excited states therefore satisfy the relationship

$$E_n = -\frac{1}{2n^2} E_h. \quad (12.29)$$

► The generalised Laguerre polynomial  $L_k^a(x)$  has  $k$  roots or nodes on the domain  $0 \leq x \leq \infty$ .

► Recall that the **full** orbital wavefunction is recovered from the radial probability density by

$$\psi_{n\ell m} = \frac{1}{r} \mathcal{P}_{n\ell}(r) Y_{\ell m}(\theta, \phi).$$

► This verifies our asymptotic analysis on the previous page; the bound state eigenvalues are in fact negative.

► When solving the Schrödinger equation for an electron interacting with a proton, the electron mass is not the only mass present in the system; the proton also has mass. It is more accurate to use the **reduced mass**

$$\mu = \frac{m_1 m_2}{m_1 + m_2},$$

which takes both into account.

However, in this case, since  $m_e \ll m_p$ , to a very good approximation, we find that  $\mu \approx m_e$ .

- ▶ This is an approach that is commonly used when solving hydrogen-like atoms; the hydrogen energy levels are used as initial trial energy values.
- ▶ Note that the radial equation is purely real; therefore, we can assume a real wavefunction in the Fortran code.

### Example 12.1 Hydrogen radial wavefunction

Using the Numerov-Cooley method, solve the radial equation 12.21 in order to determine the radial wavefunctions, and verify the resulting bound state energies match equation 12.27.

**Solution:** Let's use the Numerov algorithm, with the Cooley energy correction formula and matching approach, to solve the radial equation. Unlike the quantum harmonic oscillator, due to the energy degeneracy, we lose the correspondence between energy of the state and the number of nodes — so node counting becomes a much more complicated approach to determine the specific state we are after.

As we are verifying the analytic energy eigenvalue solution, rather than counting the nodes, we will instead use values close to the analytic energies as our starting ‘trial’ energy, to guide the numerical algorithm towards the correct states. This allows us to fix the value of  $n$  we are solving for, and vary  $\ell$  in the potential.

Note that, due to the singularity in the potential at  $r = 0$ , we cannot start our discretised  $r$  grid at 0, as the potential acting at that point is infinite. Instead, we apply our boundary condition  $P_{n\ell}(0) = 0$  at the point  $r = 10^{-15}$ , exceptionally close to the boundary.

#### Fortran solution

```
program hydrogen
    implicit none
    integer :: N, m, nth, l, i
    real(8) :: E, dE, dr
    real(8), allocatable :: r(:), V(:), psi(:)

    ! set the parameters for the algorithm
    nth = 3
    l = 0
    E = -1.d0/(2*nth**2)
    dr = 0.001d0
    N = int((60-1d-15)/dr + 1)

    ! allocate and create the r, psi, and V grids
    allocate(r(N), psi(N), V(N))
    r = [(1.d-15 + dr*i, i=1, N)]
    psi = 0.d0
    V = -1.d0/r + l*(l+1)/(2*r**2)

    ! iterate the Numerov-Cooley method for 1000 max steps
    do i=1, 1000
        call numerov(psi, dr, E, V, N, m)

        ! Cooley energy correction
        dE = cooley_correction(psi, dr, E, V, N, m)
        E = E + dE
```

```
! if dE is smaller than a set precision,
! exit the loop early.
if (abs(dE) < 1e-6) then
    exit
end if
end do

contains
subroutine numerov(psi, dr, E, V, N, m)
    integer, intent(in) :: N
    integer, intent(out) :: m
    real(8), intent(in) :: dr, E, V(N)
    real(8), intent(out) :: psi(N)
    ! local variables
    real(8) :: P(N), psi_out_m, k
    integer :: j

    P = -2.d0*(V-E)
    k = dr**2/12.d0

    ! set boundary conditions
    psi = 0.d0
    psi(2) = 1.d-6
    psi(N-1) = 1.d-6

    ! outward Numerov method
    do j=2, N-1
        psi(j+1) = 2.d0*psi(j)*(1.d0-5.d0*k*P(j)) &
                    - psi(j-1)*(1+k*P(j-1))
        psi(j+1) = psi(j+1)/(1.d0+k*P(j+1))

        ! when the first turning point is found,
        ! exit the loop and set it as the match point
        if (psi(j) < psi(j-1)) then
            m = j+1
            psi_out_m = psi(m)
            exit
        end if
    end do

    ! inward Numerov method to point m+1
    do j=N-1, m+1, -1
        psi(j-1) = 2.d0*psi(j)*(1.d0-5.d0*k*P(j)) &
                    - psi(j+1)*(1+k*P(j+1))
        psi(j-1) = psi(j-1)/(1.d0+k*P(j-1))
    end do

    ! scale outward and inward integration so psi(m)=1
    psi(:m-1) = psi(:m-1)/psi_out_m
    psi(m:) = psi(m:)/psi(m)
```

```

    end subroutine

    function cooley_correction(psi, dr, E, V, N, m)
        integer, intent(in)    :: N, m
        real(8), intent(in)    :: dr, E, V(N), psi(N)
        ! local variables
        real(8)    :: cooley_correction, Y(N)

        Y = (1.d0-2.d0*(dr**2/12.d0)*(V-E)) * psi
        cooley_correction = (psi(m)/sum(abs(psi))**2) &
            *(-0.5*(Y(m+1)-2*Y(m)+Y(m-1))/(dr**2)+(V(m)-E)*psi(m))
    end function
end program

```

Python solution:

```

import numpy as np

# discretise the r grid
dr = 0.001
r = np.arange(1e-15, 60*dr, dr)
N = r.shape[0]

# define the potential V as a function of l
Veff = lambda l: -1/r + l*(l+1)/(2*r**2)
# Hydrogen energy eigenvalues
En = lambda n: -1/(2*n**2)

# initial trial energy and potential
E = En(1)
V = Veff(0)

k = dr**2/12 # a Numerov parameter

# iterate the Numerov-Cooley method for
# 1000 maximum steps
for i in range(1000):
    P = -2*(V-E)

    # set the two initial conditions
    # for the wavefunction at both boundaries.
    psi = np.zeros_like(r)
    psi[1] = 1e-6
    psi[-1] = 1e-6

    # outward integration to point m-1
    for j in range(1, N-1):
        psi[j+1] = 2*psi[j]*(1-5*k*P[j]) - psi[j-1]*(1+k*P[j-1])
        psi[j+1] /= 1+k*P[j+1]

    # when the first turning point is found,

```

```

# set it as the match point
if psi[j] < psi[j-1]:
    m = j+1
    psi_out_m = psi[m]
    break

# inward integration to point m+1
for j in range(N-2, m, -1):
    psi[j-1] = 2*psi[j]*(1-5*k*P[j]) - psi[j+1]*(1+k*P[j+1])
    psi[j-1] /= 1+k*P[j-1]

# scale outward and inward integration so psi(m)=1
psi[:m] /= psi_out_m
psi[m:] /= psi[m]

# Cooley's energy correction formula
Y = (1+k*P)*psi
dE = (psi[m].conj()/np.sum(np.abs(psi)**2)) \
    *(-0.5*(Y[m+1]-2*Y[m]+Y[m-1])/(dr**2)+(V[m]-E)*psi[m])
E += dE

# if dE is smaller than a set precision,
# exit the loop early.
if np.abs(dE) < 1e-6:
    break

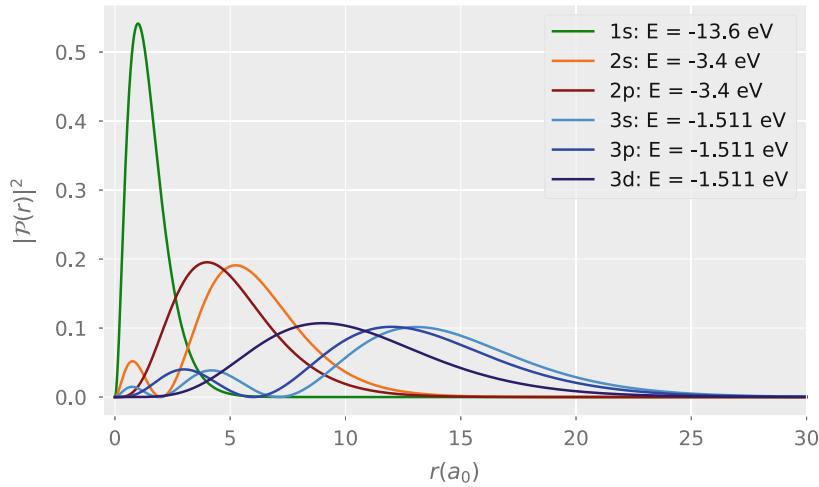
# the final wavefunction and energy in Hartrees
print(E*27.21, psi)

```

Plotting several numerical solutions to the radial probability density,

► Here, we use the naming convention  $s, p, d, f$  corresponding to angular momentum  $\ell = 0, 1, 2, 3$  respectively; a state with  $n = 2$  and  $\ell = 1$  will therefore be denoted  $2p$ .

This notation took hold in early 20th century chemistry, from the German words *scharf* (sharp), *prinzipal* (principle), *diffus* (diffuse) and *fundamental*; so-called due to the appearance of their spectra.



**Figure 12.1** Radial probability densities of the hydrogen atom determined using the Numerov-Cooley method

we can see that as  $n$  increases, the area of maximum probability moves away from the origin. Furthermore, as  $\ell$  decreases, the number of nodes in the radial density increases, and the probability of the electron being observed close to the nucleus increases slightly.

### Fine structure

In the above example and derivations, we found that the energy eigenvalues of the bound states depend only on  $n$ , while the radial densities depend on both  $n$  and  $\ell$  — resulting in energy degeneracy. While this is a result consistent with the Schrödinger equation, it is not physically what we observe.

By using spectroscopy to study the orbital energies of the hydrogen atom, not only can we see the large energy difference in the spectrum due to  $E_n$ , we also see a much ‘finer’ splitting of *each value of  $E_n$* . This is known as **fine structure** splitting, and can be derived theoretically by instead solving the Dirac equation, which takes into account relativistic effects and spin. From this approach, it can be found that the energy eigenvalues also depend on  $j$ , the total angular momentum number (associated with total angular momentum operator  $\hat{J} = \hat{L} + \hat{S}$ , where  $\hat{S}$  is the particle spin operator). Working backwards, a similar result can be determined from the Schrödinger equation by including the corrective spin terms, and using perturbation theory.

This isn’t the end of the story, however! *Additional* splitting of the spectral lines, called **hyperfine splitting**, was also observed, along with

other effects that could not be explained by the Schrödinger equation or the Dirac equation. Instead, these are explained via quantum field theories, such as quantum electrodynamics.

### Non-uniform grids

When solving the radial equation in the above example, we began our radial grid  $r$  at value  $r = 10^{-15}$ , to avoid the singularity in the potential at  $r = 0$ . Furthermore, you'll note that the resulting radial probability densities are much more oscillatory as  $r \rightarrow 0$ ; together, both of these observations suggest a better approach is a **non-uniform, variable** or **adaptive** finite-difference discretisation.

So far, we have considered only uniform grid discretisation — those that involve a constant and unchanging grid spacing  $\Delta x$  across the entire grid. In a non-uniform grid, the value  $\Delta x_j$  is dependent on the location in the grid  $x_j$  — areas where the solution is expected to be more oscillatory or have larger derivatives, will have smaller values of  $\Delta x$ , than areas where the solution is flatter. This allows us to better focus our computational resources, and reduce numerical error. Adaptive grid techniques even do this on the fly, increasing and decreasing the grid spacing where required.

Unfortunately, the finite-difference approaches we have derived generally only apply for uniform grid spacing — if we were to use them un-modified for varying  $\Delta x_j$ , we would find that errors quickly accumulate. Instead, we need to consider a coordinate transformation on the non-uniform grid that allows us to rewrite the derivatives in terms of a uniform grid.

Consider the non-uniform grid defined by

$$r_j \equiv r(s_j) = r_0(e^{sj} - 1),$$

where  $s_j = j\Delta s$  is a *uniform* grid. Then, the second order derivative of  $\mathcal{P}(r)$  becomes:

$$\frac{d^2\mathcal{P}(r)}{ds^2} = \frac{1}{r'(s_j)^2} \left( \frac{d^2\mathcal{P}(r(s))}{ds^2} - r''(s_j) \frac{d\mathcal{P}(r(s))}{dr} \right), \quad (12.30)$$

where

$$\frac{d\mathcal{P}(r)}{dr} = \frac{1}{r'(s_j)} \frac{d\mathcal{P}(r)}{ds}. \quad (12.31)$$

The right hand side can now be expressed using the same finite difference approach we have developed for uniform grids, assuming the grid transformation  $r(s_j)$  has a well defined double derivative.

**Further reading**

When studying the electronic structure of atoms and molecules, we enter the realm of atomic physics or quantum chemistry. Most introductory quantum mechanics, atomic physics, and quantum chemistry texts discuss central potentials, the hydrogen atom, and quantisation of angular momentum. A few we recommend include

- Griffiths, D. J. (2017). *Introduction to quantum mechanics* (Second edition). Cambridge: Cambridge University Press. ISBN 978-1-107-17986-8
- Levine, I. N. (2014). *Quantum chemistry* (Seventh edition). Boston: Pearson. ISBN 978-0-321-80345-0

## Exercises

**P12.1** The 2D quantum Harmonic oscillator is another example of a central potential, where  $r^2 = x^2 + y^2$ , and  $V(r) = m_e\omega r^2/2$ .

- Using the Numerov-Cooley method with node matching, solve the radial equation for  $r \in [0, 5]$ ,  $\Delta r = 0.01$ , for the lowest four eigenstates and their corresponding energies.
- Normalise the radial probabilities using numerical integration, and plot their probability distributions.
- Calculate the corresponding wavefunctions  $p\psi_{nlm}(r, \theta, \phi) = \mathcal{P}_{nl}(r)Y_{lm}(\theta, \phi)$ , and plot  $|\psi(r, \theta, \phi)|^2$  in 3D space. How do your results compare to the previous chapter, where the time-independent Schrödinger equation was solved using Cartesian coordinates?

**P12.2** For each principal quantum number  $n$ , how many degenerate energy eigenvalues  $E_n$  are there for the Hydrogen atom?

**P12.3** (a) Consider the ground state of the hydrogen atom  $n = 1$ ,  $\ell = 0$  (also referred to as the 1s orbital). By differentiating the analytic expression for the ground state radial density probability  $\mathcal{P}_{10}(r)$ , determine the maximum of the density. At what value of  $r$  does this maximum occur? This is the most probable radius of finding the electron in this orbital.  
(b) Another quantity we can consider is the radial expectation value,

$$\int_0^\infty \mathcal{P}_{nl}(r)^* r \mathcal{P}_{nl}(r) dr.$$

Perform this integration — how does the expected radius compare to the most probable radius? If there is a difference, can you explain it?

**P12.4** Using the Numerov-Cooley method, solve the radial equation in order to determine the radial wavefunctions of the Hydrogen atom. You may modify the code from Example 12.1.

- Using a numerical integration technique, normalise the resulting radial probability densities. You should get results similar to Fig. 12.1.
- Double check that the most probable radius and the expected radius of the ground state match your results from Problem 12.3.

- (c) Compare the numerical results to the analytic expression for the radial probability densities, (Eq. 12.26). How does the relative error change as a function of radius  $r$ ? Does it differ significantly for different values of  $n$  and  $\ell$ ?
- (d) Try increasing the initial value of the discretised radial grid from the current value of  $r_0 = 10^{-15}$ . How does the error change as this increases? Is there a value where a numerical solution does not converge? If so, which value of  $\ell$  does it affect first?

**P12.5** Rather than performing the Numerov-Cooley method for the radial densities of hydrogen by starting from an initial trial energy, modify the program so that it accepts a bracketed energy range  $[E_{min}, E_{max}]$ , and bisects this energy until the correct number of nodes have been found. Once the correct number of nodes has been found, determine the exact energy of the state using Cooley's energy correction formula.

How does this method compare to the previous approach, where you simply guess trial energies close to the expected energy?

*Hint:* the number of nodes in a state is a function of both  $n$  and  $\ell$ . Since the value of  $\ell$  is fixed by the potential we choose, the node count gives an indication of the value of  $n$ .

**P12.6** An alternative to the Numerov-Cooley method is the direct finite-difference matrix approach.

1. Using the fourth-order finite difference approximation, write down a system of linear equations approximating the radial equation 12.21.
2. Using an eigenvector algorithm, or a library such as SciPy or LAPACK, solve for the eigenvalues and eigenvectors of the corresponding matrix equation.
3. Plot the resulting radial probability densities and the corresponding energy eigenvalues — how do they compare to the results obtained via the Numerov-Cooley approach?
4. Time the finite-difference matrix method, and compare it to the Numerov-Cooley method. Which method is less computationally intensive, or converges to a solution faster?

### P12.7 Non-uniform discretisation

In order to increase the accuracy and speed of convergence of the Numerov-Cooley method for the Coulomb potential, it is beneficial

to consider a non-uniform grid, with smaller spacing closer to the origin  $r = 0$ . A common non-uniform grid spacing is given by

$$r_j \equiv r(s_j) = r_0(e^{s_j - 1}),$$

where  $s_j = j\Delta s$  is a uniform grid with spacing  $\Delta s$ .

- (a) Using the chain rule, determine expressions for  $d\mathcal{P}/dr$  and  $d^2\mathcal{P}/dr^2$  in terms of  $r'(s_j)$ ,  $r''(s_j)$ ,  $d\mathcal{P}/ds$ , and  $d^2\mathcal{P}/ds^2$ .
- (b) Take the finite difference approximation to this result, and substitute it into the radial equation, along with  $r(s_j)$ ; you should now have a radial equation in terms of  $\mathcal{P}_{n\ell}$  and  $s$ .
- (c) Using an appropriate numerical method, solve for the energy eigenvalues  $E_n$  and the radial probability densities  $R_{n\ell}(s)$ .
- (d) Using the inverse mapping  $s(r) = \ln\left(1 + \frac{r}{r_0}\right)$ , transform the radial probabilities to determine  $R_{n\ell}(r)$ , and plot the probability densities over  $r$ . How do your results compare to using a uniform grid?



## Chapter 13

# Multi-electron Systems

When studying the hydrogen atom, we are able to easily determine not only numerical solutions to the Schrödinger equation, but also analytical solutions. What happens when we have multiple electrons interacting with a nucleus, such as for the helium or lithium atom? We now have a wavefunction that is a function of the position of each electron,  $\psi(\mathbf{r}_1, \dots, \mathbf{r}_M)$ . However, not only do we need to consider the interaction of each electron with the nucleus, we must also consider the interactions of the electrons with *each other*. This complicates matters considerably, and, except for in a few specific cases, we cannot solve the resulting Schrödinger equation analytically, or even numerically!

One solution to this problem is the notion of *self-consistent field theories*. In a self-consistent field theory, we approximate the wavefunction as the product of independent electron wavefunctions, and then solve the Schrödinger equation for each independent case. The effective potential interacting with each electron is calculated by taking into account the charge distributions of the remaining electrons and the nucleus — essentially, reducing the many-body problem down to  $M$  single-body problems.

In this chapter, we will introduce the concepts and theory behind one of the most common self-consistent field theory used in quantum mechanics — the Hartree–Fock method. We will then apply this approach to the Lithium atom, as well as to quantum dots.

### 13.1 The Multi-electron Schrödinger Equation

Consider a multi-electron wavefunction

$$\psi(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_M), \quad (13.1)$$

where  $\mathbf{r}_i$  represents the coordinates of electron  $i$ . The Hamiltonian of each electron will contain the following terms:

- the **electron kinetic energy**, given by

$$-\frac{\hbar^2}{2m_e} \nabla_i^2,$$

where the subscript on the Laplacian indicates the coordinate system  $\mathbf{r}_i$  it acts on;

- an attractive **electron-nucleus Coulomb potential**,

$$V(\mathbf{r}_i) = -\frac{Zq_e^2}{4\pi\epsilon_0 r_i},$$

where  $r_i = |\mathbf{r}_i|$ ;

- a repulsive **electron-electron Coulomb potential**,

$$V_{12}(r_{ij}) = \frac{q_e^2}{4\pi\epsilon_0 |\mathbf{r}_i - \mathbf{r}_j|}.$$

Combining this into a single Hamiltonian describing the full wavefunction,

$$\hat{H} = -\sum_i \frac{\hbar^2}{2m_e} \nabla_i^2 - \sum_i \frac{Zq_e^2}{4\pi\epsilon_0 r_i} + \sum_i \sum_{j>i} \frac{q_e^2}{4\pi\epsilon_0 |\mathbf{r}_i - \mathbf{r}_j|}. \quad (13.2)$$

Note that the last summation in the electron-electron potential is from  $j = i + 1$  to  $M$ , the number of electrons; this is to avoid counting the same pairs of electrons multiple times. This Hamiltonian fully describes the multi-electron system.

If we were to neglect the electron-electron potentials, the Hamiltonian would simply reduce to the sum of multiple hydrogen-like Hamiltonians, which we know how to solve. Each electron would in this approximation satisfy a hydrogen-like orbital, and the total energy would be the sum of the hydrogen energy levels. Unfortunately, this is an exceptionally poor approximation; we somehow need a way of incorporating the electron-electron potential, in some form, in our model.

## 13.2 The Hartree Approximation

The goal of self-consistent field theories is to approximate a many-body Hamiltonian by multiple single-body problems; to do that in our case, a reasonable guess or *ansatz* for the wavefunction is

$$\psi(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_M) = \phi_1(\mathbf{r}_1)\phi_2(\mathbf{r}_2)\cdots\phi_M(\mathbf{r}_M). \quad (13.3)$$

where  $\phi_i(\mathbf{r}_i)$  represents the single-electron wavefunction of electron  $i$ , normalised such that

$$\langle \phi_i(\mathbf{r}_i)|\phi_j(\mathbf{r}_j) \rangle = \iint \phi_i(\mathbf{r}_i)^* \phi_j(\mathbf{r}_j) d\mathbf{r}_i d\mathbf{r}_j = \delta_{ij}. \quad (13.4)$$

For convenience, we can combine the kinetic energy and electron-nucleus potential terms in the Hamiltonian for each electron, into a single  $\hat{H}_i$  term, representing the hydrogen like terms;

$$\hat{H}_i = -\frac{\hbar^2}{2m_e} \nabla_i^2 - \frac{Zq_e^2}{4\pi\epsilon_0 r_i}, \quad (13.5)$$

and the Hamiltonian can now be written

$$\hat{H} = \sum_i \hat{H}_i + \sum_i \sum_{j>i} V_{12}(r_{ij}). \quad (13.6)$$

In order to uncouple the electron–electron Coulomb term, we need to return to an old friend — the variational principle.

### Applying the Variational Principle

Let's calculate the energy expectation value  $E = \langle \hat{H} \rangle$  of the above Hamiltonian, using the independent wavefunction *ansatz*.

$$\begin{aligned} E &= \langle \phi_1(\mathbf{r}_1)\phi_2(\mathbf{r}_2) \cdots \phi_M(\mathbf{r}_M) | \hat{H} | \phi_1(\mathbf{r}_1)\phi_2(\mathbf{r}_2) \cdots \phi_M(\mathbf{r}_M) \rangle \\ &= \int \cdots \int \phi_1(\mathbf{r}_1)^* \cdots \phi_M(\mathbf{r}_M)^* \hat{H} \phi_1(\mathbf{r}_1) \cdots \phi_M(\mathbf{r}_M) d\mathbf{r}_1 \cdots d\mathbf{r}_M. \end{aligned} \quad (13.7)$$

Substituting in the expression for  $\hat{H}$ , we find that

$$\begin{aligned} E &= \sum_i \int \phi_i(\mathbf{r}_i)^* \hat{H}_i \phi_i(\mathbf{r}_i) d\mathbf{r}_i \\ &\quad + \sum_i \sum_{j>i} \iint \phi_i(\mathbf{r}_i)^* \phi_j(\mathbf{r}_j)^* V_{12}(r_{ij}) \phi_i(\mathbf{r}_i) \phi_j(\mathbf{r}_j) d\mathbf{r}_i d\mathbf{r}_j \end{aligned} \quad (13.8)$$

as  $\hat{H}_i$  and  $V_{12}(r_{ij})$  act only on states  $\{\phi_i(\mathbf{r}_i)\}$  and  $\{\phi_i(\mathbf{r}_i), \phi_j(\mathbf{r}_j)\}$  respectively — the remaining integrals simply integrate to 1 as the single-electron wavefunctions are normalised.

Now, recall the variational principle (Sect. 10.3); states which are a stationary point of the energy expectation value must satisfy the Schrödinger equation. We can use the variational principle to find the equation that must be satisfied by the above states, for a small perturbation  $\delta\phi_i$  in the orbital functions  $\phi_k(\mathbf{r}_i)$ ; note that we will restrict ourselves to perturbations that retain normalization, that is

$$\langle \phi_i(\mathbf{r}_i) + \delta\phi_i | \phi_i(\mathbf{r}_i) + \delta\phi_i \rangle = 1. \quad (13.9)$$

Introducing this small perturbation into the above equation and integrating by parts, the following equation, as well as its complex conjugate, must be satisfied for each value of  $i$ :

$$\int \delta\phi_i(\mathbf{r}_i)^* \left[ \hat{H}_i \phi_i(\mathbf{r}_i) + \sum_{j \neq i} \int \phi_j(\mathbf{r}_j)^* V_{12}(r_{ij}) \phi_j(\mathbf{r}_j) \phi_i(\mathbf{r}_i) d\mathbf{r}_j - E_i \phi_i(\mathbf{r}_i) \right] d\mathbf{r}_i = 0. \quad (13.10)$$

► Note that, each integral over  $d\mathbf{r}_i$  in this section is actually a **triple integral** or a **volume integral**, since  $\mathbf{r}_i = (x_i, y_i, z_i)$  or  $\mathbf{r}_i = (r_i, \theta_i, \phi_i)$ , depending on your coordinate system.

For this equation to be satisfied for all arbitrary perturbations  $\delta\phi_i(\mathbf{r}_i)$ , the bracketed term must be zero — this leads to

$$\hat{H}_i\phi_i(\mathbf{r}_i) + \sum_{j \neq i} \int \phi_j(\mathbf{r}_j)^* V_{12}(r_{ij}) \phi_j(\mathbf{r}_j) \phi_i(\mathbf{r}_i) d\mathbf{r}_j = E_i \phi_i(\mathbf{r}_i). \quad (13.11)$$

Substituting in the values of the operators, this can also be written in the form

$$\boxed{-\frac{\hbar^2}{2m_e} \nabla_i^2 \phi_i(\mathbf{r}_i) - \frac{Zq_e^2}{4\pi\epsilon_0 r_i} \phi_i(\mathbf{r}_i) + \int \frac{q_e^2 \sum_{j \neq i} |\phi_j(\mathbf{r}_j)|^2}{4\pi\epsilon_0 |\mathbf{r}_i - \mathbf{r}_j|} d\mathbf{r}_j \phi_i(\mathbf{r}_i) = E_i \phi_i(\mathbf{r}_i)}. \quad (13.12)$$

This is known as the **Hartree equation**, and when solved for each  $i$ , provides the wavefunctions  $\phi_i(\mathbf{r}_i)$  for each orbital, and corresponding energies  $E_i$ . It looks remarkably similar to the Schrödinger equation — we still have the kinetic energy term, as well as the attractive electron-nucleus term. Now, however the intractable electron-electron Coulomb term has been replaced by a slightly modified form.

In fact, it looks similar to the classical equation of the potential from a charge distribution  $\rho(\mathbf{r})$ ,

$$V(\mathbf{r}) = \int \frac{\rho(\mathbf{r}')}{4\pi\epsilon_0 |\mathbf{r} - \mathbf{r}'|} d\mathbf{r}'. \quad (13.13)$$

Continuing this analogy, we can define the **Hartree potential**

$$\boxed{V_H(\mathbf{r}_i) = \int \frac{\rho_i(\mathbf{r}_j)}{4\pi\epsilon_0 |\mathbf{r}_i - \mathbf{r}_j|} d\mathbf{r}_j}, \quad (13.14)$$

where  $\rho_i(\mathbf{r}_j) = q_e^2 \sum_{j \neq i} |\phi_j(\mathbf{r}_j)|^2$  is the **charge density** of all remaining electrons. Thus, the Hartree potential is the repulsive Coulomb potential experienced by electron  $i$  due to the presence of all other electrons.

### 13.2.1 Energy Eigenvalues

It was stated above that the eigenvalues  $E_i$  of the Hartree equation are the eigenvalues of the orbital  $\phi_i(\mathbf{r}_i)$ . But how do we know for sure that this is the case? You're right to be cautious! It turns out that, in general, they are **not** the orbital energies, but in practicality they are usually assumed to be equivalent to the orbital energies. This is because of well-known theorem in quantum chemistry known as **Koopman's theorem**.

**Theorem 13.1.** (Koopman's theorem) *The energy required to remove an electron from orbital  $k$  is given by Hartree eigenvalue  $E_k$ .*

- Tjalling Charles Koopmans (1910–1985) did pivotal work in physics and quantum chemistry, but surprisingly won the Nobel prize in economics, for his work on optimal use of resources.

Note that this is sometimes referred to as Koopman's approximation, as it was derived in the context of the Hartree–Fock method itself. Nevertheless, it provides a reasonable justification for referring to  $E_i$  as the orbital bound energies.

What about the total energy of the atom? You might assume that we can simply sum all values of  $E_k$ , but there is a slight subtlety that means this is not the case. Consider the Hartree equation (13.11); by contracting both sides with  $\langle \phi_i(\mathbf{r}_i) |$  (i.e., integrating by  $\int d\mathbf{r}_i \phi_i(\mathbf{r}_i)^*$ ), and summing over all  $i$ , we get

$$\begin{aligned} \sum_i E_i &= \sum_i \int \phi_i(\mathbf{r}_i)^* \hat{H}_i \phi_i(\mathbf{r}_i) d\mathbf{r}_i \\ &\quad + \sum_i \sum_{j \neq i} \iint \phi_i(\mathbf{r}_i)^* \phi_j(\mathbf{r}_j)^* V_{12}(r_{ij}) \phi_i(\mathbf{r}_i) \phi_j(\mathbf{r}_j) d\mathbf{r}_i d\mathbf{r}_j. \end{aligned} \quad (13.15)$$

Now, compare this with Eq. 13.8, where we calculated the expected energy value of the multi-electron Hamiltonian. While looking very similar, they differ in the summation term over the electron–electron potential — above we have  $\sum_{j \neq i}$ , whereas the earlier expression contains  $\sum_{j > i}$ . It is easy to see that

$$\sum_{j > i} f(i, j) = \frac{1}{2} \sum_{j \neq i} f(i, j);$$

on the left hand side, we are only considering pairs  $(i, j)$  where  $j$  is larger, whereas the right hand side sums over *all* pairs, counting both  $(i, j)$  and  $(j, i)$ .

We need to take this double counting of electron pairs into account if we want to express the total energy as a sum of the Hartree orbital energies. Therefore, the total or expected energy of the total atom is given by

$$E = \langle \hat{H} \rangle = \sum_i E_i - \sum_i \sum_{j > i} \iint \phi_i(\mathbf{r}_i)^* \phi_j(\mathbf{r}_j)^* V_{12}(r_{ij}) \phi_i(\mathbf{r}_i) \phi_j(\mathbf{r}_j) d\mathbf{r}_i d\mathbf{r}_j. \quad (13.16)$$

Or, using Dirac notation, this can be written in terms of the expectation energy of the electron–electron Coulomb potential, summed over all pairs

$$E = \langle \hat{H} \rangle = \sum_i E_i - \sum_i \sum_{j > i} \langle \phi_i(\mathbf{r}_i) \phi_i(\mathbf{r}_j) | V_{ij} | \phi_i(\mathbf{r}_i) \phi_i(\mathbf{r}_j) \rangle. \quad (13.17)$$

► Here, we use the shorthand  $V_{ij} \equiv V_{12}(r_{ij})$  to refer to the electron–electron Coulomb potential.

### 13.2.2 Self-consistency

You may be wondering about a slight contradiction in the Hartree equation — while it looks similar to the Schrödinger equation, the Hartree potential is no simple potential; to calculate the Hartree potential acting on one electron,

you need the orbital wavefunction solutions of all the remaining electrons! So how do we even begin solving the Hartree equation?

This is where the ‘self-consistent’ part of the self-consistent mean field theory designation comes into play. As you’re by now familiar with, we solve the Hartree equation in a iterative procedure.

**(1) Start with trial wavefunctions.**

Choose some initial trial electron wavefunctions  $\phi_1^{(0)}(\mathbf{r}_1), \dots, \phi_M^{(0)}(\mathbf{r}_M)$  — these can be chosen from any set of basis functions that suit the problem at hand.

For example, the plane wave basis is well-suited to periodic boundary conditions, the Gaussian basis set works well for molecular structure, and the hydrogen orbital wavefunctions work well for atomic modelling.

**(2) Solve the Hartree equation.**

Using these wavefunctions, calculate all Hartree potentials  $V_H(\mathbf{r}_i)$  acting on each wavefunction, and solve the Hartree equation using an appropriate numerical method. These corresponding solutions are denoted  $\phi_1^{(1)}(\mathbf{r}_1), \dots, \phi_M^{(1)}(\mathbf{r}_M)$ .

**(3) Check for consistency.** For the Hartree potential to be self-consistent, the electron wavefunction solutions should be identical to the input trial wavefunctions  $\phi_i^{(0)}(\mathbf{r}_i)$ . If this is not the case, use the solutions  $\phi_i^{(1)}(\mathbf{r}_i)$  as your new trial wavefunctions, and return to step 1.

**(4)** Repeat until the Hartree equation is self-consistent; that is, for all  $i$ , we have

$$\begin{aligned} |V_H^{(n+1)}(\mathbf{r}_i) - V_H^{(n)}(\mathbf{r}_i)| &\leq \epsilon \\ |\phi_i^{(n+1)}(\mathbf{r}_i) - \phi_i^{(n)}(\mathbf{r}_i)| &\leq \epsilon \\ |E_i^{(n+1)} - E_i^{(n)}| &\leq \epsilon \end{aligned}$$

for some  $\epsilon \ll 1$ .

Thus, with a self-consistent mean field approximation, we know we have achieved convergence when the Hartree equation for a given Hartree potential produces states, which, in turn, reproduce the same Hartree potential. The system is self-consistent.

### 13.3 The Central Field Approximation

In Chap. 12, we saw that Hamiltonians containing only central potentials of the form  $V(r)$  allow us to massively simplify the differential equation under investigation. As the potential terms depend only on the radial distance from the origin, separation of variables allows us to write the wavefunction of a single electron orbital in the form

$$\phi_i(\mathbf{r}_i) = \frac{1}{r_i} \mathcal{P}_{n\ell}(r_i) Y_{n\ell}(\theta_i, \phi_i),$$

where  $r_i = |\mathbf{r}_i|$  is the radius from the origin,  $\mathcal{P}_{n\ell}$  is the radial probability density, and  $Y_{n\ell}(\theta_i, \phi_i)$  the spherical harmonic functions. This is convenient for a multitude of reasons;  $Y_{n\ell}(\theta_i, \phi_i)$  are known analytically for central potentials, which do not depend on  $r$ , so we need only solve a (simpler) single-variable PDE for  $\mathcal{P}_{n\ell}(r_i)$ . Furthermore, this easily quantises the wavefunction according to three quantum numbers — the principal quantum number  $n$  associated with the energy  $\hat{H}$ , the azimuthal quantum number  $\ell$  associated with angular momentum magnitude  $\hat{L}^2$ , and magnetic quantum number  $m$  associated with the  $z$ -component of angular momentum  $\hat{L}_z$ .

Ideally, we would like to take a similar approach with the Hartree Eq. (13.12); as we can see, the kinetic energy and the nucleus-electron Coulomb terms depend only on  $r_i$ , and thus satisfy the criterion for a central potential. Unfortunately, the third term — the Hartree potential — does *not* depend solely on radial distance from the origin except in special cases, sabotaging our ability to perform the separation of variables.

#### Closed Shells

One of these special cases is when the atom contains a **closed shell**. Let's consider a three electron example, where the orbital  $\phi_j$  corresponding to quantum numbers  $n$  and  $\ell$  is fully occupied by two electrons with opposite spin. We refer to a fully occupied orbital as a **closed shell**.

For an electron occupying the next excited state  $\phi_i$ , the Hartree potential is given by

$$V_H(\mathbf{r}_i) = \int \frac{2q_e^2 \sum_j |\phi_j(\mathbf{r}_j)|^2}{4\pi\epsilon_0 |\mathbf{r}_i - \mathbf{r}_j|} d\mathbf{r}_j, \quad (13.18)$$

where we have a factor of two due to summing over both electrons in the closed shell. Now, let's approximate the closed shell via separation of variables, as we did with hydrogen:

$$\phi_0(\mathbf{r}_j) = \frac{1}{r_j} \mathcal{P}_{n\ell}(r_j) Y_{n\ell}(\theta_j, \phi_j). \quad (13.19)$$

As  $n$  and  $\ell$  are fixed for the two electrons, the sum over all orbital functions  $\sum_j$  reduces to a sum over our free quantum number  $m$ , where  $m \in [-\ell, \ell]$ .

Therefore, the Hartree potential becomes

$$V_H(\mathbf{r}_i) = \int \frac{1}{r_j^2} |\mathcal{P}_{n\ell}(r_j)|^2 \frac{q_e^2 \sum_m |Y_{\ell m}(\theta_j, \phi_j)|^2}{2\pi\epsilon_0 |\mathbf{r}_i - \mathbf{r}_j|} d\mathbf{r}_j. \quad (13.20)$$

We still have an angular dependence in the spherical harmonics — however, the spherical harmonics satisfy a well-known result known as the **addition theorem**, namely

$$\sum_{m=-\ell}^{\ell} |Y_{\ell m}(\theta_j, \phi_j)|^2 = \frac{2\ell + 1}{4\pi}. \quad (13.21)$$

Thus, substituting this into Eq. 13.20,

$$V_H(\mathbf{r}_i) = (2\ell + 1) \int \frac{q_e^2 |\mathcal{P}_{n\ell}(r_j)|^2}{8\pi^2 \epsilon_0 r_j^2 |\mathbf{r}_i - \mathbf{r}_j|} d\mathbf{r}_j. \quad (13.22)$$

Note that there are no longer any angular components, the Hartree potential now only depends on the radius, and thus the Hartree potential in the case of a closed shell is *exactly* a central potential!

In general, however, we can't always guarantee we will be working with closed shells. Instead, there is a further approximation we can make to recover the spherical symmetry of a central potential — we can replace the Hartree potential for each electron by its **spherical or angular average**, by averaging the potential over a solid angle  $\Omega$ ,

$$\langle V_H \rangle(r_i) = \frac{1}{4\pi} \int V_H(\mathbf{r}_i) d\Omega = \frac{1}{4\pi} \iiint \frac{\rho_i(\mathbf{r}_j)}{4\pi\epsilon_0 |\mathbf{r}_i - \mathbf{r}_j|} d\mathbf{r}_j \sin \theta_i d\theta_i d\phi_i. \quad (13.23)$$

This is known as the **central field approximation**, as we are approximating a potential with angular dependence by its spherical average. While the Hartree potential does have angular dependence in the case of open shells, even then the central field approximation remains a reasonably accurate approximation, and is commonly used.

### 13.3.1 Gauss's Law and the Charge Density

When it comes to numerically implementing the central field approximation, an easier approach is to apply it to the charge density before calculating the potential. Under the central field approximation, the charge density becomes spherically symmetric, depending only on  $r_j$ :

$$\langle \rho \rangle(r_j) = \frac{1}{4\pi} \int \rho(\mathbf{r}_j) d\Omega = \frac{1}{4\pi} \int_0^{2\pi} \int_0^\pi \rho(\mathbf{r}_j) \sin \theta_j d\theta_j d\phi_j. \quad (13.24)$$

Substituting this into the expression for the charge density,

$$\rho_i(\mathbf{r}_j) = q_e^2 \sum_{j \neq i} |\phi_j(\mathbf{r}_j)|^2, \quad (13.25)$$

where we are using separation of variables  $\phi_j(\mathbf{r}_j) = \frac{1}{r_j} \mathcal{P}_j(r_j) Y_{\ell m}(\theta_j, \phi_j)$ . After normalisation of the spherical harmonics, this integral becomes

$$\begin{aligned} \langle \rho \rangle(r_j) &= \sum_{j \neq i} \frac{q_e^2}{4\pi r_j^2} |\mathcal{P}_j(r_j)|^2 \int_0^{2\pi} \int_0^\pi Y_{\ell m}(\theta_j, \phi_j) \sin \theta_j d\theta_j d\phi_j \\ &= \sum_{j \neq i} \frac{q_e^2}{4\pi r_j^2} |\mathcal{P}_j(r_j)|^2. \end{aligned} \quad (13.26)$$

► Here, we use the shorthand  $\mathcal{P}_j(r_j)$  to refer to a particular  $n\ell$  configuration.

Since we now have a spherically symmetric charge density, we can use **Gauss's Law** to calculate the Hartree potential; the potential at radius  $r$  is simply given by considering the charge enclosed  $Q_{enc}$  by a spherical Gaussian surface at point  $r$ :

$$\langle V_H \rangle(r_i) = \frac{Q_{enc}(r_i)}{4\pi\epsilon_0 r_i}, \quad (13.27)$$

where  $Q_{enc}$  is determined by integrating the charge density over the volume of the Gaussian surface:

$$\begin{aligned} Q_{enc}(r_i) &= \iiint_V \langle \rho \rangle(r_j) d\mathbf{r}_j \\ &= \int_0^{2\pi} \int_0^\pi \int_0^{r_i} \langle \rho \rangle(r_j) r_j^2 \sin \theta dr_j d\theta_j d\phi_j \\ &= 4\pi \int_0^{r_i} \langle \rho \rangle(r_j) r_j^2 dr_j. \end{aligned} \quad (13.28)$$

► Gauss's Law is often stated in terms of the electric field, and can be written in integral form or differential form:

$$\iint_S \nabla \cdot \mathbf{E} dA = \frac{Q_{enc}}{\epsilon_0}$$

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\epsilon_0}.$$

For a spherically symmetric charge distribution, it is convenient to choose  $S$  such that it is a sphere at the origin of radius  $r$ .

Combining this with (13.26) and (13.27), we arrive at the expression for the central field approximation of the Hartree potential:

$$\boxed{\langle V_H \rangle(r_i) = \frac{q_e^2}{4\pi\epsilon_0 r_i} \int_0^{r_i} \sum_{j \neq i} |\mathcal{P}_j(r_j)|^2 dr_j.} \quad (13.29)$$

Compare this significantly simplified form to Eq. 13.14 — this will allow us to make use of the radial equation to perform the Hartree calculation.

**Summary:** To solve the Hartree equation in the central field approximation:

- (1) Choose some initial radial probability densities

$$\{\mathcal{P}_1^{(0)}(r_1), \mathcal{P}_2^{(0)}(r_2), \dots, \mathcal{P}_M^{(0)}(r_1)\}$$

and corresponding energies  $\{E_1^{(0)}, E_2^{(0)}, \dots, E_M^{(0)}\}$ . Common methods to ensure the trial states are appropriate include:

- Using known solutions from similar multi-electron systems.
- Solving the radial equation in the case of no electron–electron interactions.

- (2) For a particular  $\mathcal{P}_i(r_i)$  and energy  $E_i$ :

- (a) Using numerical integration, calculate the spherically averaged Hartree potential due to all remaining electrons,

$$\langle V_H \rangle(r_i) = \frac{q_e^2}{4\pi\epsilon_0 r_i} \int_0^{r_i} \sum_{j \neq i} |\mathcal{P}_j(r_j)|^2 dr_j.$$

- (b) Use this to determine the effective central potential experienced by electron  $i$ ,

$$V(r_i) = -\frac{Z}{4\pi\epsilon_0 r_i} + \langle V_H \rangle(r_i).$$

- (c) Perform the Numerov–Cooley algorithm to solve the radial equation for  $\mathcal{P}_i^{(1)}(r_i)$  with this potential. This is the *first* approximation to the radial probability density for electron  $i$ , and has a corresponding first approximation to the energy  $E_i^{(0)}$ .

- (d) Normalise the radial density such that  $\int_0^\infty |\mathcal{P}_i^{(1)}(r_i)|^2 dr_i = 1$ .
- (e) Return to step (1) and repeat this procedure for all other states  $i$ . Make sure you use the new approximation for the state just calculated,  $\mathcal{P}_i^{(1)}(r_i) \rightarrow \mathcal{P}_i(r_i)$ .

- (3) Continue this iteration  $n$  times to determine the  $n$ th approximations to the radial probability densities.

- (4) The iteration has converged when the spherically averaged Hartree potential becomes self-consistent; that is, for one full iteration, the Hartree potential  $\langle V_H \rangle(r_i)$  of the input states  $\mathcal{P}_i^{(n)}(r_i)$  is identical to the Hartree potential of the output states  $\mathcal{P}_i^{(n+1)}(r_i)$ .

## 13.4 Modelling Lithium

With the steps of the Hartree model set out, we can now use it to model a multi-electron atom slightly more complicated than the hydrogen atom we encountered earlier — the lithium atom. Lithium has atomic number 3, and thus has 3 protons in the nucleus (along with 4 neutrons), giving a total nucleus charge of  $Z = 3$ . A neutral lithium atom therefore contains 3 orbital electrons; likewise, the ions  $\text{Li}^+$  and  $\text{Li}^{2+}$  contain 2 and 1 electrons respectively. In this section, we will work through solving the Lithium orbitals using the Hartree method, with Python and Fortran code provided as we go. As always, Fortran code (with grey background) will come first, followed by Python (with yellow background).

### 13.4.1 Generating the Trial Wavefunctions with $\text{Li}^{2+}$

To start with, we must choose trial wavefunctions and energy values to begin our self-consistent iterative field approach. One option is to use the radial density functions and the energy eigenvalues of the hydrogen atom — these are known analytically, and can be hard-coded as the ‘zeroth’ approximation. A slightly better approach, however, is to instead consider the lithium ion  $\text{Li}^{2+}$ ; this will likely be a closer match to the neutral lithium atom, and as it involves only a single electron, can be solved using the Numerov–Cooley technique as in the previous chapter.

To find the ground state and first excited state of  $\text{Li}^{2+}$ , we can solve the radial equation

$$-\frac{\hbar^2}{2m_e} \frac{d^2\mathcal{P}_{n\ell}}{dr^2} + \left[ \frac{\hbar^2\ell(\ell+1)}{2m_e r^2} - k \frac{Zq_e}{r} - E_n \right] \mathcal{P}_{n\ell}(r) = 0, \quad (13.30)$$

for the case  $Z = 3$ . Using the same approach as for hydrogen (see Example 12.1), we can apply the Numerov method with the Cooley energy correction formula, to determine the ground state  $1s$  ( $n = 1, \ell = 0$ , which has  $n - \ell - 1 = 0$  nodes), and the first excited state  $2s$  ( $n = 2, \ell = 0$ , which has  $n - \ell - 1 = 1$  node). We will use a radial grid of size  $10^{-15} \leq r \leq 20$ , with spacing  $\Delta r = 0.001$ . Note that, as before, we choose  $r_{min} = 10^{-15} > 0$  small enough so as to get the proper behaviour for small  $r$ , but still avoid the singularity at  $r = 0$ .

#### Example 13.1 Fortran

```
program lithium
    use numerovcooley
    use integration
    implicit none
    real(8), parameter :: Eh = 27.211386018d0
    integer :: Z, N, m, i, j, k
```

► The subroutines `numerov` and `normalise` come from the custom modules `numerovcooley` and `integration` respectively.

For brevity, these subroutines and modules are not provided here, but are based on examples we have covered previously. Have a go coding them yourself to complete this example!

```

real(8)          :: dr
real(8), allocatable :: E(:, ), r(:, ), V(:, ), phi(:, :, ), &
                      rho(:, ), Qenc(:, )

! set the parameters for the algorithm
Z = 3
dr = 0.001d0
N = int((20-1d-15)/dr + 1)

! allocate arrays and create the discretised r grid
allocate(r(N), E(0:2), phi(0:2, N), V(N), Qenc(N), rho(N))
r = [(1.d-15 + dr*i, i=1, N)]

! create empty array for storing radial probability
! densities, phi(0), phi(1), phi(2)
phi = 0.d0

! create the electron-nucleus Coulomb potential
V = -Z/r

! Find the ground state (0 nodes)
! using an energy range -10 < E < -1
call numerov(phi(0,:), E(0), V, dr, N, -10.d0, -1.d0, 0)
call numerov(phi(1,:), E(1), V, dr, N, -10.d0, -1.d0, 0)

! Find the first excited state (1 node)
! using an energy range -10 < E < -0.5
call numerov(phi(2,:), E(2), V, dr, N, -10.d0, -1.d0, 1)

! normalise the wavefunctions
do i=0, 2
    call normalise(phi(i,:), dr, N)
end do
write(*,*) 'Li2+ energy:', E*Eh

```

- For brevity, the functions `numerov` and `normalise` are not provided here, but are based on examples we have covered previously. Have a go defining them yourself to complete this example!

### Example 13.2 Python

```

import numpy as np

# Hartree energy
Eh = 27.211386018

# create the discretised radial grid

```

```

dr = 0.001
r = np.arange(1e-15, 20+dr, dr)
N = len(r)

# set number of protons in the lithium nucleus
Z = 3

# create an empty array for storing the three
# radial probability densities, phi[0], phi[1], phi[2]
phi = np.zeros([3, N])
# empty array for storing the energies
E = np.zeros([3])

# create the electron-nucleus Coulomb potential
V = -Z/r

# Find the ground state (0 nodes)
# using an energy range -10 < E < -1
E[0], phi[0] = numerov(V, dr, [-10, -1], nodes=0)
E[1], phi[1] = numerov(V, dr, [-10, -1], nodes=0)

# Find the first excited state (1 node)
# using an energy range -10 < E < -0.5
E[2], phi[2] = numerov(V, dr, [-10, -0.5], nodes=1)

# normalise the wavefunction
for i in range(3):
    phi[i] = normalise(phi[i], dr)

print('Li2+ energy:{:.4f}'.format(E*Eh))

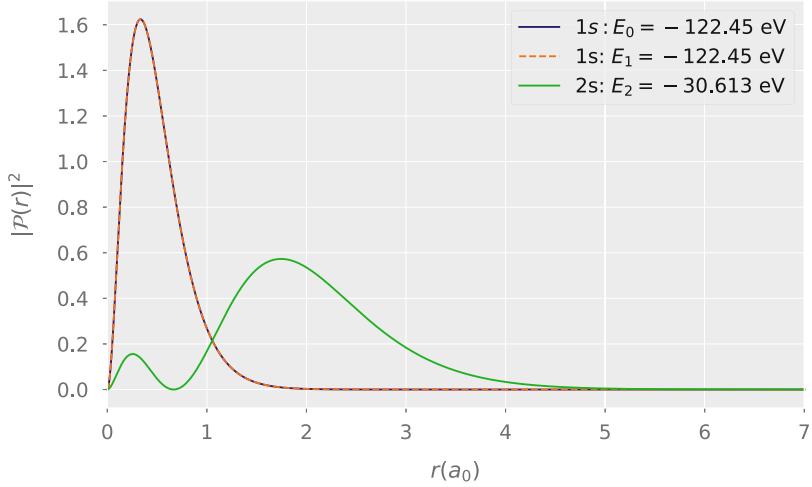
```

A couple of things to note regarding the above code examples:

- Using the hydrogen atom as a guide, the two lowest energy orbitals are  $1s$  and  $2s$ . In both cases  $\ell = 0$ , and so the effective potential is simply  $V(r) = -Z/4\pi\epsilon_0 r$ ; there is no angular momentum component.
- The `numerov` function/subroutine uses node counting and bisection of an initial energy range to ensure we are solving for the correct state. Once the node count is correct, Cooley's energy correction formula is then used to converge on the final energy eigenvalue.

Aside from that, this should be relatively familiar — it is the same process we used to solve the hydrogen atom.

Let's have a look at the resulting output energies and wavefunctions for the  $\text{Li}^{+2}$  ion:



**Figure 13.1** Radial probability densities of the  $\text{Li}^{+2}$  ion determined using the Numerov–Cooley method

This looks good, but how can we be sure that these correspond to the states  $n = 1, \ell = 0$  and  $n = 2, \ell = 0$ ? Since the number of nodes of the radial probability density is given by  $n - \ell - 1$ , there are multiple combinations of  $n$  and  $\ell$  that could potentially give the same number of modes. Luckily, since  $\text{Li}^{+2}$  is a single electron atom, we can simply use the analytic expression for the energy levels (Eq. 12.27):

$$\begin{aligned} E(n=1) &= -\frac{Z^2}{2(1)^2} E_h = -4.5 E_h \approx -122.4512 \text{ eV} \\ E(n=2) &= -\frac{Z^2}{2(2)^2} E_h = -2.25 E_h \approx -30.6128 \text{ eV}. \end{aligned} \quad (13.31)$$

This is a less than 1% relative error compared to our Numerov–Cooley implementation! We now have three initial energy values and radial probability densities with which to perform the Hartree calculation.

### 13.4.2 Performing the Hartree Iteration

In Sect. 13.3.1, we summarised the process of performing the Hartree iterative procedure when using the central field approximation. We have now completed step (1), determining the trial wavefunctions (in this case, we have used  $\text{Li}^{+2}$ ). Let's move on to step (2), and break down exactly what we must do to solve the Hartree equation in the central field approximation.

Continuing our Fortran and Python programs above,

**Example 13.3** Fortran

```
program lithium
....  
  
! begin the Hartree iterative procedure.  
! Let's perform 5 iterations.  
do k=1, 5  
    ! loop over all electrons i  
    do i=0, 2  
  
        ! calculate the charge density  
        rho = 0.d0  
        do j=0, 2  
            if (j /= i) then  
                rho = rho + abs(phi(j,:))**2  
            end if  
        end do  
  
        ! calculate the enclosed charge for each  
        ! radius r_j by using Simpsons integration  
        Qenc = 0.d0  
        do j=1, N  
            Qenc(j) = simps(rho(:j+1), dr, j+1)  
        end do  
  
        ! determine the effective potential  
        V = -Z/r + Qenc/r  
  
        ! perform the Numerov-Cooley method  
        ! starting from the trial energy  
        call numerov_cooley(phi(i,:), E(i), V, dr, N)  
        call normalise(phi(i,:), dr, N)  
    end do  
    write(*,*)'Iteration', k  
    write(*,*)'Energies', E*Eh
end program lithium
```

► This is a continuation of the Fortran program above!

- This is a continuation of the Python program above!

### Example 13.4 Python

```

from scipy.integrate import simps

# begin the Hartree iterative procedure
# Let's perform 5 iterations.
for k in range(5):
    # loop over all electrons i
    for i in range(3):
        # use list comprehension to create a list
        # of all remaining electrons
        j = [n for n in range(3) if n != i]

        # calculate the charge density
        rho = np.sum(np.abs(phi[j, :])**2, axis=0)

        # calculate the enclosed charge for each radius r_j
        # by using Simpsons integration and list comprehension
        Qenc = [simps(rho[:j+1], dx=dr) for j in range(N)]
        Qenc = np.array(Qenc)

        # determine the potential experienced by the electron
        V = -Z/r + Qenc/r

        # perform the Numerov-Cooley method,
        # starting from the trial energy
        E[i], phi[i] = numerov(V, r, dr, E[i])
        phi[i] = normalise(phi[i], dr)

    print('Iteration: ', k)
    print('Energies: ', E*Eh)

```

Some notes on the above code snippets:

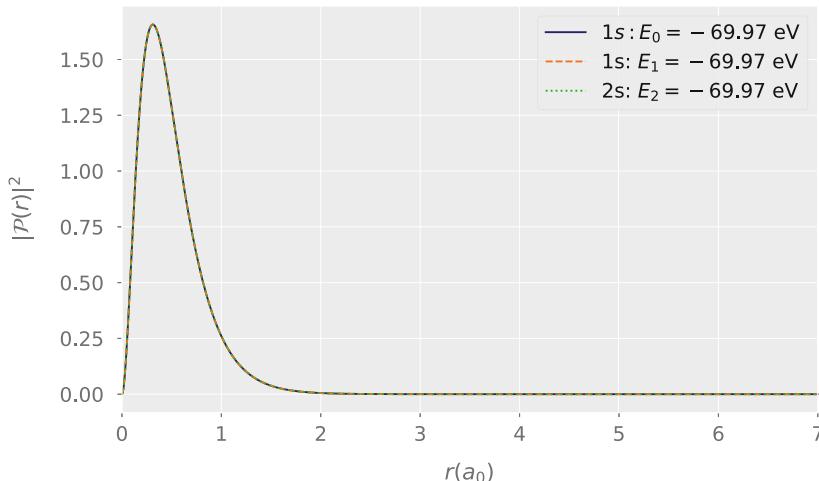
- The function `simps` performs numerical integration (in this case, Simpson's method) on the provided discretised function. In Fortran, this will have to be coded separately, whereas in Python we can simply import `scipy.integrate.simps`.
- We are no longer using node counting within the Numerov method; instead, we start with the provided trial energy, and use the Cooley energy correction formula straight away.

### Integration subtleties

Recall that Simpson's method only applies to functions discretised into  $N$  points, where  $N$  is odd. If  $N$  is even, one solution is to integrate to  $N - 1$  using Simpson's integration, and then integrating the final interval using another method, such as the Trapezoidal method. This is done *automatically* by `scipy.integrate.simps`.

Running our Hartree program thus far, we get the following output:

```
Iteration 0. Energies: [-94.69418119 -94.03694427 -70.60649543]
Iteration 1. Energies: [-70.19102859 -69.7720818 -69.91822247]
Iteration 2. Energies: [-69.98321353 -69.97026775 -69.96592323]
Iteration 3. Energies: [-69.96630355 -69.96352596 -69.96256218]
Iteration 4. Energies: [-69.96513772 -69.96602171 -69.9668367 ]
```



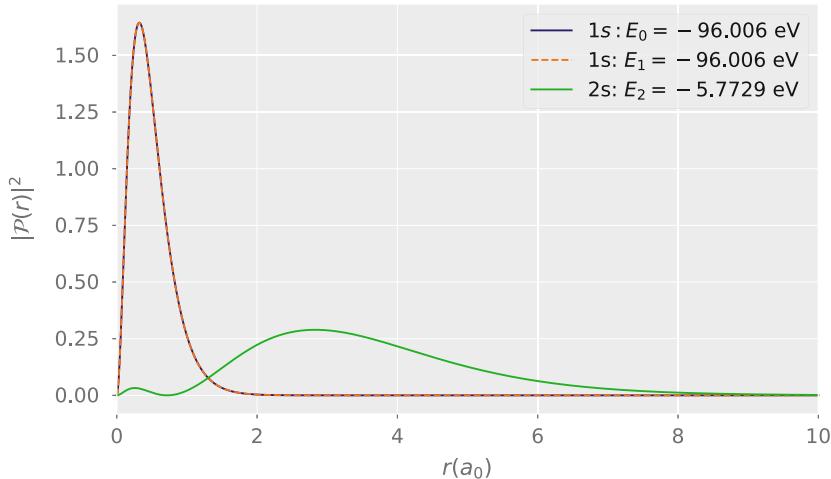
**Figure 13.2** Radial probability densities of the lithium atom determined using 5 iterations

While the Hartree method has converged, indicating the Hartree potential is self-consistent, they have converged in such a way as to place all three electrons into the same 1s orbital! This is **unphysical**, as orbitals can only contain, at most, two electrons. To avoid this scenario, let's reduce the trial value of the excited state, from the  $E_2 = -30.6128\text{ eV}$  of  $\text{Li}^{2+}$  down to  $E_2 = -7\text{ eV}$ . Manually setting the new trial value of  $E_2$  and running the program again gives

```

Iteration 0. Energies: [-94.69418119 -94.03694427 -5.769109 ]
Iteration 1. Energies: [-95.97366275 -96.01061142 -5.77298305]
Iteration 2. Energies: [-96.00893483 -96.00572614 -5.77284937]
Iteration 3. Energies: [-96.0056394 -96.00588079 -5.7728553 ]
Iteration 4. Energies: [-96.00639346 -96.0064626 -5.77286636]

```



**Figure 13.3** Radial probability densities of the lithium atom determined using five Hartree iterations

The Hartree method continues to converge, and this time to the result we are after — two electrons in the ground state, and the outer or valence electron in the first excited state. While we can't verify these results analytically, since Koopman's theorem relates the ionisation energy to the Hartree energy eigenvalues, there is a multitude of experimental data we can use, including the online Handbook of Basic Atomic Spectroscopic Data provided by the National Institute of Standards and Technology, or the CRC Handbook of Chemistry and Physics — for decades *the* reference for chemical and physical data.

- First published in 1913, the CRC Handbook is currently on its 98th print edition. So pervasive, it was even nicknamed the 'Blue bible'.

The CRC Handbook of Chemistry and Physics lists the first three ionisation energies of Lithium as I: 5.39172 eV, II: 75.64018 eV, and III: 122.45429 eV, corresponding to removing an electron from Li,  $\text{Li}^+$ , and  $\text{Li}^{2+}$  respectively. We have already verified the third ionisation energy analytically when we considered the single electron  $\text{Li}^{2+}$ , and we can now see that the Hartree method applied to neutral lithium determined the energy of the excited 2s state to within  $\sim 7\%$ . Not a bad result!

## 13.5 The Hartree–Fock Method

Unfortunately, the Hartree method has one major downfall — *it is not correct*. We saw hints of this in the previous section, when applying the Hartree method to the lithium atom resulted in convergence to a non-physical state — three electrons in the ground state. Electrons are Fermions, particles with non-integer spin, and therefore *must* obey the **Pauli exclusion principle**, which states that no two Fermions can inhabit the same quantum state. Since electrons have an intrinsic spin  $s = 1/2$ , this allows at most two electrons per orbital:

$$\phi_{n,\ell,m} : \quad S_z = +\frac{1}{2}\hbar, \quad S_z = -\frac{1}{2}\hbar,$$

one with a positive  $z$ -component of spin (**spin up**) and one with a negative  $z$ -component of spin (**spin down**). As a consequence, we can define an additional quantum number  $m_s$  known as the **spin quantum number**; combined with the other quantum numbers, these now completely identify a unique quantum state:

- the principal quantum number  $n$  associated with the energy  $\hat{H}$ ,
- the azimuthal quantum number  $0 \leq \ell \leq n - 1$  associated with angular momentum magnitude  $\hat{L}^2$ ,
- the magnetic quantum number  $-\ell \leq m_\ell \leq \ell$  associated with the  $z$ -component of angular momentum  $\hat{L}_z$ ,
- the spin quantum number  $-s \leq m_s \leq s$  associated with the  $z$ -component of particle spin  $\hat{S}_z$ .

► Note that we now denote the magnetic quantum number by  $m_\ell$ , to avoid confusion with the spin quantum number.

This allows us to slightly rephrase the Pauli exclusions principle: no two Fermions can be described by the same set of quantum numbers. This is what the Hartree method fails to account for.

So, how do we take into account the Pauli exclusion principle in the Hartree method? It turns out that the Pauli exclusion principle follows from a more elementary property of Fermions; their wavefunctions are always **antisymmetric**.

### 13.5.1 Antisymmetric Wavefunctions

Fermions are indistinguishable particles; that means, if we swap or ‘exchange’ the states of two Fermions, the overall state must remain unchanged up to a phase factor. Making use of the particle exchange operator  $\hat{P}_{12}$ , which swaps particles labelled 1 and 2, all this requires is that

$$|\hat{P}_{12}\psi(\mathbf{r}_1, \mathbf{r}_2)|^2 = |\psi(\mathbf{r}_2, \mathbf{r}_1)|^2. \quad (13.32)$$

- $\mathbf{r}_i$ , in addition to referring to the spatial components of particle  $i$ , now also includes the **spin** components as well.

If we are to use the orbital approximation,  $\psi(\mathbf{r}_1, \mathbf{r}_2) = \phi_1(\mathbf{r}_1)\phi_2(\mathbf{r}_2)$ , where  $\phi_1$  and  $\phi_2$  are potentially different orbitals,

$$|\hat{P}_{12}\phi_1(\mathbf{r}_1)\phi_2(\mathbf{r}_2)|^2 = |\phi_1(\mathbf{r}_2)\phi_2(\mathbf{r}_1)|^2 \neq |\phi_1(\mathbf{r}_1)\phi_2(\mathbf{r}_2)|^2. \quad (13.33)$$

Thus, requiring the particles be indistinguishable restricts the allowed form of the wavefunction to the following:

$$\psi(\mathbf{r}_1, \mathbf{r}_2) = \frac{1}{\sqrt{2}} (\phi_1(\mathbf{r}_1)\phi_2(\mathbf{r}_2) \pm \phi_1(\mathbf{r}_2)\phi_2(\mathbf{r}_1)). \quad (13.34)$$

Double check that this indeed preserves the wavefunction probability under particle exchange!

As Fermions are antisymmetric, in addition to being indistinguishable, this places an additional constraint on the form of the wavefunction; only those which *change sign* under the particle exchange operator are allowed:

$$\hat{P}_{12}\psi(\mathbf{r}_1, \mathbf{r}_2) = -\psi(\mathbf{r}_2, \mathbf{r}_1). \quad (13.35)$$

Therefore, for two Fermions, an antisymmetric orbital approximation requires

$$\psi(\mathbf{r}_1, \mathbf{r}_2) = \frac{1}{\sqrt{2}} (\phi_1(\mathbf{r}_1)\phi_2(\mathbf{r}_2) - \phi_1(\mathbf{r}_2)\phi_2(\mathbf{r}_1)) = \frac{1}{\sqrt{2}} \begin{vmatrix} \phi_1(\mathbf{r}_1) & \phi_2(\mathbf{r}_1) \\ \phi_1(\mathbf{r}_2) & \phi_2(\mathbf{r}_2) \end{vmatrix}. \quad (13.36)$$

Notice that this has the same form as a  $2 \times 2$  matrix determinant; as a result, we can generalise this to  $N$  Fermions:

$$\psi(\mathbf{r}_1, \mathbf{r}_2) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \phi_1(\mathbf{r}_1) & \phi_2(\mathbf{r}_1) & \cdots & \phi_N(\mathbf{r}_1) \\ \phi_1(\mathbf{r}_2) & \phi_2(\mathbf{r}_2) & \cdots & \phi_N(\mathbf{r}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\mathbf{r}_N) & \phi_2(\mathbf{r}_N) & \cdots & \phi_N(\mathbf{r}_N) \end{vmatrix}. \quad (13.37)$$

This is known as the **Slater determinant**.

### 13.5.2 The Hartree–Fock Equations

If we were to apply the variational principle to the multi-electron Schrödinger equation, this time using the Slater determinant orbital approximation as our trial wavefunction or *ansatz*, we would find that each orbital wavefunction needs to satisfy the following differential equation:

$$-\frac{\hbar^2}{2m_e} \nabla_i^2 \phi_i(\mathbf{r}_i) - \frac{Zq_e^2}{4\pi\epsilon_0 r_i} \phi_i(\mathbf{r}_i) + V_{HF}(\mathbf{r}_i) \phi_i(\mathbf{r}_i) - \hat{K} \phi_i(\mathbf{r}_i) = E_i \phi_i(\mathbf{r}_i),$$

(13.38)

where

$$V_{HF}(\mathbf{r}_i) = \int \frac{q_e^2 \sum_j |\phi_j(\mathbf{r}_j)|^2}{4\pi\epsilon_0 |\mathbf{r}_i - \mathbf{r}_j|} d\mathbf{r}_j$$

(13.39)

is the **Hartree–Fock potential**, and

$$\hat{K}\phi_i(\mathbf{r}_i) = \sum_j \int \phi_j^*(\mathbf{r}_j) \frac{q_e^2 \phi_j(\mathbf{r}_i) \phi_i(\mathbf{r}_j)}{|\mathbf{r}_i - \mathbf{r}_j|} d\mathbf{r}_j \quad (13.40)$$

is the **exchange term**, where  $\hat{K}$  is the exchange operator, and comes about simply by assuming a solution that is antisymmetrised.

The total energy of the multi-electron atom can then be written as

$$E = \sum_i E_i - \frac{1}{2} \sum_j \sum_k [\langle \phi_i \phi_j | V_{ij} | \phi_i \phi_j \rangle - \langle \phi_j \phi_i | V_{ij} | \phi_i \phi_j \rangle] \quad (13.41)$$

where  $V_{ij}$  is the electron–electron Coulomb potential. Using these equations, the Hartree–Fock method can be solved in an iterative fashion, in much the same way as the Hartree method.

### Hartree–Fock and other methods

Due to the involvement of spin and the exchange operator, the Hartree–Fock method is slightly more complicated to use than the Hartree method; we won’t derive or use it here, however it is important to see how the Hartree–Fock method corrects oversights in the Hartree method. If you are interested to read more, see the further reading at the end of the chapter.

It is also important to note that the Hartree–Fock method is still an approximation — we are assuming the multi-electron system has a wavefunction that can be described by a Slater determinant. Often, this is not the case; as such, higher order methods and approximations have been developed such as **density functional theory** (DFT) and **configuration interaction** (CI) theory. These methods are pervasive in modern quantum chemistry.

- Unlike the Hartree potential  $V_H$ , the Hartree–Fock potential sums over the case  $j = i$ , essentially allowing for the electron under consideration to ‘self-interact’ with its own Coulomb potential. This is not too much of a cause for concern, since this is cancelled out by the exchange interaction of the electron with itself!

## 13.6 Quantum Dots (and Atoms in Flatland)

It's been a wild ride so far — we have covered topics including scientific programming, numeric methods, and their applications to solving the Schrödinger equation. These have included a wide variety of systems, from the quantum harmonic oscillator to the lithium atom. Let's finish with something a little different; a brief discussion of how these concepts and theories have led to discoveries and applications in nanotechnology and engineering.

- ▶ Quantum dots were first fabricated by Louis E. Brus at Bell Labs in 1984.

A major objective of physics is the use of the scientific method and mathematics to discern and understand the workings of the natural world. So far in this text, we have considered atoms that occur naturally in nature, like hydrogen and lithium; in the latter case, applying numerical techniques to solve a system that cannot be solved analytically. However, with this understanding, comes the ability to harness what we have learnt to construct *artificial* systems such that they display certain properties not found naturally. **Quantum dots** are such an example; an interdisciplinary field of study spanning engineering, nanotechnology, and physics, quantum dots can be manufactured today, and have seen uses in display technology, solar panel design, biotechnology, and quantum optics.

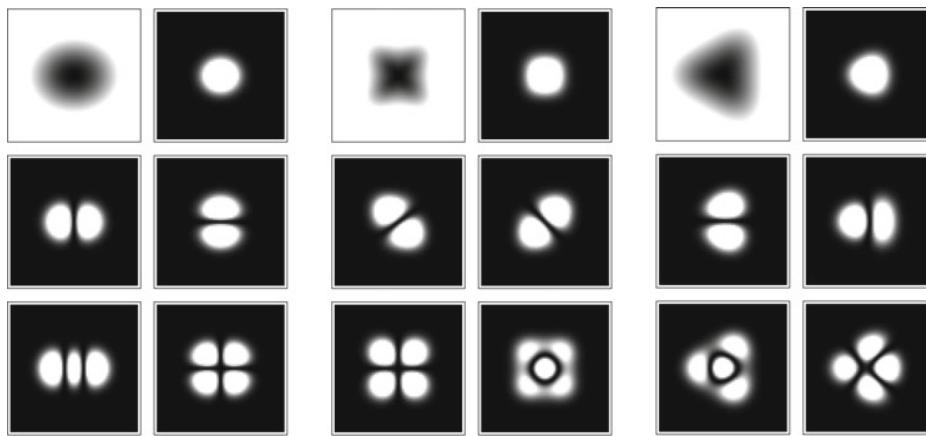
Quantum dots are generally fabricated using semi-conductors, where all electrons are tightly bound to the nuclei except for a very small fraction of mobile electrons. These mobile electrons can then be trapped between two layers of semiconductors, effectively confining the electrons to two dimensions. In fact, we can then confine the electrons even further, by applying a **confining potential**, to create a quantum dot. With the confining potential generally around 10 nm in diameter, the electrons confined to the quantum dot are now *extremely* restricted in all three spatial dimensions. Thus, even though the quantum dot exists in an extremely small 3-dimensional space, due to this restriction we commonly refer to the quantum dot as **low-dimensional** or **zero-dimensional**.

As we have seen in earlier chapters, particles confined to a potential satisfy the time-independent Schrödinger's equation with discrete, orthonormal wavefunctions and energies — referred to as bound states. The same is true of quantum dots, and by varying the number of confined electrons, the shape/symmetry of the confining potential, we can design energy levels to serve particular purposes. As a result, quantum dots are often nicknamed **artificial atoms**, and by modifying the confining potential, a periodic table of artificial atoms can even be constructed.

- ▶ In addition to the zero-dimensional quantum dot, a **two-dimensional quantum well** constrains the dynamics of an electron in one spatial directions, and a **one-dimensional quantum wire** constrains the dynamics in two dimensions
- ▶ A common use of quantum dots is to provide specific energy levels needed for emission of particular wavelengths of light.

### 13.6.1 The Quantum Dot Radial Equation

Depending on the fabrication method of quantum dots, there are a wide array of differing forms the corresponding Schrödinger equation can take. A simple model involves quantum dots that are confined to a spherical three-dimensional potential well; as the spherical well is a form of a central potential



(a) Circular potential      (b) Square potential      (c) Triangular potential

**Figure 13.4** Various quantum dots or artificial atoms, confined to a circular, square, and triangular potential respectively. The first panel in each figure shows the confinement potential; the other panels show the radial density probability distributions of the first five states

(the spherical potential well only depends on radius from the centre of the charges  $r$ ), this allows us to make use of the orbital approximation  $\phi_i(\mathbf{r}_i) = \frac{1}{r_i} \mathcal{P}_{nl}(r_i) Y_{nl}(\theta_i, \phi_i)$ , and therefore reduces the problem of solving the time-independent Schrödinger equation down to one of solving the radial equation

$$-\frac{\hbar^2}{2m_e} \frac{d^2\mathcal{P}_{nl}}{dr^2} + \left[ \frac{\hbar^2 \ell(\ell+1)}{2m_e r^2} + V(r) - E_n \right] \mathcal{P}_{nl}(r) = 0, \quad (13.42)$$

where  $V(r)$  is the confining potential of the quantum dot. The form of  $V(r)$  is highly dependent on the form of the quantum dot; for a quantum dot of approximate radius  $R$ , common modelling choices include:

- **A finite well**  $V(r \geq R) = V_0$ ,  $V(r < R) = 0$ . This is a useful approximation for large quantum dots (i.e. where  $R \gg 1a_0$ ).
- **The inverse Gaussian potential**  $V(r) = -V_0 e^{r^2/R^2}$ .
- **The Pöschl–Teller potential**  $V(r) = -V_0 / \cosh^2(-r^2/R^2)$ .
- **The quantum harmonic potential**  $V(r) = -V_0 + \frac{1}{2}m_e \omega^2 r^2$ . This is a useful approximation for electrons near the ground state of the quantum dot.

►  $a_0$  is the Bohr radius.

Thus, for single electron quantum dots, finding the energy eigenvalues and bound states is a relatively straight-forward process, and one-dimensional boundary conditions techniques such as the Numerov–Cooley method can be used.

For multi-electron quantum dots, the process is similar to when we looked at multi-electron atoms; we can use the Hartree(-Fock) iterative method to compute the bound states and total energy of the quantum dot. The one difference is that the electron-nucleus attractive Coulomb potential is replaced by the *artificial* quantum dot potential  $V(r)$ .

#### Further reading

The Hartree–Fock method has been an incredibly useful process for the numerical study of atomic physics and quantum chemistry. Most texts covering the Hartree–Fock method are aimed at the graduate level, and also include discussions on more advanced methods, such as density functional theory and configuration interactions. Below, we recommend some texts for further reading on these computational quantum chemistry techniques.

- Levine, I. N. (2014). Quantum chemistry (Seventh edition). Boston: Pearson. ISBN 978-0-321-80345-0
- McQuarrie, D. A. (2008). Quantum chemistry (2nd ed). Sausalito, Calif: University Science Books. ISBN 978-1-891389-50-4
- Jensen, F. (2017). Introduction to computational chemistry (Third edition). Chichester, UK; Hoboken, NJ: John Wiley & Sons. ISBN 978-1-118-82599-0

For more details on the theory and applications behind quantum dots, the following article and book chapter are technical but go into significant detail on the applying the numerical methods to solve the structure of quantum dots:

- McCarthy, S. A., Wang, J. B., & Abbott, P. C. (2001). Electronic structure calculation for N-electron quantum dots. Computer Physics Communications, 141(1), 175204.
- Wang, J. B., Hines, C., & Muhandiramge, R. D. (2006). Electronic structure of quantum dots and other nanosystems, Handbook of Theoretical and Computational Nanotechnology (vol 10, chapter 10, American Scientific Publishers, Los Angeles)

## Exercises

- P13.1** The expression for the total energy of a system with  $N$  electrons in the Hartree approximation is given by

$$E = \langle \hat{H} \rangle = \sum_i^N E_i - \sum_i^N \sum_{j>i}^N \langle \phi_i(\mathbf{r}_i) \phi_i(\mathbf{r}_j) | V_{ij} | \phi_i(\mathbf{r}_i) \phi_i(\mathbf{r}_j) \rangle.$$

Consider the case where the  $k$ th electron is removed from the system, leading to a  $N-1$  electron system. Using the above expression, show that the change in energy,

$$\Delta E = E^{(N-1)} - E^{(N)},$$

is given by  $E_k$ , the Hartree energy eigenvalue of the  $k$ th orbital.

- P13.2** Complete the code examples in Sect. 13.4 by defining your own functions for integration, normalisation, the Numerov method (with and without node counting/energy bisection) and Cooley's energy correction formula. Verify that the program output agrees with the results presented within the section.

- (a) Compare the radial probability densities of hydrogen, the  $\text{Li}^{2+}$  ion, and the neutral lithium atom. How does the radial probability density of the excited state 2s compare across the three? How might you explain this?
- (b) (Re)calculate and plot the enclosed charge  $Q_{enc}(r)$  at radius  $r$ , as experienced by the outer/valence electron. What value does this take for large  $r$ ? Is this expected?

*Hint:* recall that the enclosed charge is calculated by integrating the charge density of the remaining electrons for all radii less than  $r$ .

- (c) Modify the program such that, instead of performing a fixed number of Hartree iterations, the program terminates when convergence is achieved and the potential becomes self-consistent. How many iterations are required until self-consistency?
- (d) Add a final computation to the program to compute the total energy of the system, using Eq. 13.17. How does your result compare with the total electronic energy of lithium determined via experimental means, 203.48619 eV?

► Double checking that the enclosed charge matches what we expect for the physical system is a useful sanity check when coding the Hartree method.

**P13.3** A computational bottleneck in the Hartree method is the calculation of the Hartree potential. In the central field approximation, we must integrate from 0 to  $r_i$  for each value of  $r_i$  in the domain:

$$\langle V_H \rangle(r_i) = \frac{q_e^2}{4\pi\epsilon_0 r_i} \int_0^{r_i} \sum_{j \neq i} |\mathcal{P}_j(r_j)|^2 dr_j, \quad \forall r_i \in [0, r_{max}].$$

This is known as a **cumulative integral**; values for a particular  $r_i$  can be determined simply by knowing the result of the integral for  $r'_i < r_i$ . As a result, simply calculating the integral numerous times for different values of  $r_i$  is wildly inefficient.

A better approach to cumulative integrals is to re-use initial integral computations, greatly increasing the efficiency. For example, for the cumulative integral

$$I(x_k) = \int_{x_0}^{x_k} f(x) dx,$$

we can use the following **cumulative Simpson's rule**:

$$I(x_k) = I(x_{k-2}) + \frac{1}{3}\Delta x(f(x_{k-2}) + 4f(x_{k-1}) + f(x_k)), \quad 3 \leq k \leq N.$$

Here, we are simply using a previous value of the cumulative integral, and adding on Simpson's rule for the subsequent points. The initial values for this recursion formula are simply

$$I(x_0) = 0,$$

$$I(x_1) = \frac{1}{3}\Delta x \left( \frac{5}{4}f(x_1) + 2f(x_2) - \frac{1}{4}f(x_3) \right).$$

► If using Python, SciPy includes a convenient function for the cumulative trapezoidal method: `scipy.integrate.cumtrapz`

- (a) Implement a numerical function that uses the cumulative Simpson's rule, and verify it gives the same result as the standard Simpson's rule.
- (b) Modify your Hartree program to use the cumulative Simpson's rule to calculate the Hartree potential. How does the wall-time of this program compare to the non-cumulative approach?

**P13.4** Using the Hartree method, calculate the energy levels, wavefunctions, and total energy of two electrons in the ground state 1s ( $n = 1, \ell = 0$ ) orbital for the helium atom ( $Z = 2$ ).

Compare your results to the experimental result for the first ionisation energy of Helium, I: 24.587387 eV, and the total energy -79.013702 eV.

**P13.5 Bonus:** Use the Hartree–Fock method to calculate the orbital wavefunctions and total energy of two electrons in the ground state orbital for the helium atom. How do your results compare to the Hartree method?

### P13.6 Quantum dot project

Using various central potentials presented in Sect. 13.6.1, or perhaps even using some of your own designs, construct a periodic table of ‘artificial atoms’ or quantum dots, each differing in a property of the potential. This could be the strength of the potential, the potential radius, it’s up to you!

Some ideas to investigate for this project include:

- (a) In the case of a single electron, solve the radial equation for each ‘atom’, to determine the ground state energy and wavefunction.
- (b) Using the Hartree(-Fock) iterative method, solve for the orbitals and total energy of each quantum dot for various numbers of electrons.
- (c) If you are using the Hartree–Fock iterative method, the Pauli exclusion principle is taken into account via the use of antisymmetrisation and the Slater determinant. Can you determine the order in which the electron orbitals are filled? How does this compare to the hydrogen atom?

# Index

- Accuracy, 13, 186, 251, 257, 336, 372, Born rule, 244, 245  
379, 384, 399–402, 419, 422, Bound state, 361, 362, 383, 403, 417,  
423, 425, 445, 448–450, 482–484  
Adaptive discretisation, 434, 437, 438 Boundary value problem, 361, 363, 365–  
Adaptive grid, 455 370, 389, 449  
Addition theorem, 468 Bracketed method, 24, 178, 201, 232,  
Additive commutativity, 10 331, 405, 464  
Affine transformation, 254 Butcher tableau, 214, 215, 218, 220,  
Aliasing, 336 229  
Angular momentum, 397, 411, 441– Butcher tableaux, 243  
443, 445, 448, 454, 456, 467, Butterfly diagram, 341–344  
473, 479  
Antisymmetric wavefunction, 479 Cartesian, 137, 140, 397, 399, 410,  
Arnoldi basis, 280, 304 429, 439, 441, 457  
Artificial atom, 482, 483, 487 Central difference method, 185–187,  
Associativity, 10–12, 15, 98 204, 224, 370, 382, 390  
Asymptotically stable, 203 Central field approximation, 467–470,  
Atomic decay, 207, 209, 228 474, 486  
Central potential, 441, 442, 445, 447,  
Backwards difference, 185–187 456, 467, 468, 470, 482, 487  
Backwards Euler Method, 193, 194, Characteristic polynomial, 268  
196, 198–200, 202, 203, 211, Charge density, 263, 464, 468, 469,  
213, 215–217, 421, 422 474, 476, 485  
Baker-Campbell-Hausdorff, 423 Chebyshev expansion, 427–429  
Banded Hermitian matrix, 294, 297 Chebyshev polynomial, 256, 427–429  
298, 302, 303 Chebyshev–Gauss quadrature, 256  
Base-10, 3–8, 10–12, 15, 91, 138, 155 Closed shell, 467, 468  
Base-2, 3–5 Commutativity, 10  
Basis diagonalization method, 399 Complete basis, 251  
Bessel function, 427, 428 Complete orthogonal set, 324  
Big-O notation, 167, 168, 339, 391 Complexity, 57, 243, 277, 418  
Binary, 3–6, 9, 11, 15, 21, 94, 130 Composite midpoint rule, 239  
Bisection method, 170, 171, 173, 175, Computational complexity, 168, 339,  
178, 373 445  
Block-diagonal matrix, 394 Computational error  
Bohr radius, 448, 483 absolute error, 171  
Boole’s Rule, 247 relative error

- round-off error, 12, 21, 208  
 systematic error  
     truncation error, 5, 208, 209, 211  
 Condition number, 286, 287  
 Conditionally stable, 213  
 Configuration interaction, 481, 484  
 Confining potential, 482, 483  
 Convergence, 163, 169–171, 173–175,  
     178, 224, 271, 274–277, 368,  
     374, 380, 419, 427–429, 466,  
     479  
 Cooley-Tukey algorithm, 339, 340, 344,  
     345, 347–349, 354  
 Coulomb potential, 447, 448, 458, 462,  
     464, 465, 481, 484  
 Crank-Nicolson method, 422  
 Criterion, 173–175, 274, 467  
     de Broglie equation, 312  
     de Broglie relation, 430  
     Degeneracy, 395, 411, 450, 454  
     Degenerate state, 395–398, 411  
     Denormalised, 7–10  
     Density functional theory, 481, 484  
     Diagonisable matrix, 287  
     Diagonalization, 399  
     Dirac notation, 399, 465  
     Direct matrix method, 382, 384, 388,  
         389, 395, 397  
     Discrete Fourier transform (DFT), 81,  
         163, 311, 321–358,  
     Discretise, 188, 311–314, 370, 379, 380,  
         382, 390, 395, 399, 418, 420,  
         424, 429, 438  
     Distributivity, 10, 12, 15  
     Dominant eigenvalue, 271–276, 278,  
         279, 307  
     Dominant eigenvector, 274, 278–280,  
         307  
     Eigenspectrum, 278, 304, 397  
     Eigenvalue, 267–309, 336, 362, 363,  
         372–384, 395–408, 417, 419,  
         423, 427–429, 442, 449, 464,  
         471, 478, 483  
     Eigenvector, 267–309, 362, 383, 395–  
         408  
     Energy correction, 380, 382, 450, 458,  
         471, 473, 476, 485  
     Error scaling, 168, 228, 391  
     Euler method, 188, 189, 192, 198, 199,  
         207, 208, 210–212, 215, 217,  
         226, 228, 420–422, 437  
     Exchange, 479–481  
     Explicit, 20, 56, 57, 59, 64, 90, 167,  
         193, 213, 215, 217–219, 224,  
         256  
     Exponent bit, 7, 9, 10  
     False position method, 178  
     Fast Fourier Transform (FFT), 311  
     Fermion, 479, 480  
     FFTW, 350–353, 357, 358, 444  
     Fine structure, 454  
     Finite Difference, 61, 181, 183, 186,  
         187, 204, 224, 321, 357, 376,  
         382, 384, 388–390, 393, 399,  
         402, 409, 410, 420, 421, 424,  
         425, 429, 437, 438  
     Finite difference, 455  
     Finite well, 483  
     First-order approximation, 173, 199,  
         420  
     Fixed-point, 4, 5, 169  
     Flatland, 482  
     Floating-point, 5–7, 9–15, 91, 92, 155,  
         282, 287  
     Fortran  
         Array  
             allocate, 45  
             dimension, 45  
             dot product, 44  
             matmul, 44  
             maxloc, 44  
             maxval, 44  
             merge, 45  
             minloc, 44  
             minval, 44  
             product, 44  
             reshape, 45

- shape, 45
- sum, 44
- transpose, 45
- Control statements
  - do loops, 26, 27
  - do while loops, 27
  - if, 25
  - select case, 26
- Input and output
  - close, 34
  - open, 34
  - read, 36
  - write, 35, 36
- Intrinsic function
  - abs, 39
  - acos, 38
  - aimag, 39
  - arg, 39
  - asin, 38
  - atan, 38
  - ceiling, 39
  - conj, 39
  - cos, 38
  - cosh, 38
  - exp, 38
  - floor, 39
  - log, 38
  - max, 39
  - min, 39
  - mod, 39
  - modulo, 39
  - real, 39
  - sign, 39
  - sin, 38
  - sinh, 38
  - sqrt, 38
  - tan, 38
  - tanh, 38
- Operators
  - arithmetic, 23, 24
  - logical, 23, 24
  - relational, 23, 24, 95
- Procedure
  - assumed-shape arrays, 55
  - call, 73
  - contains, 52
  - dummy argument, 53
  - external procedure, 53
  - function, 49
  - global variable, 54
  - global variable, 53
  - interface, 56
  - internal procedure, 52, 56
  - local variable, 53
  - local variables, 53
  - module, 64
  - optional, 54
  - overloading, 59
  - recursive procedure, 62
  - subroutine, 49
- String manipulation, 29
  - adjustl, 29
  - adjustr, 29
  - index, 29
  - len, 29
  - substrings, 30
  - trim, 29
- Variable declaration
  - character, 19
  - complex, 18, 19
  - integers, 18
  - logical, 19
  - real, 18
- variable declaration
  - implicit none, 22
- Forwards difference, 183
- Forwards Euler method, 193, 198, 202, 207, 211, 421, 422
- Fourier coefficients, 322–324, 336
- Fourier differentiation, 316–321, 354–357, 402, 407, 438
- Fourier matrix, 317, 327
- Fourier series, 321–325, 329, 336, 340, 401
- Fourier transform, 311–315, 317, 318, 321–323, 325–332, 335, 338, 345, 347, 355, 356, 358, 402, 423, 430, 431

- Fubini's theorem, 257  
 Gauss's Law, 468, 469  
 Gauss–Hermite quadrature, 255, 265  
 Gauss–Laguerre quadrature, 255  
 Gauss–Legendre quadrature, 249–251, 254, 255, 264  
 Gaussian quadrature, 249, 253–255, 260, 261  
 Gaussian wave packet, 430, 431, 437, 438, 440  
 Grid discretisation, 356, 395, 438  
 Hamiltonian, 308, 309, 356, 361, 362, 383, 394, 395, 397, 399, 403, 404, 406, 417, 419, 422, 423, 427–429, 442, 443, 461–463, 465, 467  
 Harmonic oscillator, 244, 245, 308, 309, 356–358, 361, 378, 385–387, 396, 401, 410–414, 439, 441, 447, 450, 457, 482  
 Hartree approximation, 462, 485  
 Hartree equation, 464–467, 470, 474  
 Hartree potential, 464–470, 477, 481, 486  
 Hartree-Fock equation, 480  
 Hartree-Fock method, 461, 465, 479, 481, 484, 487  
 Hartree-Fock potential, 481  
 Helium, 461, 486, 487  
 Hermite polynomial, 255, 265  
 Hermitian matrix, 270, 272, 273, 275, 278, 281–284, 294–296, 302, 303, 306, 307  
 Hermitian power iteration, 275  
 Heun's method, 215  
 Higher order approximation, 186  
 Hilbert space, 338, 362, 370, 399, 405, 417, 419, 446  
 Homogeneous ODE, 369  
 Hydrogen atom, ix, 397, 441, 447, 449, 461, 471, 473, 487  
 Hyperfine splitting, 454  
 IEEE754, 6, 7, 9, 10, 12, 13, 91, 139  
 Ill-conditioned, 287  
 Implicit, 6, 19–21, 64, 86, 90, 93, 97, 176, 196, 203, 215, 217–219, 290, 294, 379, 421, 424, 450  
 Indistinguishable particle, ix, 479  
 Infinite square well, ix, 388, 401  
 Initial value problem, 183, 188, 189, 192, 200, 211, 214, 215, 221, 222, 225, 229, 237, 286, 363–373, 375, 389, 417  
 Integral, 231, 236, 239–241, 249, 252–254, 257, 259, 260, 311–313, 322, 324, 329–331, 381, 400–403, 405, 407, 463, 469  
 Integration, 154, 231–261, 311, 321, 336, 372, 374, 402, 403, 417, 470, 476, 477  
 Intermediate value theorem, 170, 178  
 Inverse Gaussian potential, 412–414, 483  
 Inverse power iteration, 276, 306  
 Koopman's theorem, 464, 478  
 Kronecker product, 409  
 Krylov subspace technique, 280, 283, 290, 294, 304, 305, 412, 436  
 Lagrange polynomial, 263  
 Laguerre polynomial, 411, 449  
 Lanczos algorithm, 282–284, 294, 302, 303, 306, 307, 399, 403, 406  
 Lanczos basis, 280, 281, 304  
 Lanczos iterative algorithm, 281  
 LAPACK, 288–290, 293, 294, 298, 301–305, 307–309, 350, 383, 387, 388, 407, 410, 412, 444  
 Laplacian operator, 423, 441  
 Leakage, 336, 338  
 Leap-frog method, 204–208, 210–213, 220, 228, 240, 424, 425  
 Left eigenvector, 286, 287, 290, 301, 308  
 Legendre Polynomial, 251, 252, 254, 264

- Linear multi-step methods, 225  
Lithium, 461, 471, 477–479, 482, 485  
Local minimum, 406  
Machine epsilon, 12, 13, 15, 21, 139, 208  
Mantissa bit, 9, 10  
Matching method, 375, 377, 379, 380, 386–389  
Midpoint rule, 239–241, 262  
Momentum space, 311–314, 318, 321, 423, 430, 431  
Monte Carlo method, 257, 408  
Monte-Carlo Integration, 259, 402, 415  
Morse potential, 386  
Multi-electron system, ix, 461, 462, 470, 481  
Multiplicative commutativity, 10  
NaN, 9  
Network centrality, 274  
Newton–Cotes Rules, 247, 248, 260  
Newton–Raphson method, 194, 196, 216, 268  
Newton–Raphson method, 173–175, 178, 179, 366  
Node counting, 373, 376, 378, 450, 473, 476, 485  
Non-Cartesian, 398, 441  
Non-uniform grid, 455, 459  
Normalised, 5, 7, 8, 244–246, 272, 275–279, 282, 287, 301, 306, 371, 412, 419, 446, 462, 463  
Numerical integration, 212, 231, 248, 260–262, 372, 402, 407, 439, 457  
Numerov–Cooley method, 382, 387, 450, 457, 458, 474, 483  
Nyquist condition, 315, 330, 331, 430–432, 437, 438  
Nyquist theorem, 315, 328, 336, 431  
Nyquist wavenumber, 330, 336  
ODE, 154, 183, 188, 199, 201–204, 206, 207, 217, 219–224, 237, 238, 286, 325, 366–369, 379, 380, 426  
ODEPACK, 224  
Orbital, 340, 407, 449, 457, 462–467, 471, 473, 477, 480, 483, 486, 487  
Order of approximation, 168, 186, 383  
Order of convergence, 169  
Orthogonalisation algorithm, 278  
Orthogonality, 251, 255, 313, 324  
Overlap matrix, 407, 408, 415  
Pöschl-Teller potential, 414, 483  
PageRank, 274  
Plane-wave basis, 402  
Position space, 312, 314, 318, 361, 380, 382, 420, 423, 437  
Power iteration, 268, 271–277, 279, 280, 283, 307  
Precision  
double precision, 9, 10, 12, 18, 21, 59, 60, 69, 79, 91, 139, 155, 209  
fixed precision, 208  
quadruple precision, 10  
single precision, 6–12, 15, 21, 32, 69, 81, 155, 207, 209  
Pseudo code, 168  
Python  
Control statement  
for loops, 119  
if, 119  
while loop, 119  
Data structures  
count, 108  
index, 108  
len, 108  
list.append, 111  
list.clear, 112  
list.copy, 112  
list.extend, 111  
list.insert, 112  
list.pop, 112  
list.remove, 112  
list.reverse, 112  
list.sort, 112

max, 108  
min, 108  
sorted, 108  
Functions  
    argument unpacking, 135  
    default argument, 133  
    function argument, 133  
    global scope, 132  
    lambda function, 135, 136  
    local scope, 132  
    return, 133  
Input and output  
    close, 128  
    open, 127  
    read, 128  
    readlines, 128  
    write, 129  
    writelines, 129  
NumPy  
    np.abs, 139  
    np.angle, 139  
    np.append, 148  
    np.arccos, 138  
    np.arccosh, 138  
    np.arcsin, 138  
    np.arcsinh, 138  
    np.arctan, 138  
    np.arctanh, 138  
    np.around, 139  
    np.cbrt, 138  
    np.ceil, 139  
    np.concatenate, 148  
    np.conj, 139  
    np.copysign, 139  
    np.cos, 138  
    np.cosh, 138  
    np.deg2rad, 138  
    np.diagonal, 147  
    np.dot, 146  
    np.exp, 138  
    np.fftpack, 153  
    np.floor, 139  
    np.fmtpack, 153  
    np.hstack, 148  
                np.imag, 139  
                np.log, 138  
                np.max, 147  
                np.min, 147  
                np.outer, 147  
                np.prod, 147  
                np.rad2deg, 138  
                np.real, 139  
                np.reshape, 148  
                np.rint, 139  
                np.roll, 148  
                np.sin, 138  
                np.sinh, 138  
                np.sort, 148  
                np.sqrt, 138  
                np.sum, 147  
                np.T, 146  
                np.tan, 138  
                np.tanh, 138  
                np.trace, 147  
                np.trunc, 139  
                np.vstack, 148  
                numpy.diff, 153  
                numpy.loadtxt, 153  
                numpy.savetxt, 153  
Operators  
    Bitwise operator, 94  
    logical operators, 97  
    membership operators, 96  
    relational operators, 95, 96  
SciPy  
    scipy.fftpack, 353  
    scipy.integrate, 260  
    scipy.integrate.solve, 224  
    scipy.linalg, 300  
    scipy.linalg.eig, 301  
    scipy.misc.derivative, 224  
    scipy.optimize, 179  
String manipulation  
    len, 100  
    str.center, 102  
    str.count, 102  
    str.find, 102  
    str.join, 102

- str.lower, 102  
str.replace, 102  
str.split, 102  
str.strip, 102  
str.upper, 102  
Variables  
  bool, 89  
  complex, 89  
  float, 89  
  int, 89  
  str, 89
- QUADPACK, 260, 261  
Quadrature weight, 247, 248, 255  
Quantum dot, 461, 482–484, 487  
Quantum harmonic potential, 370, 385, 401, 412, 413  
Quantum number, 397, 444, 449, 457, 467, 479  
Quotient polynomial, 252
- Radial equation, 445, 446, 448–450, 457, 469–471, 483, 487  
Radial probability density, 445, 448, 449, 453, 467, 470, 474, 485  
Radix, 344  
Rayleigh quotient iteration, 277, 278, 306  
Remainder polynomial, 252  
Right eigenvector, 286, 287, 290, 292, 293, 302, 307, 308  
Root-finding, 167, 170, 178, 364, 366, 372, 376, 377  
Runge’s phenomenon, 247, 249  
Runge–Kutta method, 237, 240  
Runge–Kutta method, 365
- Sample mean, 259  
Schrödinger equation, ix, x, 231, 244, 245, 267, 288, 305, 338, 356, 361–363, 367, 370–373, 375, 376, 378–383, 386–389, 395–401, 404, 406, 408, 410–412, 461, 463–465, 480, 482  
Scientific binary notation, 15
- Secant method, 175, 176, 178–180, 385  
Self-consistent field, ix, 461, 462  
Shooting method, 363, 365–371, 375, 378, 379, 384–386, 389  
SHTOOL, 444  
Sign bit, 6, 9, 10  
Significant figure, 5, 6, 10–12, 21, 140  
Similarity transform, 269, 307, 397  
Simpson’s rule, 241–245, 250, 260, 264, 372, 403, 486  
Slater determinant, 480, 481, 487  
Sparse matrix, 305  
Spherical coordinates, 441, 442  
Spherical harmonics, 444, 468, 469  
Spin, 397, 454, 467, 479, 481  
Square integrable, 338, 370, 446  
Stability, xi, 279, 286, 287  
Stationary point, 173, 406, 463  
Stationary state, 362, 373, 378, 381, 405, 406  
Step size, 27, 61, 107, 109, 111, 262  
Stiffness, 287  
Sublinear convergence, 170  
Superlinear, 170, 175, 180  
Symmetric matrices, 275, 277, 301
- Taylor series, 81, 145, 163, 167, 169, 170, 232, 239, 241, 249, 250, 254, 321, 379, 390, 405
- Time-independent Schrödinger’s equation, 362, 363, 367, 368, 370, 379, 380, 382, 395, 399, 403, 482, 483
- Tolerance, 171, 172  
Trapezoidal approximation, 231–237, 239
- Trial wavefunction, 404–407, 414, 415, 466, 471, 480
- Tridiagonal matrix, 281–283, 303, 304, 308
- Unitary matrix, 270, 281
- Variational principle, 404–407, 414, 415, 463, 480

- Vectorization, 348  
Wavefunction, 231, 244–246, 311, 314, 316–320, 338, 345, 355, 361, 363, 371–374, 379–384, 391, 392, 395, 396, 401–407, 420–424, 449, 450, 457, 461–467, 479–482  
Wavelength, 312, 321, 322, 330, 482  
Wavenumber, 312, 315, 318, 322, 328–330, 336, 338  
Well-conditioned, 287  
Window function, 337, 338  
Zeno’s paradox, 178