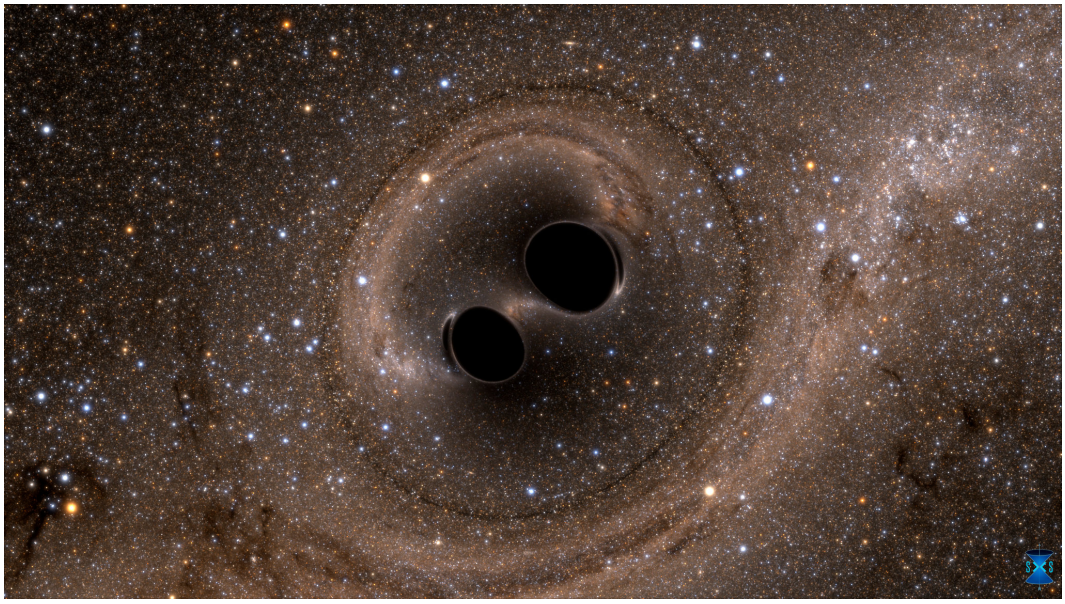
Patrick E. Farrell
University of Oxford

2023–2024
January 29, 2024

Computational Mathematics Student Handbook



A mathematical simulation of the merger of two black holes. Credit: SXS project.

Contents

1 Preface

This course is in two parts: Part I in Michaelmas Term Weeks 3–8 and Hilary Term Weeks 1–2, and Part II in Hilary Term Weeks 3–9. For Part I, you will attend scheduled practical sessions every fortnight starting in Week 3 Michaelmas Term. The practical sessions are held in the Mathematical Institute, Radcliffe Observatory Quarter, and are organised by college.

Each practical session will be run by a demonstrator and may include a short lecture. **You will need to bring your laptop to these sessions.** Please follow the instructions in Chapter 3 to install the necessary software **before** the first session.

There are four problem sheets for Part I. You will start problem sheet n during demonstration session n, and return your completed work during demonstration session $n + 1$ for marking during the session. Each problem sheet is associated with one main chapter of this handbook. This is summarised in the following table.

Weeks	Chapters to read	Problem sheet to start
1–2 MT	1–3	-
3–4 MT	4–5	1 (chapter 6)
5–6 MT	7–8	2 (chapter 9)
7–8 MT	10	3 (chapter 11)
1–2 HT	12–13	4 (chapter 14)

During Part I of the course, you may work collaboratively with others and—as always— you are encouraged to discuss mathematics and your studies with your peers. None of the work in Part I will be formally assessed. Instead, the material acts as a foundation enabling you to work individually during Part II. This individual work will be assessed and will count towards your Preliminary Examination as described in Examination Decrees & Regulations and the current Course Handbook. (See your college tutors if you have any questions about this aspect of the course.)

The course director is Prof. Patrick Farrell. The demonstrators will be able to answer most questions, but please feel free to contact

Prof. Farrell at patrick.farrell@maths.ox.ac.uk, especially with feedback on the course and edits to the course materials.

This manual and any additional course material can be found on the course website:
https://courses.maths.ox.ac.uk/course/view.php?id=4931

2 Introduction

2.1 What is computational mathematics?

Computational mathematics is the subject that studies the use of computation to solve mathematical problems. These mathematical problems may be solved for purposes within mathematics (for example, proving theorems, or finding counterexamples to conjectures), or beyond mathematics, in applications. Computational mathematics is therefore a very broad subject that cuts across the traditional divisions of pure and applied mathematics.

This is best seen with examples. In 1769, Leonhard Euler observed that it was possible to find natural solutions to

$$a_1^2 + a_2^2 = b^2, \quad \text{e.g.} \quad 3^2 + 4^2 = 5^2,$$

and to

$$a_1^3 + a_2^3 + a_3^3 = b^3, \quad \text{e.g.} \quad 3^3 + 4^3 + 5^3 = 6^3,$$

but that he could not find solutions to

$$a_1^3 + a_2^3 = b^3 \text{ (Fermat's last theorem)}$$

or to

$$a_1^4 + a_2^4 + a_3^4 = b^4.$$

He thus conjectured that you need at least three cubes to sum to a cube, four quartics to sum to a quartic, etc. Mathematically, Euler's Conjecture on the sums of powers is that

$$\exists\, k > 1, n > 1, a_1, \ldots, a_n, b \in \mathbb{N}_+ : a_1^k + a_2^k + \cdots a_n^k = b^k \implies k \leqslant n.$$

Euler's Conjecture remained open for nearly 200 years. In 1966, Lander and Parkin[1] computed a counterexample to Euler's conjecture:

$$27^5 + 84^5 + 110^5 + 133^5 = 144^5.$$

This was found by a direct search on a CDC 6600 mainframe. Leaving the computer to check for counterexamples overnight might save months or years of trying to prove a false statement.[2,3]

Leonhard Euler, 1707–1783

[1] L. J. Lander and T. R. Parkin. Counterexample to Euler's conjecture on sums of like powers. *Bulletin of the American Mathematical Society*, 72(6):1079, 1966

[2] Euler's conjecture is currently known to be true for $k = 3$, false for $k = 4$ and $k = 5$, and unknown for other values of k.

[3] Many more counterexamples found by clever computational mathematics can be found at https://math.stackexchange.com/questions/514/conjectures-that-have-been-disproved-with-extremely-large-counterexamples.

Computers are not merely useful for finding counterexamples, however. In 1852, Francis Guthrie was colouring a map of the counties of England, and noticed that only four colours were needed to satisfy the constraint that no two adjacent counties shared the same colour. Was this true for any map?[4]

In 1879, Alfred Kempe produced a clever proof[5]. He introduced the concept of what is now called an 'unavoidable set', a set of subgraphs such that every graph must contain at least one of them. If there exists a graph that cannot be coloured with four colours, then there must be a minimal such graph. For each subgraph in the unavoidable set, imagine removing that subgraph from the minimal non-colourable graph; since the result is smaller, it can be coloured with four colours. Kempe then reintroduced the removed subgraph and computed that for each element of the unavoidable set it was then possible to colour the whole graph with the partial four-colouring in hand. This reasoning showed that a minimal counterexample cannot exist, and that all graphs are four-colourable.

Unfortunately the proof was incorrect, as pointed out by Percy Heawood in 1890. Kempe's computations for the final element of his unavoidable set were wrong. However, the *proof strategy* was correct.

The first correct proof was given by Appel and Haken in 1976[6]. They devised an algorithm to build an unavoidable set such that a minimal counterexample could be ruled out in each case; computations were required both to build the unavoidable set, and to check that each case could be coloured with four colours. Whereas Kempe's unavoidable set had 6 elements, Appel & Haken's had 1834 (which has since been reduced). The proof was initially greeted with suspicion and dismay, and stimulated a great deal of philosophical debate on the nature of proof itself. Nevertheless, the proof is correct, and since then many major results have been first proven using extensive computer assistance, including the universality of the Feigenbaum constants, the Kepler conjecture on packing cannonballs, and Keller's conjecture on tiling Euclidean space with hypercubes.[7] Another important example was the computational exploration of the sporadic groups that arise in the classification of finite simple groups, one of the triumphs of 20$^{\text{th}}$ century algebra.

In fact, there is a substantial ongoing effort to develop automated proof checkers and automated theorem provers—computer programs that can automatically generate proofs of technical lemmas, or suggest relevant results, or identify flaws in published proofs. (These are more common in the literature than one would like.) In the coming years these systems will know all undergraduate- and graduate-level mathematics—will they revolutionise teaching and research over the course of your mathematical careers? Time will tell.[8]

[4] For a full history, see the book by Robin Wilson of Keble College on the subject:

R. Wilson. *Four Colors Suffice: How the Map Problem Was Solved*. Princeton University Press, 2013

[5] A. B. Kempe. On the geographical problem of the four colours. *American Journal of Mathematics*, 2(3):193–200, 1879

[6] K. Appel and W. Haken. Every planar map is four colorable. *Bulletin of the American Mathematical Society*, 82(5), 1976

[7] For a list, see `https://en.wikipedia.org/wiki/Computer-assisted_proof`.

[8] For more details on the current status and future potential of these systems, you might start with Kevin Buzzard's talk at the 2022 ICM. Or play the natural numbers game, if you are prepared to lose a weekend.

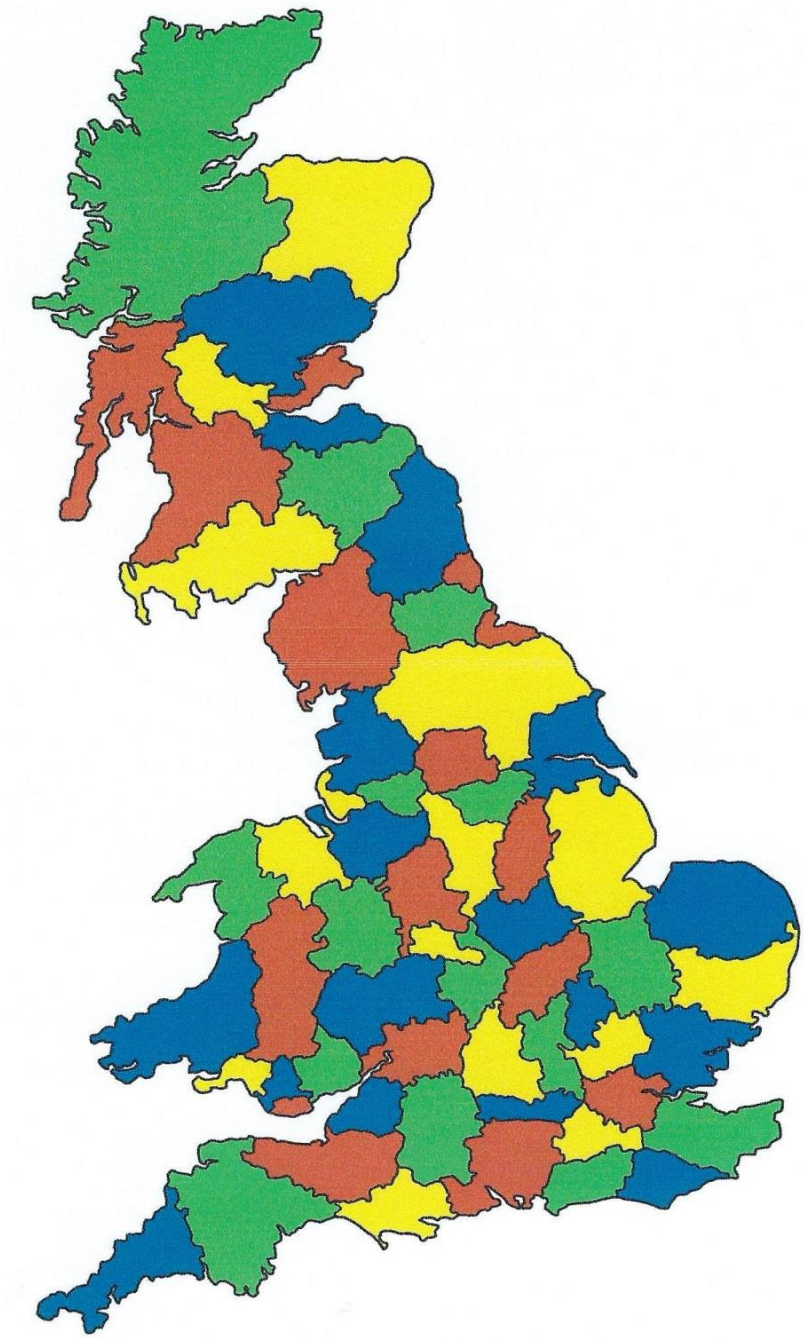


Figure 2.1: A map of the counties of Britain; four colours suffice. This image was taken from Robin Wilson's presentation on the history of the four-colour problem.

In applied mathematics, almost every triumph has involved computation. Here are some examples. The first programmable electronic digital computer, Colossus, was invented in 1943 by Tommy Flowers to aid with the cryptanalysis of the Lorenz cipher, used to communicate high-level strategic messages between Hitler and his generals. Colossus was used to identify the starting positions of two of the Lorenz system's wheels, by calculating statistics about the ciphertext for each of the 1271 possible configurations. Incorrect configurations gave true and false values appearing with frequencies close to 50%; possibly correct configurations produced measurable deviations from this. Promising configurations were then taken forward in further calculations to identify the starting positions of the other wheels. These computations enabled the Allies to decrypt the German High Command's messages within hours; the resulting intelligence made an incalculable contribution to the war effort, and saved many lives on both sides by shortening the war.

In the 18th century, astronomers formulated the Titius–Bode Law, which predicts the spacing between planets in the solar system. (The Law is now known to be a coincidence.) When Uranus was discovered by William Herschel in his garden in Bath in 1781, it was found to fit the Titius–Bode Law. The only undiscovered entry in the sequence predicted by the Law would correspond to an unknown planet between Mars and Jupiter. This caused many astronomers to search intensely for the missing planet.

In January 1801, Giuseppe Piazzi in Palermo discovered the planet Ceres between Mars and Jupiter, as predicted.[9] Unfortunately, he could only observe Ceres for 41 days before it vanished behind the sun, sweeping out an orbital angle of just 9°. The race was then on to predict where and when Ceres could next be observed. Carl Friedrich Gauss applied his method of least squares to estimate the orbital parameters of Ceres from Piazzi's data; the calculations were very involved, and required over 100 hours of computation. Ceres was rediscovered in December 1801, very close to where Gauss had predicted. This made the 24-year-old Gauss famous in the mathematical and scientific communities of Europe, and won him the position of the director of the Göttingen Observatory.[10]

One could write an entire book exploring important successes of computational mathematics. Sadly, such a book does not yet exist. We briefly mention: Richardson's calculation of the stresses in a masonry dam, pioneering methods which underpin every modern building[11]; the same Richardson's first attempt to predict the weather and climate by solving equations, which he did while serving in a Quaker ambulance unit on the Western Front in World War I[12]; Kantorovich's in-

Tommy Flowers, 1905–1998.

Carl Friedrich Gauss, 1777–1855.

[9] In the modern classification, Ceres is now considered a dwarf planet, because its gravity is not strong enough to clear its orbit.

[10] I have drawn this story from https://sites.math.rutgers.edu/~cherlin/History/Papers1999/weiss.html.

[11] D. R. Emerson, A. J. Sunderland, M. Ashworth, and K. J. Badcock. High performance computing and computational aerodynamics in the UK. *Aeronautical Journal*, 111(1117):125–131, 2007

[12] https://arxiv.org/abs/2210.01674

vention of linear programming, improving agricultural and industrial production in every sector of human endeavour[13]; Fermi & Ulam's invention of Monte Carlo to simulate neutron diffusion in fissionable material, for the development of fusion bombs[14]; the Kalman filter, which enabled the navigation of the Apollo spacecraft to the moon[15]; the invention of the Fast Fourier Transform by Cooley & Tukey to detect underground nuclear tests, and now used ubiquitously in audio, video, radar, and beyond[16]; Diffie–Hellman key exchange[17], which revolutionised cryptography; Hounsfield's invention of the CT scanner, which was funded by profits generated by the Beatles[18]; Fedorenko's invention of multigrid, which revolutionised physical modelling with partial differential equations, and has led to dramatic improvements in e.g. aircraft performance[19]; the insight of Page & Brin that searching the web can be cast as finding the dominant eigenvector of a matrix, which launched perhaps the most powerful company in the world[20]; the detection by LIGO of the gravitational waves induced by the merger of two black holes, which relied centrally on numerical simulation; and Hassabis et al.'s development of deep learning techniques for protein folding, crucial for understanding the function of proteins from its amino acid sequence[21].

[13] For a fascinating fictional account of Kantorovich's invention, and its impact on the 20th century competition between capitalism and communism, see

F. Spufford. *Red Plenty*. Faber & Faber, 2010

[14] https://library.lanl.gov/la-pubs/00326866.pdf

[15] https://www.lancaster.ac.uk/stor-i-student-sites/jack-trainer/how-nasa-used-the-kalman-filter-in-the-apollo-program/

[16] https://youtu.be/nmgFG7PUHfo

[17] https://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange

[18] https://catalinaimaging.com/history-ct-scan/

[19] A. Jameson. Time dependent calculations using multigrid, with applications to unsteady flows past airfoils and wings. In *10th Computational Fluid Dynamics Conference*. American Institute of Aeronautics and Astronautics, 1991

[20] https://en.wikipedia.org/wiki/History_of_Google

[21] A. W. Senior et al. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020

2.2 *Why should we study computational mathematics?*

Computational mathematics has revolutionised our world, and will continue to do so for centuries to come. It has minted billionaires, transformed old industries and created new ones, extended our lifespans; should you be a part of this?

Philosophically, for those who cannot program, computers will only do what *other people* have decided they should do; if you can program, your computer does what *you* decide.

As a practical matter, a core part of studying computational mathematics is learning to program efficiently. A very large fraction of Oxford mathematics undergraduates will pursue careers where knowledge of computer programming is useful, if not essential. This set of careers includes mathematical and scientific research, quantitative finance, teaching, data science, and management consulting.

2.3 *How should we study computational mathematics?*

One must learn computational mathematics by doing it—by iterating between ideas, algorithms, programming, experiments, and theorems. You can read an arbitrary number of books about playing the violin, but unless you practice, you will never be able to play yourself.

In this course, we will focus on the practical side of computational

mathematics. We will learn to program in the Python programming language. Python is a very popular choice for a first programming language to study, for several good reasons.

First, Python is one of the most popular programming languages in the world. If your future career involves programming, there is a very good chance it will involve programming in Python. This is especially so in the kinds of academic sectors and industries where Oxford graduates go. In particular, Python is the leading language in scientific data analysis and machine learning. Python's popularity also means there are large and well-established libraries for tasks like numerical computing or machine learning.

Second, Python's syntax is particularly simple and elegant. Programming involves spending as much time reading code as writing it; reading and understanding a Python program is (usually) much more straightforward than the corresponding program in a lower-level language.

Guido van Rossum, 1956–, the inventor of Python.

Third, Python is quick and easy to learn. Other programming languages allow users to do dangerous things like access out-of-bounds or deallocated memory; in pure Python these are not possible.

Fourth, Python is free/open-source software. You can study its implementation, scrutinise it for correctness, adapt it to run on any hardware, run as many instances as you please, all without paying anyone a penny.

The students coming to this course have a wide variety of backgrounds—some will be expert programmers, some will have studied Python programming in school, and some will have never programmed before. This course is intended to be accessible to everyone familiar with operating a computer.

Of course, Python is not perfect, and other programming languages exist for good reasons. In particular, the speed of execution of a Python program is not as fast as some other compiled languages like C, C++, or Fortran, or those that build-in just-in-time compilers like Julia and MATLAB. For many applications, the reduced speed of execution in Python is more than offset by the much faster speed of program development—and programmer time is much more expensive than computer time. A common model to overcome this is to first write the entire program in Python, then identify the core bottlenecks where most of the computation time is spent, and selectively write only these parts of the program (perhaps only a few lines) in a compiled language. The use of such facilities is beyond the scope of the course, but interested readers might investigate Numba[22].

Of course, the practical side is just one side of the subject. You will study a more theoretical view of algorithms, the core idea in computation, in Prelims *Constructive Mathematics* in Trinity term.

[22] S. Kwan Lam, A. Pitrou, and S. Seibert. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. Association for Computing Machinery, 2015

3 Getting started

This chapter describes how to get started with Python programming. You should follow the steps in this chapter *before* attending the demonstration sessions for this course.

3.1 Installing things

There are two pieces of software you need to install on your computer. Both are available for no cost for all common operating systems.

3.1.1 Installing Python

The first is Python itself. Go to `https://www.python.org/downloads/` and download the latest stable version for your operating system[1]. Run the downloaded executable to install Python on your machine. We now walk through the process of installing Python 3.11.15 on Windows 11 with screenshots.

[1] Some operating systems, especially Linux-based operating systems, will come with Python already installed. If you plan to use the system-installed Python, please check which version is installed with `python -V`; you will need a Python version greater than or equal to 3.6.

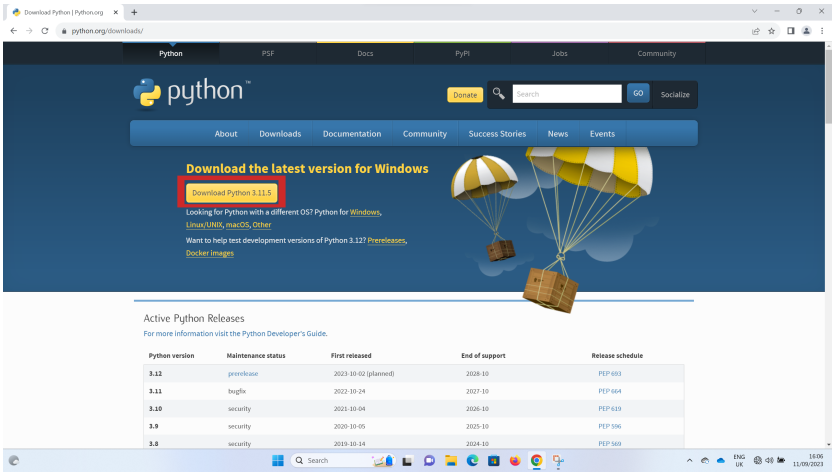


Figure 3.1: The Python download page, `https://www.python.org/downloads/`. The box you need to click on is highlighted in red.

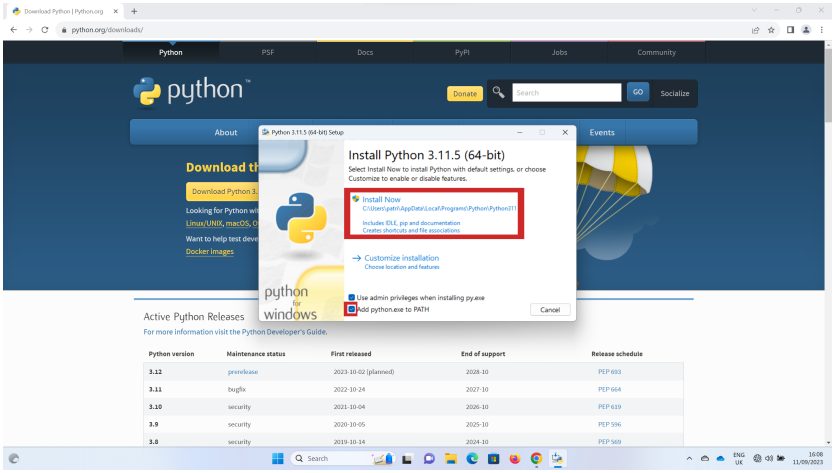


Figure 3.2: Execute the downloaded installer. Give it any permissions required by your operating system. Check the 'Add python.exe to PATH' button (or similar). Click 'Install Now'.

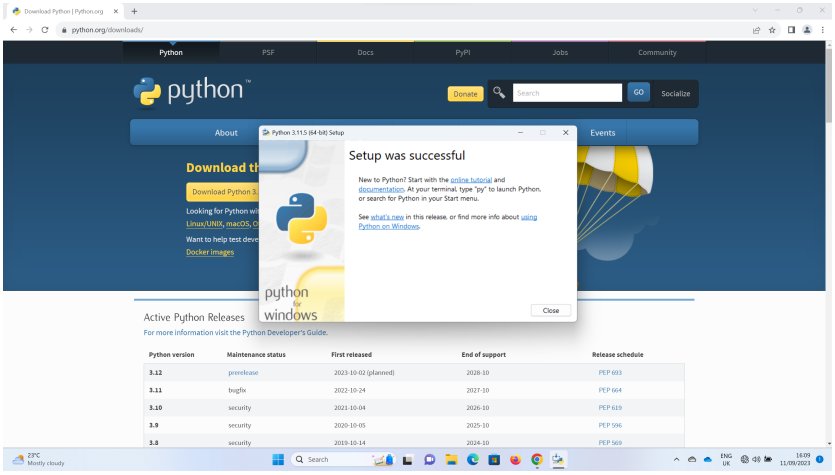


Figure 3.3: The installer should complete successfully.

We can now check if the Python installation has succeeded. We will open a terminal—this is a program on your computer where you can type commands for your computer to execute. On Windows, open the run dialog (with Windows key + R) and type 'cmd'. Select the 'Command Prompt' app, as shown in fig. 3.4. On a Mac, the easiest

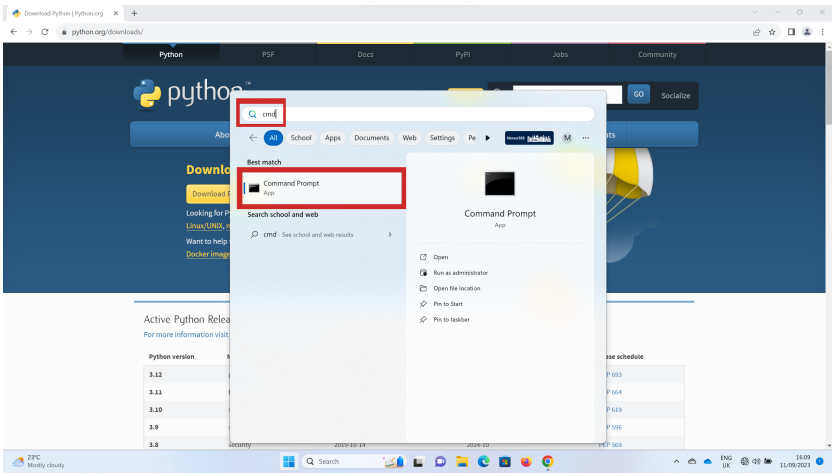


Figure 3.4: Opening a terminal on Windows: open the Run dialog (Windows key + R) and type 'cmd'. Select the 'Command Prompt' app.

way to open the Terminal is to open Spotlight Search (with Command + Space) and type 'terminal', as shown in fig. 3.5.

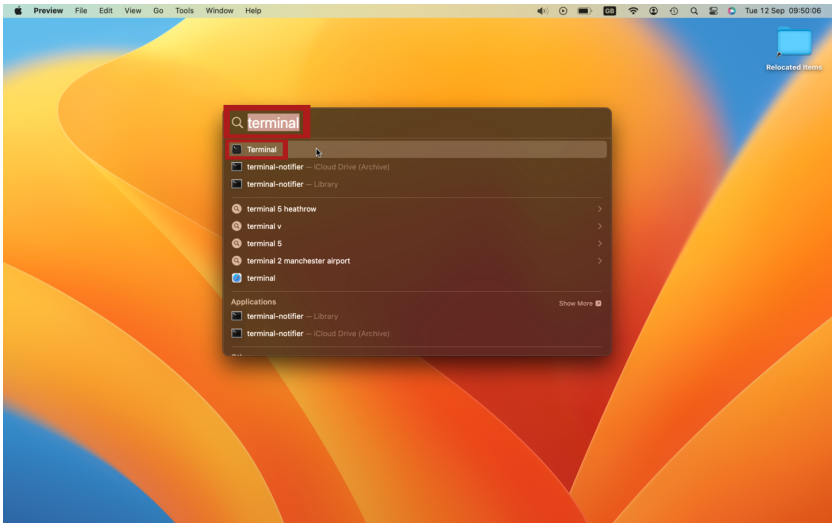


Figure 3.5: Opening a terminal on OSX: open Spotlight Search (Command + Space) and type 'terminal'. Select the 'Terminal' app.

We can now start the Python interpreter. At the terminal, type

(terminal) `python`

and hit enter[2]. The Python interpreter should start, as in fig. 3.6. (If it

[2] All commands to be typed at the terminal will be denoted with (terminal); you **do not type** (terminal).

doesn't, try running

(terminal) `python3`

or

(terminal) `py`

since, annoyingly, some operating systems call the Python interpreter something different.) Once you have the Python interpreter open,

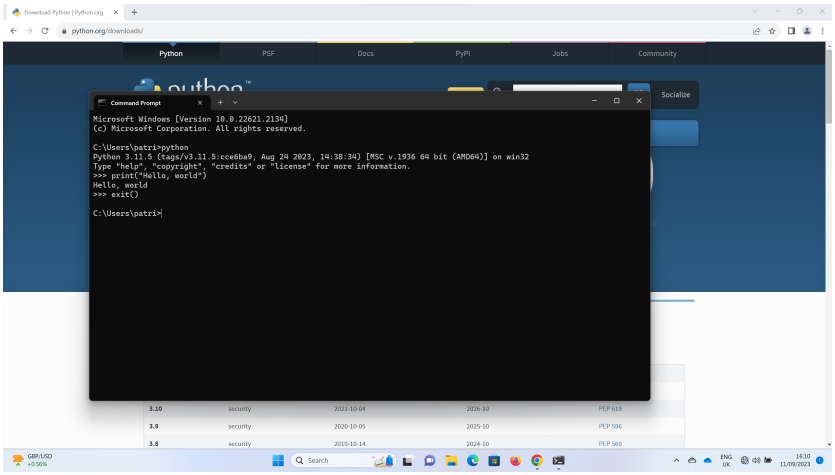


Figure 3.6: At the terminal, type `python` and hit enter. The Python interpreter should start. (If it doesn't, your operating system might have called it `python3` or `py`.) Typing `print("Hello, world!")` should cause Python to output this string to the terminal, as shown.

type

(python) `print("Hello, world!")`

and hit enter[3]. Type this exactly as written; one must be absolutely precise when programming, as programming languages do not understand ambiguity. It should cause the Python interpreter to print "Hello, world!" to the terminal. With this completed, type

(python) `exit()`

to close the Python interpreter.

You have now installed Python! Typing Python code into the Python interpreter is an excellent way to interactively explore features of the language and access documentation. It is not the main way we will run our code, however; we will describe that below.

While we are at the terminal, let us install one package, using the Python package manager `pip`. At the terminal, type

(terminal) `pip install ipython`

This will install an improved Python interpreter, IPython, that has

[3] All commands to be typed into the Python interpreter will be denoted with (python); you **do not type** (python).

code highlighting and tab completion, among other features. IPython has many dependencies on other Python packages; `pip` figures out these dependencies and installs them all for you. (Almost all Python packages are conveniently installable with `pip`, even large complex ones like the `tensorflow` or `pytorch` machine learning packages.) The output of the installation is shown in fig. 3.7.

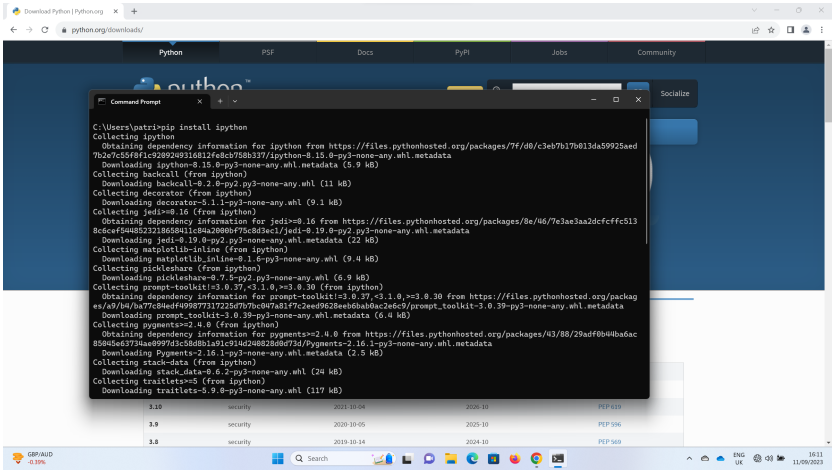


Figure 3.7: At the terminal, type 'pip install ipython' and hit enter.

We can now quickly test the IPython interpreter. At the terminal, type

(terminal) `ipython`

or possibly

(terminal) `ipython3`

The IPython interpreter should start. You can then type the same Python code

(python) `print("Hello, world!")`

and it should be printed to the terminal as before, as shown in fig. 3.8. The most visible difference between the standard Python interpreter and the IPython interpreter is that the latter uses colours to help us quickly read the code. You can then type

(python) `exit()`

to close the IPython interpreter, as before.

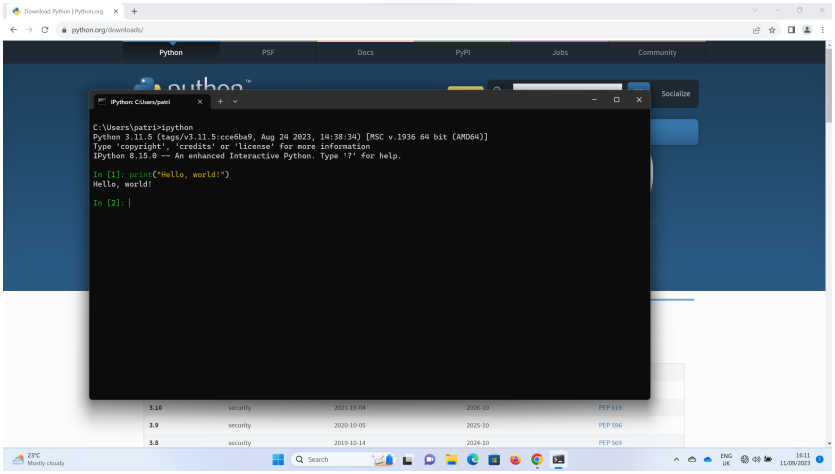


Figure 3.8: At the terminal, start IPython and type `print("Hello, world!")`.

3.1.2 *Installing Visual Studio Code*

The main way we will execute our code is by *saving it to a file* and executing the Python interpreter on it at the terminal. For example, if we save our Python code to a file `hello.py`[4], we can execute it with

(terminal) `python hello.py`

[4] Files containing Python code should always end with the '.py' suffix, to let your operating system know what it contains.

Now, you are free to write Python code in whatever program you like, so long as it can save plaintext files to disk. Basic choices might be Notepad on Windows or TextEdit on Macs. However, most programmers use specialist programs to edit their code, as they can offer many convenient features for programming—code highlighting, in-built access to documentation, integration with source code control, graphical interfaces to debugger programs, and more besides. Some popular choices for these code editors include Atom, Vim[5], Emacs, Spyder, and Sublime Text. If you wish to use one of these (or any other code editor) for this course, you are free to do so. However, if you are a novice programmer, it is recommended that you use Visual Studio Code.

[5] Prof. Farrell uses Vim.

Visual Studio Code is the most popular code editor in the world (by some reasonable measures). We will show screenshots of the installation process on Windows 11; installing on OSX should be analogous. To start, go to `https://code.visualstudio.com/download` and download the version for your operating system (fig. 3.9). Start the downloaded installer (fig. 3.10). When it has finished, launch Visual Studio Code (fig. 3.11). On the first execution of the program, Visual Studio Code opens a screen letting you set some options (like colours) and shows you some features (fig. 3.12). Feel free to browse around; when finished, click 'Mark Done'. The standard welcome page should now be visible (fig. 3.13). Click 'Open Folder'. Choose a

folder to store your *Computational Mathematics* coursework in; I created a new directory called 'Computational Mathematics' on my Desktop (fig. 3.14–fig. 3.15). Visual Studio Code will ask if you trust the authors of code in this folder; click 'Yes, I trust the authors' (fig. 3.16). You can now create a file in your directory with the icon indicated on fig. 3.17. Call the resulting file 'hello.py'. The file will now be opened in a new tab (fig. 3.18). Visual Studio Code notices you are coding in Python and asks if you wish to install the relevant extensions for the Python language. Click 'Install' in the bottom right. Visual Studio Code should now install the Python extension (fig. 3.19). Close the tab when it has installed.

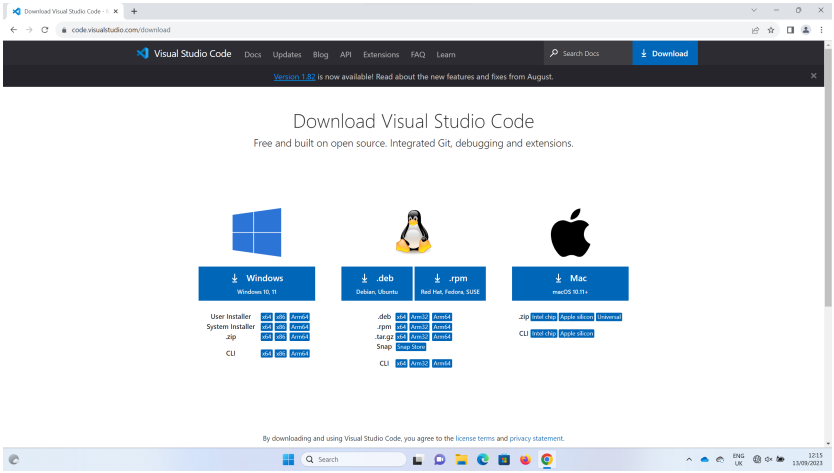


Figure 3.9: Open `https://code.visualstudio.com/download`. Download the version appropriate for your operating system.

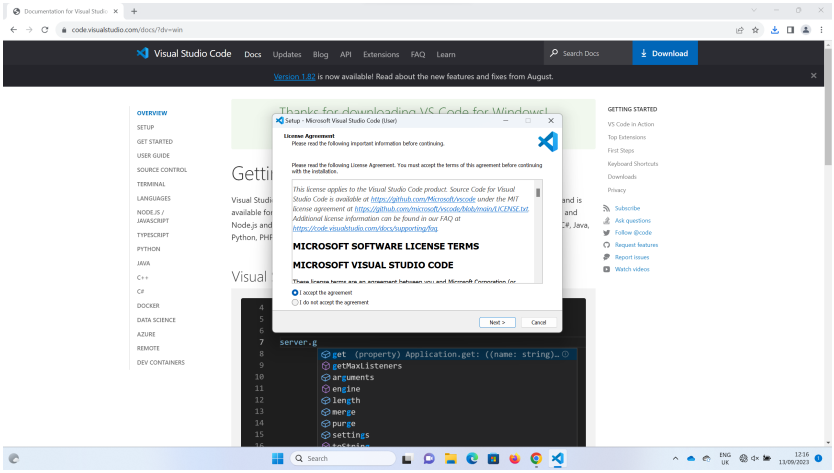


Figure 3.10: Start the downloaded installer. Agree to the license terms and click Next.

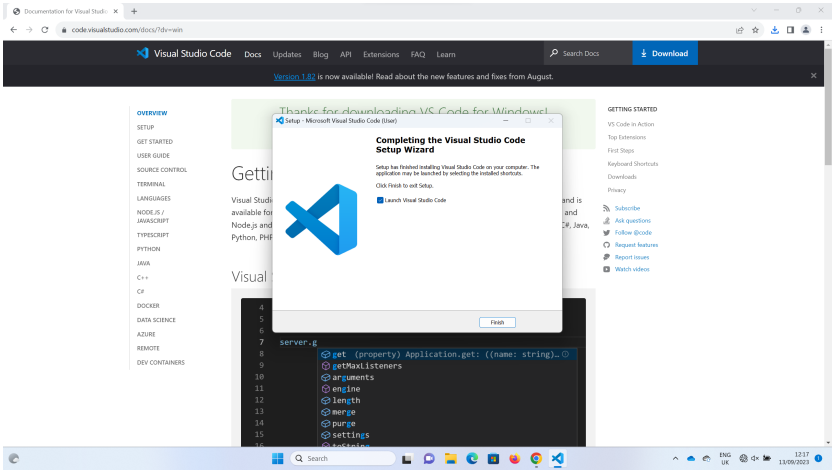


Figure 3.11: When the installer has finished, launch Visual Studio Code.

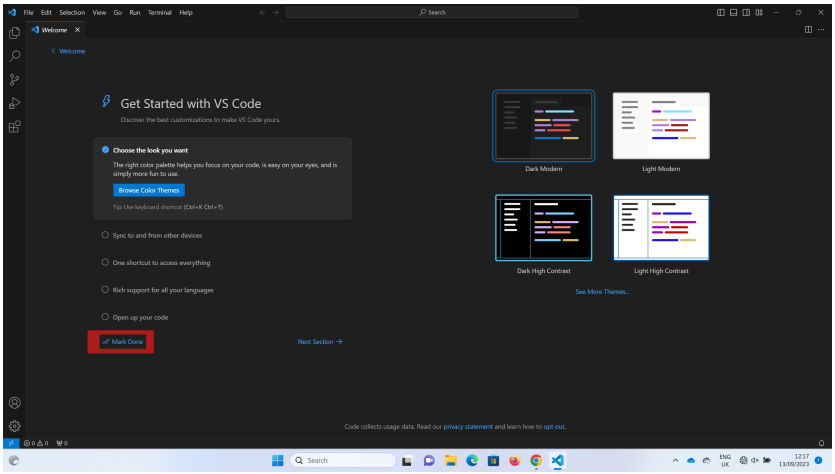


Figure 3.12: Visual Studio Code shows a tour on first execution. Browse around; when finished, click 'Mark Done'.

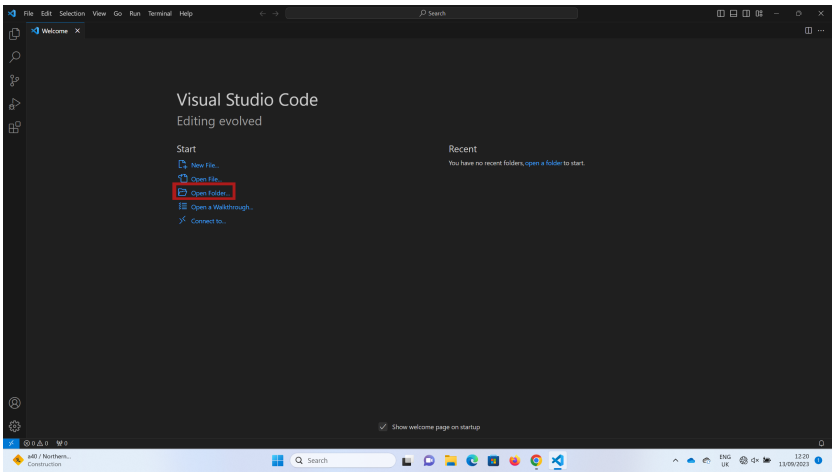


Figure 3.13: The standard welcome page. Click 'Open Folder'.

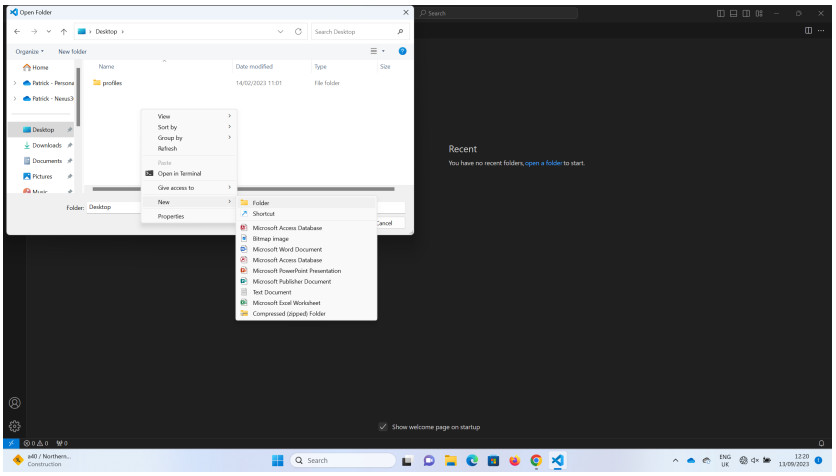


Figure 3.14: Choose a folder to store your code; I created a new folder on my Desktop called 'Computational Mathematics'.

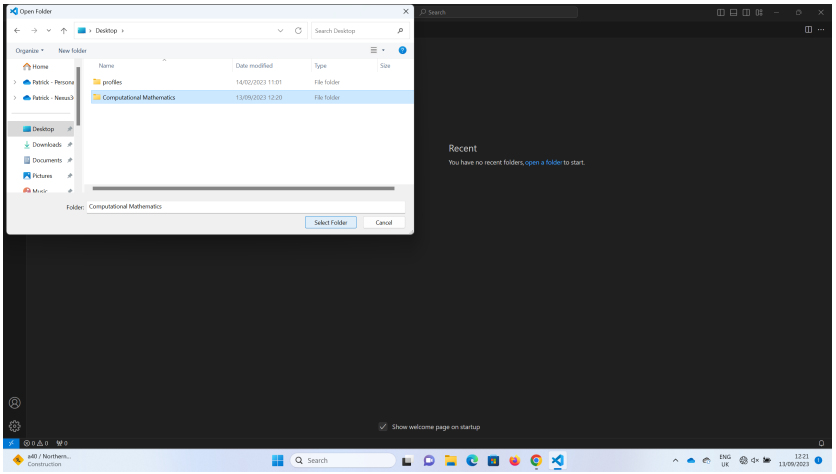


Figure 3.15: Select your chosen folder.

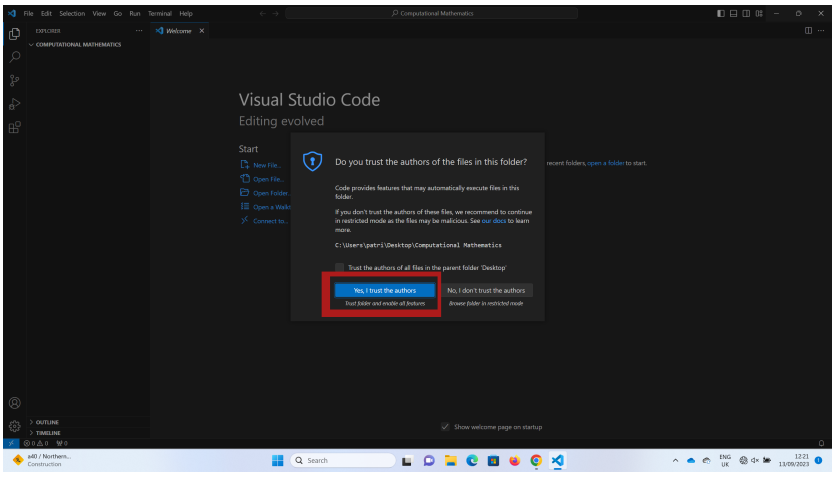


Figure 3.16: Choose 'Yes, I trust the authors' when asked.

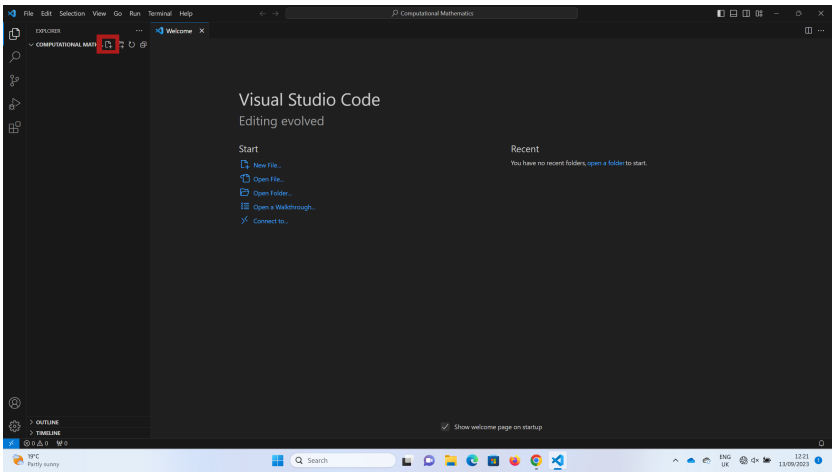


Figure 3.17: Click the indicated button to create a new file. Call the file 'hello.py'.

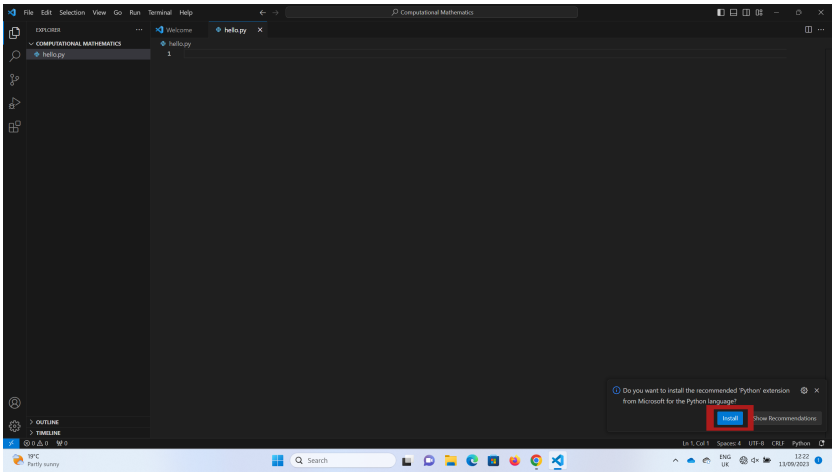


Figure 3.18: Your new file is opened in a new tab. Visual Studio Code notices you are coding in Python and asks if you wish to install the relevant extensions for the Python language. Click 'Install' in the bottom right.

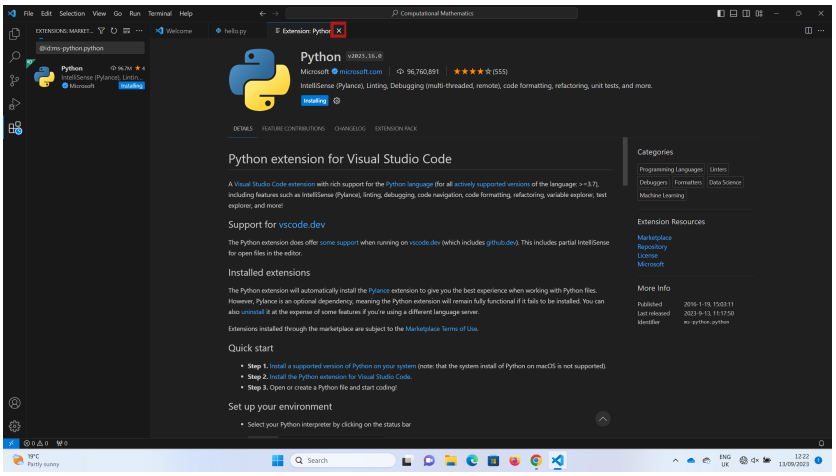


Figure 3.19: Visual Studio Code now installs the Python extension. Close the tab when it has installed.

We can now proceed to writing our first script. In 'hello.py', type the code[6]

[6] Programs intended to be saved to a script and executed are typeset as in code block 3.1.

Code block 3.1. Our first code: printing a string.

```
print("Hello, world!")
```

When you type `print`, Visual Studio Code should bring up the Python documentation for the function (fig. 3.20). Once you have typed your code, save the file (fig. 3.21). Then open a terminal by going to 'Terminal' > 'New Terminal' (fig. 3.22). At the terminal, type

(terminal) `python hello.py`

which should print to the screen as before (fig. 3.23).

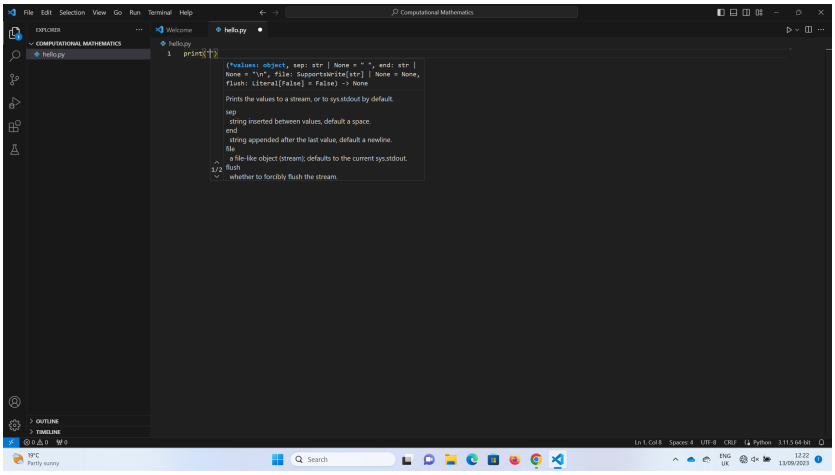


Figure 3.20: Visual Studio Code offers inline documentation relevant to the code you are writing.

Congratulations! You are now ready to begin.

If you have any problems installing Python or Visual Studio Code, first google it; these are popular programs and there will be a great deal of help online. If that is not satisfactory, your demonstrator will assist you in the first demonstration session.

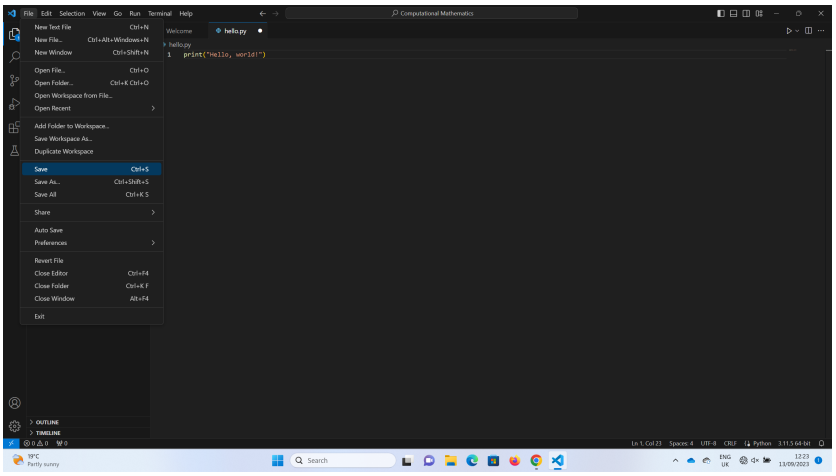


Figure 3.21: Once you have typed your code, save the file.

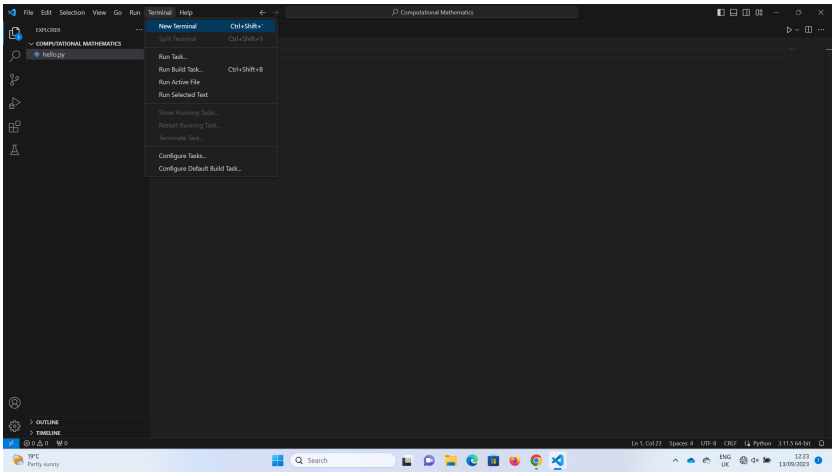


Figure 3.22: Open a terminal to execute your program.

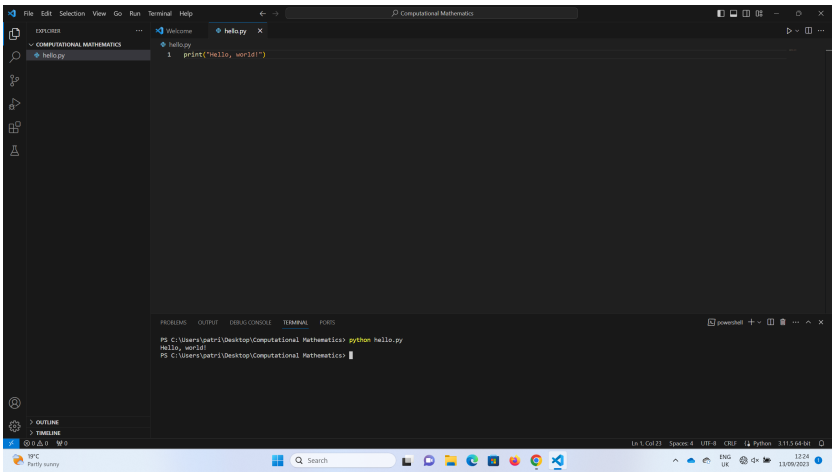


Figure 3.23: The program should execute, printing the message to screen.

3.2 *Other resources*

Python is a very popular programming language, and there are *many* resources online and in print to help you learn it. We mention some here.

The primary textbook recommended for this course is Hill (2020)[7]. This book goes at just the right pace, is comprehensive, and is aimed at a mathematically and scientifically literate audience.

[7] C. Hill. *Learning scientific programming with Python*. Cambridge University Press, second edition, 2020

We now mention some resources available at no cost. Microsoft have prepared a series of short video lectures introducing Python. These videos also use Visual Studio Code. They are available on YouTube at

https://www.youtube.com/playlist?list=PLlrxD0HtieHhS8VzuMCfQD4uJ9yne1mE6

Jake VanderPlas of Google has written two open-source books about Python[8]. They are available at no cost at

https://jakevdp.github.io/WhirlwindTourOfPython/ and
https://jakevdp.github.io/PythonDataScienceHandbook/

[8] J. VanderPlas. *A Whirlwind Tour of Python*. O'Reilly Media, Inc., 2016; and J. VanderPlas. *Python Data Science Handbook*. O'Reilly Media, Inc., 2016

The first is a brief introduction to the Python programming language for those with some programming background, while the latter focuses on the use of Python for data science.

For an introduction to Python suitable for scientists and mathematicians with no programming experience, see the excellent book by Hans Petter Langtangen[9]. It is available at no cost at

https://hplgit.github.io/primer.html/doc/pub/half/book.pdf

[9] H. P. Langtangen. *A Primer on Scientific Programming with Python*. Springer Berlin Heidelberg, 2016

This book uses Python version 2 (rather than version 3 of Python that we will study), so the syntax is *very slightly* different[10], but there is great value in the gentle pace and mathematical examples.

A good resource for a *second* course on programming is that by David Ham[11], which is available at no cost at

https://object-oriented-python.github.io/

Chapter 10 of this handbook considers symbolic computing using the SymPy library. A short course of video lectures on SymPy is available at no cost at

https://www.youtube.com/playlist?list=PLSE7WKf_qqo1T5VV1nqXTj2iNiSpFk72T

[10] The main difference that matters to us is that in Python 2 they used to say

```
(python)  print "Hello!"
```

instead of

```
(python)  print("Hello!")
```

The other difference that matters to us is that integer division was denoted differently.

[11] D. A. Ham. *Object-oriented Programming in Python for Mathematicians*. 2023. Independently published

3·3 *Thinking like a programmer*

Before diving in to the nuts and bolts of how one programs in Python, a few words about how programmers think are in order. The major obstacle to learning how to program is not mastering the syntax or standard library of your chosen programming language; it is learning a new mode of thinking, of *how to think like a programmer*. This manner of thinking shares quite a few properties with mathematical thinking. We outline some attributes of this manner below.

Programming requires an extreme degree of precision. The difference between `"Hello, world!"` and `"Hello, world!'` would likely not be noticed by most readers in a large block of text. But to a computer these are completely different; the first is a syntactically-valued string, whereas the second is not. Printing the first will produce the desired output, while attempting to print the second will cause the Python interpreter to complain that a string has been opened with `"` but not closed. Another example might be the difference between

(python)
```
3*5 + 8 - 3 + 2**3
```

where `**` means exponentiation (i.e. `2**3` $= 2^3$), and

(python)
```
3*5 + 8 - 3 + 2^3
```

where `^` is a totally different operator (logical exclusive or[12]). The first evaluates to 28, the second to 21 (and is likely a subtle mistake that will be very hard to debug). Similarly, a program that loops

(python)
```
while sum(values) < 100:
```

may be incorrect in edge cases if what was desired was actually

(python)
```
while sum(values) <= 100:
```

In mathematics, one missing or misplaced subscript, or one logical quantifier in the wrong place, can completely change the meaning (and validity) of a statement. Programming demands the same kind of pedantry and meticulousness.

Programming also requires specifying an answer to a problem to the last detail. In writing proofs of theorems, we can sometimes elide mechanical details, usually with some exclamation of how simple the omitted details are: ''By induction, it is straightforward to show that ...'', or ''It is simple to generalise Lemma 4.3 to the case that ...''. (The omitted steps are often far from simple.) By contrast, the computer will not let us elide any details of any kind. We must express every detail of the solution of the problem we are facing in terms of operations

[12] This operation takes two Boolean values, true or false, and returns true if exactly one of them is true. Applied to an integer, it does this bitwise on its representation in base two. We will not use it on our course.

that the machine already knows (either coded by us or by others). No vagueness is permitted. Thinking like a programmer requires exhaustiveness and persistence.

Programming requires patience. It can be frustrating when the computer does not do what you want it to. Debugging programs requires thinking like a detective—forming hypotheses, gathering evidence, ruling out possibilities, until finally the culprit is identified.

Programming requires being literal and specific. Computers lack any common sense whatsoever[13]. Keep in mind that computers will do exactly what you program them to, *not* what you wanted them to do—and expressing the latter can be surprisingly difficult[14]. A famous example of this was the failure of the maiden launch of the Ariane 5 rocket in 1996. The rocket was readied on the launchpad at the Centre Spatial Guyanais in French Guiana. It launched on a different trajectory with much greater acceleration than its predecessor, the Ariane 4. The flight computers and software responsible for monitoring speed and orientation aboard the Ariane 5 were reused from the Ariane 4, but the greater speed of the new rocket caused the computers to experience a so-called ''hardware exception'' when converting a 64-bit floating point number to a 16-bit integer. This caused the numbers stored in the computer to immediately flip sign from 32,768 to -32,768, confusing the control system and initiating a sudden turn downward that resulted in a catastrophic breakup and aerial explosion. Both the rocket and its payload were destroyed[15]. The onboard flight computer did exactly what the progammers designed it to do, but that was not what was desired at all!

As New College's Richard Dawkins wrote[16],

> If you don't know anything about computers, just remember that they are machines that do exactly what you tell them but often surprise you in the result.

[13] This is also true for artificial intelligence, at least for now. Consider the DARPA robot that was trained to identify humans approaching it. It was fooled by humans creeping up to it under a cardboard box, giggling as they went. See https://www.extremetech.com/defense/342413-us-marines-defeat-darpa-robot-by-hiding-under-a-cardboard-box.

[14] Indeed, machine learning first revolutionised computational tasks precisely where it is difficult to specify mathematically what you really want—image classification, language translation, speech recognition, and the like.

[15] See https://en.wikipedia.org/wiki/Ariane_flight_V88 and https://www-users.cse.umn.edu/~arnold/disasters/ariane.html

[16] R. Dawkins. *The Blind Watchmaker*. WW Norton & Company, 1996

4 Arithmetic, conditionals, and iteration

4.1 Arithmetic

Let us begin our study of Python with arithmetic. In mathematics, we study different kinds of numbers: $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{C}$, and we consider these in turn. Integers are represented in Python in the obvious way. Open up an IPython interpreter in the terminal and try the following[1]:

```
(python)  5 + 3
           8
(python)  9 / 2
           4.5
```

[1] The output is coloured in grey.

Here we see that the integers in Python are not closed under division, as in mathematics[2]; the division of two integers has returned a *floating point number* (a representation of real numbers used by almost all computers). In many programming languages, the data structure used for integers can only represent a maximum value[3], but Python's integer data type can represent integers of any size so long as your computer has enough memory:

[2] Confusingly, this is not true in many programming languages, including C and the old version 2 of Python.

[3] For example, an `int` in the C programming language can usually hold numbers no bigger than 2147483647.

```
(python)  import math
(python)  math.factorial(10)
           3628800
(python)  math.factorial(100)
           933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

Here we have imported our first *module*, the `math` module. The `math` module is built-in (it always comes with Python) and implements many of the standard functions you would expect a computer to know, like exponentiation, logarithms, trigonometric functions, etc.[4].

[4] For a list, see https://docs.python.org/3/library/math.html. To find this in the IPython shell, type `math.` and press TAB on your keyboard (without pressing enter). Another way is to type `help(math)`.

`math.factorial` accesses the `factorial` function that is defined in the `math` module, and (unsurprisingly) calculates $n!$ for given n. The function is called by passing its arguments in brackets, as in mathematical notation.

How much memory does it take to store these integers? Let us ask:

```
(python)  math.factorial(10).bit_length()
           22
(python)  math.factorial(100).bit_length()
           525
(python)  math.factorial(1000).bit_length()
           8530
(python)  type(math.factorial(1000))
           int
```

Everything in Python is an *object*; each object has a type (in this case, the type is `int`). Objects have functions associated with them, called *methods*. Here we are calling the `bit_length` method on instances of the `int` type to determine how many bits (binary digits) are required to store them[5]. You can determine the type of any object with the function `type`.

[5] The computer I am typing this on has 1610838310912 bits of memory, so we could store much larger integers if we wish.

The other common arithmetical operations on integers work as you would expect:

```
(python)  2 + 5
           7
(python)  2 * 5
           10
(python)  2**5
           32
```

where `**` means exponentiation. The symbols for two other useful operations might be less familiar. The symbol `//` calculates the integer part of division, and `%` takes the remainder on division[6]:

```
(python)  3 // 3
           1
(python)  4 // 3
           1
(python)  5 // 3
           1
(python)  6 // 3
           2
(python)  7 // 3
           2
(python)  4 % 3
```

[6] Mathematically, if

$$a = qr + b,$$

then in code `q = a // b` and `r = a % b`.

```
          1
(python)  5 % 3
          2
(python)  6 % 3
          0
```

With these, and Python's facility with large integers, we can verify Lander & Parkin's counterexample to Euler's Conjecture:

```
(python)  27**5 + 84**5 + 110**5 + 133**5 - 144**5
          0
```

Let us take a digression. What happens if we attempt to divide by zero?

```
(python)  3 / 0
          ---------------------------------------------------
          ZeroDivisionError Traceback (most recent call last)
          <ipython-input-3-e1965806ec03> in <module>
          ----> 1 3 / 0
          ZeroDivisionError: division by zero
```

Python raises an *exception*. Exceptions are errors generated at runtime. A mechanism exists for catching them and handling the error state gracefully without stopping the program's execution, which we won't go in to here. The particular kind of exception seen here is a `ZeroDivisionError`, which unsurprisingly is raised when one attempts to divide by zero[7]. Notice how the error message attempts to point (with `---->`) to the line of code where the exception was raised (which is line number 1, since IPython feeds lines of code to the interpreter one at a time).

[7] Python's error handling makes it impossible to ignore erroneous program states, and is one of the features that makes it very suitable for new programmers. By contrast, in lower-level languages like C, errors like this can propagate silently until the entire program ends in disaster, and e.g. your rocket explodes.

Another kind of exception you might see is a `SyntaxError`:

```
(python)  3 /
            File "<ipython-input-1-62321043ab1b>", line 1
            3 /
              ^
          SyntaxError: invalid syntax
```

which reports that the code we entered does not conform to the grammar of the language: `/` is a binary operation that must take in two inputs, but we have only supplied one.

Exercise 4.1. Compute[8] (as an `int`, without casting to a string) the last ten digits of 55^{55}.

[8] We recommend that you write your program for this and all other exercises in a script, rather than at the interactive Python interpreter. Call it `ex41.py`.

Let us now consider how a computer represents real numbers. Unlike integers, where an exact representation is possible (so long as your computer has enough RAM inside), in principle no exact representation of all reals is possible on a discrete computer, one that uses ones and zeros to represent information[9]. No finite amount of memory can represent all digits of π, for example. Floating point numbers represent a given real number to a certain precision, typically to 15 or 16 decimal places[10]. For example, the number 4/3 is stored as

$$1.33333333333333325931846502499 0\cdots,$$

which is close but not quite the same. Floating-point arithmetic has been wildly successful at representing real numbers on computers, and is the bedrock underpinning essentially every scientific and engineering calculation since the 1980s. Encoding and decoding your voice, music, photographs and videos on your phone, performing large-scale climate simulations, finding a location from GPS satellites, training a neural network, and playing computer games all involve billions of floating point operations.

Let us see how this works:

```
(python)  5/3
           1.6666666666666667
(python)  math.pi
           3.141592653589793
(python)  math.sqrt(2)
           1.4142135623730951
(python)  math.sin(0)
           0.0
(python)  math.sin(math.pi)
           1.2246467991473532e-16
```

It's a little tedious to type `math.sin(math.pi)`, so we can instead do

```
(python)  from math import *
(python)  sin(pi)
           1.2246467991473532e-16
```

which imports everything from the `math` namespace to our working namespace.

Python interprets anything with a `.` as a floating-point number:

```
(python)  type(4)
           int
(python)  type(4.0)
```

[9] This relates to Cantor's proof of the uncountability of the real numbers.

[10] With the most popular standard for floating-point arithmetic, in the interval $[1,2]$ there are $2^{52} \approx 10^{16}$ floating point numbers. By contrast, along a line of 1 metre length in a solid block of concrete there are approximately 10^9 particles; computer arithmetic is roughly a million times finer than the continuum approximation in physics. See

L. N. Trefethen. Floating point numbers and physics. *Newsletter of the London Mathematical Society*, November, 2021

William Kahan, 1933–, the primary architect of floating-point arithmetic. Kahan was awarded the Turing Prize for this work.

```
float
```

You can also express scientific notation using the notation `e` or `E`: `5e3` or `5E3` both mean 5×10^3.

```
(python) 1e1
          10.0
(python) 1E2
          100.0
(python) 314.1592653589793e-2
          3.141592653589793
```

You can *cast* (change the type of a number) by calling the relevant type on it:

```
(python) float(4)
          4.0
(python) int(4.0)
          4
(python) int(4.9)
          4
```

We see that `int` always rounds down. To round a number, use the function `round`[11].

To close this section we study the complex numbers. The imaginary unit is denoted `1j` in Python[12]. Complex numbers can be made by adding real and imaginary parts, like so:

```
(python) 4 + 3j
          (4+3j)
(python) type(4 + 3j)
          complex
(python) (4 + 3j).real
          4.0
(python) (4 + 3j).imag
          3.0
(python) (4 + 3j).conjugate()
          (4-3j)
(python) abs(4 + 3j)
          5.0
```

The object returned by the expression `4 + 3j` is of type `complex`. Complex numbers in this format always store their real and imaginary parts as floating point numbers (even when we pass integers). You can access the real and imaginary parts with the object *attributes* `real` and `imag`. We can calculate the conjugate of a complex number by calling the `conjugate` method. `conjugate` is a function that takes

[11] This implements so-called banker's rounding, which breaks ties by rounding to the nearest even integer.

[12] Python adopted this convention from engineering, as opposed to the mathematical convention where it is denoted i.

no arguments, which is we we call it with open and closed brackets[13]. Lastly, we can calculate the magnitude of a complex number with the built-in `abs` function, which also works on `int`s and `float`s.

[13] If you write `(4+3j).conjugate`, you are referring to the method itself, which is an object in its own right.

Exercise 4.2. Calculate $(5-8i)^2$, i^{-1}, i^2, $i^{1/2}$, $\frac{1+i}{5+2i}$, and $\left(\frac{4}{3}+\frac{2i}{5}\right)^4$.

4.2 *Variables*

When an object is created in a Python program (or in the Python interpreter), Python allocates memory for it. In other languages you have to manage the computer memory explicitly for yourself, but Python takes care of all of this for you, making it much more user-friendly[14]. If you wanted to find the *address* in memory where something is stored, you can use the `id` function:

[14] For example, Microsoft announced in 2019 that about 70% of the security vulnerabilities it fixes in its products each year are to do with managing memory; see `https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/`. This class of bugs simply cannot happen in pure Python.

```
(python)  id(9.81)
          139945303740048
```

where the output will almost surely differ if you run it again. When we type this, the memory required to store `9.81` is allocated, the value is assigned, and the value is passed to the `id` function. Once the `id` function has returned, Python notices that nothing refers to this memory anymore, and so the memory is automatically deallocated for you.

Variables allow you to store objects and to refer to them. These are necessary for any program more complex than simple arithmetical calculation. A variable name can be assigned to an object using the = symbol:

```
(python)  g = 9.81
(python)  g
          9.81
```

Here you should read = as 'gets', rather than 'equals'; it assigns the right-hand value to the left-hand variable. The memory required to store this value will persist for as long as the variable `g` exists. You can then use this in subsequent calculations, combined with other variables:

```
(python)  L = 2
```

```
(python)  period = 2 * pi * sqrt(L/g)

(python)  period
          2.837006706885775
```

Notice that we do not have to declare variables in advance, or declare their types; Python is a so-called *dynamically-typed* language, where the programming language infers the types of objects for us.

At this point we should describe some rules around variables. You cannot use as a variable name any *reserved keywords*, words that Python uses to define its grammar[15]. Variable names are case-sensitive: `g` and `G` are different variables. Variable names can include letters, digits, and underscores, but cannot start with a digit.

There are also some recommendations to be made around variables. Variable names should be meaningful (`vol` or `volume` are better than `v`) but not overly verbose (`volume_of_my_cube` is cumbersome). Use lower-case words separated by underscores for variable names, so prefer `avg_score` over `AvgScore`[16].

One way to make sure that you remember what each variable is for is to add a *comment* to your source code. Comments begin with the `#` key; anything to the right of the `#` is ignored by Python. For example, you might write

```
(python)  g = 9.81 # acceleration due to gravity
```

Frequent and judicious use of comments is core to good programming: a program has to convey information both to the computer (what instructions to execute) but also to other programmers, since code will need to be maintained and updated and improved as time goes on[17].

[15] So far we have met **`import`** and **`from`**. Others include **`for`**, **`while`**, and **`if`**; we will meet them as we continue our study. For a full list, see `https://docs.python.org/3/reference/lexical_analysis.html#keywords`. In mathematical programming, the only clash that sometimes annoys us is **`lambda`**, which Python uses to define anonymous functions.

[16] This is formalised in PEP8, the official style guide for Python. Many large Python projects use automated tools to enforce compliance with PEP8.

[17] The 'other programmer' may well be you in six months' time, when you have forgotten what on earth you were thinking of when you wrote the code in front of you.

Exercise 4.3. [Taken from Hill (2020)[18], exercise P2.2.4]

The World Geodetic System is a set of international standards for describing the shape of the Earth. In the latest WGS-84 revision, the Earth's geoid is approximated by a reference ellipsoid that takes the form of an oblate spheroid with semi-major and semi-minor axes

$$a = 6378137.0 \text{ m and } c = 6356752.314245 \text{ m}.$$

Use the formula for the surface area of an oblate spheroid,

$$S = 2\pi a^2 \left(1 + \frac{1-e^2}{e}\operatorname{atanh}(e)\right), \quad \text{where } e^2 = 1 - \frac{c^2}{a^2}$$

to calculate the surface area of this reference ellipsoid. Compare it to

[18] C. Hill. *Learning scientific programming with Python*. Cambridge University Press, second edition, 2020

the surface area of a spherical model of the Earth, with radius 6371 km.

4.3 *Accessing documentation*

It is very easy to access documentation about a given Python object. In the IPython interpreter, try

```
(python)  help(math)
```

(you will need to have imported the math module first) or

```
(python)  help(float)
```

You will not understand everything the documentation says at first, but you will understand more as you learn more Python.

You can also access documentation in Visual Studio Code. For example, in the code editor, if you have `import math` in your code and then type `math.`, it will list all functions in the `math` module. If you choose one (say `math.sin`) and then type `(` (i.e. open brackets to call the function), Visual Studio Code shows you the documentation for that function.

Google is also an excellent source of help; for example, if you get an error message you don't understand, or want to know how to do something, googling for it will likely get you a long way[19].

The problem sheets have been designed so that you have *almost* all information required to solve the problems, but may have to google to see how a key method is used, or to consult the documentation for a package. This is like real life—you rarely have at the outset *all* the information you need to solve a problem.

[19] Some people say that googling well is the most important skill in programming. This is an exaggeration, but it is important.

4.4 *Comparisons and conditionals*

In our programs we will want to do different things depending on the values of variables. For example, when numerically solving an equation, we will want to terminate once our approximation is good enough.

Let us first consider comparison operators. We can use the == operator (two equal signs) to test for equality[20]:

```
(python)  7 == 8
           False
(python)  8 == 8
           True
```

[20] Contrast this with the single =, which assigns values.

This operator returns a Boolean value (either **`True`** or **`False`**), of type `bool`. So do the other comparison operators:

```
(python)  7 < 8
           True
(python)  7 <= 8
           True
(python)  7 > 8
           False
(python)  7 != 8
           True
```

Here `!=` means *not equal to*.

We can also combine conditionals using **`and`** and **`or`**. Logical **`and`** takes in two Boolean values and only returns **`True`** if and only if both inputs are **`True`**. Logical **`or`** takes in two Boolean values and returns **`True`** if and only if any inputs are **`True`**. Here are some examples:

```
(python)  7 < 8 and False
           False
(python)  7 < 8 and 9 < 10
           True
(python)  7 < 8 or 9 < 10
           True
(python)  7 < 8 or False
           True
```

We can also check two-sided inequalities with a natural syntax in Python:

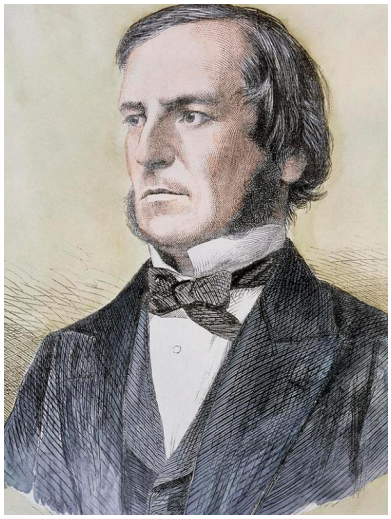
```
(python)  1 <= 5 <= 10
           True
(python)  1 <= 20 <= 10
           False
```

There is a logical **`not`** operator that returns the other Boolean value:

```
(python)  not True
           False
(python)  not False
           True
```

We can use these comparisons to do different things in our code depending on the values of variables using the **`if`** ...**`elif`** ...**`else`** construction, as in code block 4.1.

The **`if`** statement is followed by an expression that evaluates to **`True`** or **`False`**, followed by a colon (`:`). The code that should be ex-



George Boole, 1815–1864. Boole wrote his seminal work, *The Laws of Thought*, while professor of mathematics in Cork, Ireland. His book laid the foundation for the information age.

Code block 4.1. Example of basic conditional.

```
g = -9.81

if g < 0:
    print("Acceleration due to gravity is downwards")
    # some more calculations for this case ...
elif g == 0:
    print("No acceleration due to gravity. Are you in space?")
else:
    print("Acceleration due to gravity is upwards.")
```

ecuted if the conditional is `True` follows on the next lines. These lines **must** be indented (have spaces at the start); in Python, whitespace is syntactically significant. If you do not type the spaces on the lines, an `IndentationError` exception will be raised[21]. If you forget the colon, a `SyntaxError` will be raised.

After the code block following `if`, in this example we have an `elif` statement. This is short for "else if". If the `if` evaluated to True, this code is skipped over; otherwise, the expression for the `elif` is evaluated and if it is `True` *its* code block is executed[22]. You can follow an `if` statement with as many `elif` statements as you like, for as many cases as you need to consider (including no `elif` statements).

Finally, in this example we have an `else` statement. The code in this block gets executed if no other blocks were executed (i.e. the conditions for the `if` and any `elif` statements evaluated to `False`). As with the `elif` statements, the `else` statement is optional, but you can only have at most one of them.

Here is an example from Hill (2020)[23] (Example E2.21). In the Gregorian calendar a year is a leap year if it is divisible by 4 with the exceptions that years divisible by 100 are not leap years unless they are also divisible by 400. One might implement this logic as in code block 4.2.

In code block 4.2 you should read `if year % 400 == 0` as *if year is divisible by 400*, and so on. This example also demonstrates the use of *f-strings*, a convenient Python feature for formatting strings. Notice the `f` before the string in the print statement—this flags to Python that it is to scan the string for expressions in curly braces `{` and `}`, and to replace them with the evaluations of the expressions. In this case, `{year}` is replaced with the value of the variable inside the string before it is printed.

[21] The PEP8 style guide recommends that code blocks be indented with exactly 4 spaces—not 2 spaces, not tabs. However, you can break this if you wish, so long as you are consistent.

[22] This code block must again be indented with whitespace.

[23] C. Hill. *Learning scientific programming with Python*. Cambridge University Press, second edition, 2020

Code block 4.2. Gregorian leap year logic.

```
year = 1900

if year % 400 == 0:
    is_leap_year = True
elif year % 100 == 0:
    is_leap_year = False
elif year % 4 == 0:
    is_leap_year = True
else:
    is_leap_year = False

if is_leap_year:
    print(f"{year} is a leap year")
else:
    print(f"{year} is not a leap year")
```

Exercise 4.4. The Encyclopaedia Brittanica mentions a possible refinement of the Gregorian calendar[24]: that a year *not* be considered a leap year if it is divisible by 4,000[25]. Change the code above to implement this revised scheme.

[24] Pope Gregory XIII established this calendar in 1582, primarily because the Spring equinox was off from its intended target date of March 21 by 14 days, due to the mismatch between the 365.2422-day long solar year and the 365.25-day long year of the previous (Julian) calendar. With the Gregorian calendar, the year became 365.2425 days long. The Spring equinox was important because the date of Easter was a function of the Spring Equinox. Our word 'computation' comes from the Latin *computus*, which originally meant the calculation of the date of Easter.

[25] The jury is still out on whether this is a good idea, but we have just under two millennia to decide.

4.5 *Iteration*

It is frequently necessary in programming to have the computer repeat some operation for each item in some collection. In Python this is expressed with a `for` loop. The basic syntax is

```
for item in iterable_object:
```

Here is an example.

Code block 4.3. Example of basic iteration.

```
colours = ["red", "green", "blue"]
for colour in colours:
    print(colour)
```

Here we have an iterable object[26] `colours`, which is a *list* of strings. When the `for` loop is executed, the variable `colour` takes on each entry in the list in turn, and the body of the loop is executed for that value. As with conditionals, what code is in the body of the loop and

[26] An iterable object is one whose elements can be taken one at a time. Examples of built-in iterable objects include lists, tuples, dictionaries, and strings, which we will study in more depth in the next chapter.

what code is not is indicated by the indentation. Loops can be nested:

Code block 4.4. Example of nested iteration.

```
colours = ["red", "green", "blue"]
clothes = ["skirt", "jumper"]
for colour in colours:
    for article in clothes:
        print(f"I have a {colour} {article}")
```

In mathematics we often want to loop over integers. We can do this in Python with the `range` command. The code

Code block 4.5. Example of using `range`.

```
for i in range(5):
    print(i)
```

prints the integers 0, 1, 2, 3, and 4[27]. More generally, to express the sequence starting at a_0 and taking steps of size d (the stride), i.e. integers of the form

$$a_n = a_0 + nd, \quad n = 0, 1, \ldots \text{ until } a_n \geqslant \texttt{end},$$

one can use the command `range(a_0, end, d)`. For example, the code

[27] The argument to `range` is the first integer *not* returned in the iteration.

Code block 4.6. More using `range`.

```
for i in range(-10, 20, 5):
    print(i)
```

prints -10, -5, 0, 5, 10, 15.

The `range` command does not return a list—such a list would allocate memory for all entries and store them all at once. Rather, `range` returns a Python *iterable*, which is merely an object that knows how to compute the next entry to be returned. This is much more memory-efficient (imagine a list with a billion entries). This is why Python returns something somewhat opaque if you try to print a `range`:

```
(python)  range(5)
          range(0, 5)
```

In order to see at once what entries it contains, cast it to a list:

```
(python)  list(range(5))
          [0, 1, 2, 3, 4]
```

Let us now see an example. We will use a **for** loop to compute some entries of the Fibonacci sequence[28], defined by

$$F_{n+1} = F_n + F_{n-1}, \quad F_0 = 1, F_1 = 1.$$

[28] This naming is inaccurate, since the Fibonacci sequence was studied by Indian mathematicians such as Pingala some 1400 years before Fibonacci.

Code block 4.7. Calculating the first Fibonacci numbers.

```
a = 1
b = 1
print(a, b, end="")  # end="" means: do not print a newline
n = 200  # the number of entries to print
for i in range(2, n+1):  # we have already calculated 2, want to go up to n
    c = a + b  # calculate next Fibonacci number
    print("", c, end="")

    # Now we update the variables for the next iteration
    a = b
    b = c
```

Exercise 4.5. [Project Euler[29], problem 1] If we list all the natural numbers below 10 are multiples of 3 or 5, we get 3, 5, 6, and 9. The sum of these multiples is 23.

Find the sum of all multiples of 3 or 5 below 1000.

[29] Project Euler is a series of mathematical problems that computer programming skills to solve. Later ones also require substantial mathematical insight; playing the game is great fun. See https://projecteuler.net.

The other kind of loop in Python is a **while** loop. Whereas a **for** loop iterates over some existing iterable object, a **while** loop iterates until some condition is satisfied. Here is the simplest Python program that runs forever:

Code block 4.8. A program that runs forever.

```
while True:
    pass
```

The condition of the **while** loop will always be **True**, so the loop will never end. The **pass** command does nothing; it signifies an empty block of code where Python expects one.

Let us see how this works in practice. Euclid's algorithm for calculating the greatest common divisor of two numbers[30] repeatedly

[30] You will meet Euclid's algorithm in much more depth in Prelims *Constructive Mathematics* in Trinity term; our focus here is on what the code looks like.

calculates a sequence of remainders upon division, and terminates when the remainder reaches zero. The greatest common divisor is the last nonzero remainder in the sequence:

Code block 4.9. Euclid's algorithm for computing the greatest common divisor.

```
a = 3942
b = 486

# Loop until the remainder on division is zero
while b != 0:
    (a, b) = (b, a % b)

print(a)
```

Here we do not know how many iterations will be required in advance, so a **while** loop is more natural. The syntax

```
(a, b) = (b, a % b)
```

expresses *two* assignments at the same time; the corresponding components of the *tuples* on the left and right are assigned[31], with the entire expression on the right evaluated before any assignment. In other words, after the body of the loop, the new `b` is the remainder on division of the old `a` and b; the new `a` is the old `b`.

[31] We will discuss tuples more in the next chapter.

Exercise 4.6. Modify the code for Euclid's algorithm to count how many iterations of the **while** loop are executed. How many iterations are required to compute the greatest common divisor of 453973694165307953197296969697410619233825 and 2805711729925101400376119324130386771895 25?

Exercise 4.7. [Project Euler, problem 2] By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

It is sometimes useful to terminate a loop early. For example, we may wish to iterate over a list of integers, find the first entry that is even, and then stop iterating. We can do this with the **break** statement:

Code block 4.10. Breaking from a loop.

```
numbers = [3, 5, 7, 9, 12, 13, 14]
for number in numbers:
    if number % 2 == 0:
        break

print(f"The first even number is {number}.")
```

At each iteration we test whether the number is even or not, and if it is we exit the loop. This means that the variable `number` is assigned to the first even entry.

As written, the code is not quite correct. Consider the case where there are no even numbers in the list. The **`break`** statement will never be met; on exiting the loop, `number` will be assigned to the last value in the list. We can do the right thing here with the **`for`** ...**`else`** construct:

Code block 4.11. The `for-else` construct.

```
numbers = [3, 5, 7, 9, 13]
for number in numbers:
    if number % 2 == 0:
        break
else:
    print("There are no even numbers in the list!")
```

The code in the **`else`** block will be executed if no **`break`** statement was met. There is also an analogous **`while`** ...**`else`** construct.

The **`break`** statement only exits a *single* loop. If you want to break out of nested loops, one way is to create a flag variable. The following code looks for Pythagorean triples with integer values, and terminates when it finds one:

Code block 4.12. Finding a Pythagorean triple with integer values.

```
found_triple = False

for c in range(1, 100):
    for b in range(1, c):
        for a in range(1, b):
            if a**2 + b**2 == c**2:
                found_triple = True
                break

        if found_triple:
            break

    if found_triple:
        break

print(f"{a}**2 + {b}**2 == {c}**2")
```

Our final Python statement for control flow is the `continue` statement. Whereas `break` immediately stops the execution of the entire loop, `continue` merely stops the execution of the *current iteration* of the loop.

Code block 4.13. Example of using `continue`.

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

This prints 0, 1, 2, 4.

We close the chapter with a more advanced example of iteration with a `while` loop. Suppose we are seeking the root of a continuous function $f : \mathbb{R} \to \mathbb{R}$ and we have two values $a, b \in \mathbb{R}$ with function values of opposite sign, so that $f(a)f(b) < 0$. By the intermediate value theorem, there exists a root $r \in (a, b)$ such that $f(r) = 0$. The following algorithm is called *bisection*[32]. The algorithm evaluates the midpoint c of the interval (a, b), checks which sign $f(c)$ has, and chooses whichever half of the interval brackets the root (either (a, c) or (b, c)). We wish to repeat this process an unknown number of times, until our approximation of the root is such that $f(c) \approx 0$ (quantified by $|f(c)| < 10^{-10}$).

In the code, the `assert` statement evaluates a Boolean expression

[32] You will also meet the bisection algorithm for rootfinding again in Prelims *Constructive Mathematics* in Trinity term.

Code block 4.14. Finding the root of a real-valued function with bisection.

```
import math

# Set up: choose function and starting values
f = math.sin       # we will look for a root of sin
(a, b) = (-2, 1)   # starting guess for the root:
                   # it is somewhere in (-2, 1)

assert f(a) * f(b) < 0  # check that f(a) and f(b) have opposite sign

# Calculate first midpoint
c = (a + b)/2

iteration = 0  # iteration counter
print(f"{iteration}: c = {c}, f(c) = {f(c)}")
while True:
    if f(c) * f(a) < 0:
        # f(c) has the opposite sign to f(a),
        # so c replaces b
        (a, b) = (a, c)
    else:
        # f(c) must have the opposite sign to f(b),
        # so c replaces a
        (a, b) = (c, b)

    # Calculate midpoint for the next round of the loop
    c = (a + b)/2

    # Increment counter
    iteration = iteration + 1
    print(f"{iteration}: c = {c}, f(c) = {f(c)}")

    if abs(f(c)) < 1e-10:
        # Close enough, terminate with success
        break

    if iteration > 100:
        # Too many iterations, terminate with failure
        break
```

and raises an exception (an `AssertionError`) if it is `False`. Adding `assert` statements to your code to verify assumptions about your input is important for good programming.

Exercise 4.8. Adapt the bisection code example to compute an approximation to ten decimal digits of π.

5 Intermezzo: submitting problem sheets

You are now ready to work on the first problem sheet. Before you begin, a few words on how computational work should be submitted to the demonstrators. In order to make it easier for the demonstrator to give feedback on your work, we ask that you convert your `.py` file (which is good for a computer to execute) to a `.html` file containing the code *and its outputs* (which is good for a human to read). To do this, please follow these steps:

1. On the course webpage

   https://courses.maths.ox.ac.uk/course/view.php?id=4931

   there is a file, `publish.py`[1]. Download this file and place it in the same folder as your code for the problem sheets.

2. At the terminal, please type

   ```
   (terminal)  pip install ipykernel nbconvert jupytext
   ```

   which installs the dependencies of `publish.py`.

   Then, for each problem sheet, include the code

Code block 5.1. Publishing your work to a .html file

```
from publish import *
```

at the start, and

Code block 5.2. Publishing your work to a .html file

```
publish()
```

at the end. When you run your code with `python`, this command will save the code and its outputs in a `.html` file with the same name[2]. It is this `.html` file that you should submit to your demonstrator.

[1] We thank Lawrence Mitchell of NVIDIA for writing the first version of `publish.py` for this course.

[2] The path to the `.html` file is printed to the terminal.

The text in the comments you write will be formatted using *markdown* syntax. Markdown is a simple markup language for formatting plain text. Markdown is widely used in programming, for writing documentation[3], and for interacting with other programmers via bug reports and pull requests[4].

To get started, here is a template `.py` file for you to use to format your problem sheets.

[3] For example, Python packages are often documented with the `sphinx` package, which uses markdown.

[4] GitHub, the main platform used for open-source code development, formats bug reports and pull requests with markdown. If you want to know more about markdown, you can read GitHub's introduction to the subject.

Code block 5.3. Template for your problem sheets.

```
from publish import *

# # Computational Mathematics
# ## Problem sheet n
# ## 2023-09-19
# ## Firstname Surname (<my.email.address@college.ox.ac.uk>)
# ## Hogwarts College

#
# ***
# ### Question 1.
#

pass

#
# ***
# ### Question 2.
#

pass

#
# ***
# ### Question 3.
#

pass

#
# ***
#

publish()
```

You should of course change the problem sheet number, the date, your name, email address, and college. You will also need to replace the `pass` statements with the code that you write[5]! Running this code should make a HTML file that looks something like the image below.

[5] The empty lines and spaces matter in the formatting here, so please be careful.

```
from publish import *
```

Computational Mathematics

Problem sheet n

2023-09-19

Firstname Surname
(my.email.address@college.ox.ac.uk)

Hogwarts College

Question 1.

```
pass
```

Question 2.

```
pass
```

Question 3.

```
pass
```

---

6 Problem sheet 1

For your convenience, and for self-containment, we include the text of the problem sheets here.

1. Plato of Athens (c. 428 BC–348 BC) is a central figure in the history of philosophy. He founded in Athens the Academy, a school of philosophy and mathematics. A phrase reputedly above the door to the Academy, ''μηδες γεωμτρητος εστω'' (let no one ignorant of geometry enter), now adorns the entrance to our intellectual home, the Mathematical Institute.

   In the *Republic* (c. 375 BC), Plato enigmatically refers to an 'entire geometrical number'. The text is notoriously opaque, and scholars debate what exactly Plato means. The most popular opinion is that Plato's number is 216, since it is the first cube that is a sum of three cubes:

   $$216 = 6^3 = 3^3 + 4^3 + 5^3.$$

   The Pythagorean triple $(3, 4, 5)$ would of course have held immense geometrical significance to Plato (and to you, me, and everyone else).

   Plato has now requested your help. He is writing a sequel to the *Republic* (working title: *The Delian League Strikes Back*) and wishes to identify some more numbers to pontificate about. Write a program for Plato that identifies the first five cubes that can be written as the sum of three cubes.

2. Pythagoras (c. 570 BC–495 BC) founded a brotherhood in Crotona, in the south of Italy. One of the core tenets of the brotherhood was that ''All is number''; in other words, that the natural numbers (and their ratios) underpin all natural phenomena, in music, astronomy, philosophy and beyond.

   Hippasus of Metapontum (c. 530 BC–450 BC), a member of the brotherhood, has just discovered that $\sqrt{2}$ is in fact irrational; it cannot be expressed as the ratio of two natural numbers. He realises that this will be deeply shocking to his fellow Pythagoreans, since many of their proofs rely on the rationality of the lengths of line

segments. He now fears for his life. (Apocryphally, Hippasus was drowned at sea for revealing this discovery, although historians doubt any of this actually happened.)

Hippasus has now requested your help. He wishes to compute rational approximations of $\sqrt{2}$, to fool the Pythagoreans into thinking it is in fact rational, while he arranges his escape from Crotona. Write a program for Hippasus that calculates 10 different rational approximations of $\sqrt{2}$, using the `fractions` module of Python. Calculate the first 10 such approximations, in order of increasing denominator. You should create a

```
root2 = fractions.Fraction(2**(0.5))
```

on the floating-point representation of $\sqrt{2}$, and then use the

```
root2.limit_denominator(n)
```

method, which calculates the best rational approximation to the given number with denominator at most `n`.

[*Hint: It may be useful to consult the documentation for the* `fractions` *module.*]

[*Hint: you can check whether you have seen a given fraction by creating a set:* `history = set()`*. You can add a fraction you have seen to the set by* `history.add(fraction)`*. You can check whether a fraction is in the set with* `if fraction in history`*.*]

3. Heron of Alexandria (c. 10 AD–70 AD) is often considered the greatest experimenter of antiquity. He was a Greek mathematician and engineer who worked in Egypt under Roman rule. Among other achievements, he describes the first known steam engine, and the first wind-powered device on land.

   Heron invented the first fast algorithm for calculating the square root of a number s. Heron starts with an initial guess $x_0 \approx \sqrt{s}$ and then improves it by

   $$x_{n+1} = \frac{(x_n + s/x_n)}{2}.$$

   The idea here is that if x_n is an underestimate for $\sqrt{s}$, then s/x_n is an overestimate, and vice versa; averaging them yields an improved approximation.

   Heron has now requested your help. He needs to calculate $\sqrt{2}$ to 50 decimal digits, but is too busy building steam engines to do the calculations himself. Write a program for Heron to calculate $\sqrt{2}$ to 50 decimal digits.

[*Hint: without tricks, standard floating-point arithmetic will not be sufficient for this task, since it only carries* 15 *or* 16 *digits of precision. Instead, use the* `mpmath` *module. Install it with* `pip install mpmath` *at the terminal. Once it is imported, set the working precision to use* 100 *digits with* `mpmath.mp.dps = 100`*. Create your initial guess using* `x = mpmath.mpf('1.4')`*. You can then use the same arithmetical operators as in standard Python, but the calculations will be carried out with 100 digits of precision. Terminate your iteration once the difference between two successive approximations is less than* 10^{-50}.]

It now turns out that the Emperor has demanded an approximation of $\sqrt{2}$ to 200 digits. How many more iterations of Heron's method are required to calculate $\sqrt{2}$ to 100 digits, and then 200 digits? Write your answer in comments.

[*Hint: you can estimate how many digits you have correct by*

```
-int(mpmath.ceil(mpmath.log10(abs(current_guess - previous_guess))))
```

This is not foolproof, but is good enough for our purposes here.]

7 Data structures and plotting

7.1 Lists

Python offers powerful data structures that make it possible to implement complicated algorithms in a handful of lines of code. The first data structure we will meet is a *list*.

A list is an ordered array of objects. You can create a list using square brackets `[]`:

```
(python)  lista = ['one', 2, 3.0]
```

As you can see, a single list can contain references to different data types (in this case, strings, integers, and floats). A list can also contain other lists:

```
(python)  listb = [4j, lista]
(python)  listb
           [4j, ['one', 2, 3.0]]
```

An empty list can be created using `[]`.

You can access entries in a list by *indexing* it. In Python, indices start counting from zero. So to access the first entry in a list, you index it with `[0]`:

```
(python)  lista[0]
           'one'
(python)  lista[1]
           2
(python)  lista[2]
           3.0
(python)  listb[1][0]
           'one'
```

As you can see in the last example, you can index objects without

giving them names. If you attempt to use an index that is not defined, Python raises an `IndexError` exception[1].

It is often convenient to index a list backwards. The index `[-1]` refers to the last entry in a list, and so on:

```
(python)  lista[-1]
           3.0
(python)  lista[-2]
           2
```

Lists are *mutable*. Mutability is a key concept in Python. A mutable object is one where the object can be changed after the object has been created. For example, to change an entry in a list, you can do

```
(python)  lista[0] = 'five'

(python)  lista
           ['five', 2, 3.0]
(python)  listb
           [4j, ['five', 2, 3.0]]
```

Notice that changing `lista` has automatically changed what is printed by `listb`; lists contain *references* to objects, not copies. If you wanted to make a copy of a list, you could use the `list` function:

```
(python)  a = [1, 2, 3]

(python)  b = [4, a] # b refers to a

(python)  c = [4, list(a)] # c has a different copy of a

(python)  a[0] = 5

(python)  b
           [4, [5, 2, 3]]
(python)  c
           [4, [1, 2, 3]]
```

Here when constructing `c` we made a copy of the list `a`, which is why modifying `a` does not subsequently change `c`.

To test for containment, you can use **`in`**:

```
(python)  1 in [1, 2, 3, 4]
           True
(python)  1 in [0, 2, 3, 4]
           False
```

[1] This is in sharp contrast to languages like C and C++, which will happily let you access the 200th element of an array of length 100. This is a major cause of security vulnerabilities.

There is also the convenient **not in** operator:

```
(python)  1 not in [0, 2, 3, 4]
           True
```

which is more idiomatic and easier to read than `not (1 in [0, 2, 3, 4])`.

Lists can be *concatenated*, put together one after another. This can be done with the addition + symbol:

```
(python)  [100, 200] + [300, 400]
           [100, 200, 300, 400]
```

You can ask for the length of a list with `len`:

```
(python)  len([100, 200, 300, 400])
           4
```

You can make multiple copies of a list by multiplying it by an integer:

```
(python)  [0]*3
           [0, 0, 0]
```

Lists can be *sliced*. Slicing a list `l` with `l[i:j]` produces a sublist that starts at index `i` and goes up to (but does not include) index `j`:

```
(python)  l = [0, 1, 2, 3, 4, 5, 6]

(python)  l[1:4]
           [1, 2, 3]
(python)  l[3:6]
           [3, 4, 5]
```

You can also leave entries of the slice empty, to indicate no bound:

```
(python)  l[3:]
           [3, 4, 5, 6]
(python)  l[:-2]
           [0, 1, 2, 3, 4]
(python)  l[:] # returns the entire list
           [0, 1, 2, 3, 4, 5, 6]
```

The reason why the upper bound is not included is to preserve the identity `l[:i] + l[i:] == l`.

When slicing, you can also specify an optional *stride*. This is useful if you want every second or third element in a list, for example:

```
(python)  l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
(python)  l[0:10:2] # take every second entry
           [0, 2, 4, 6, 8]
(python)  l[0:10:3] # take every third entry
           [0, 3, 6, 9]
(python)  l[::3] # take every third entry, without bounds
           [0, 3, 6, 9]
```

You can reverse the sublist by taking a negative stride:

```
(python)  l[::-1]
           [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
(python)  l[-2::-2]
           [8, 6, 4, 2, 0]
```

Finally, lists have many useful methods. `list.append` appends an element to a list:

```
(python)  l = [0, 1, 2]
(python)  l.append(3)
(python)  l
           [0, 1, 2, 3]
```

`list.remove` removes the first occurrence of an element in a list:

```
(python)  l.remove(1)
(python)  l
           [0, 2, 3]
```

`list.index` finds the first index at which a given element occurs:

```
(python)  [4, 6, 3, 4, 5, 6].index(4)
           0
(python)  [4, 6, 3, 4, 5, 6].index(6)
           1
```

`list.reverse` reverses a list in-place (i.e. it modifies the existing list, rather than returning a new list). `list.sort` sorts a list in-place.

7.1.1 *List comprehensions*

In mathematics, we often iterate over an object to make another one. For example, the notation $2\mathbb{Z}$ means to construct a set by *comprehension*:

$$2\mathbb{Z} = \{2z : z \in \mathbb{Z}\}.$$

In other words, we iterate over the set $\mathbb{Z}$ and do something to its elements to produce a new set. This sometimes involves a predicate, a condition to check on the elements of the original set. For example, the rational numbers are defined by

$$\mathbb{Q} = \{p/q : p, q \in \mathbb{Z} \text{ if } q \neq 0\}.$$

Here the predicate is that $q \neq 0$.

Python has a very convenient and powerful notation for this concept, *list comprehension*:

```
(python)  [2*x for x in [0, 1, 2, 3, 4]]
           [0, 2, 4, 6, 8]
(python)  [2*x for x in [0, 1, 2, 3, 4] if x % 2 == 0]
           [0, 4, 8]
```

We loop over a given iterable (in this case `[0, 1, 2, 3, 4]`), applying an operation to each element (in this case, doubling it). One can also supply a predicate by adding `if` in the statement, in this case checking if the element of the original list is even.

We can use list comprehension to sum the entries of two lists:

```
(python)  a = [100, 200, 300]
(python)  b = [1000, 2000, 3000]
(python)  [a[i] + b[i] for i in range(len(a))]
           [1100, 2200, 3300]
```

Let us use this in an example. Pascal's triangle[2] arranges the coefficients of the binomial expansion in a triangle. The first row $n = 0$ starts with a single entry 1. Each entry $k = 0 \ldots n$ on each subsequent row n is calculated by adding its two diagonal parents (with implicit zeros off the triangle).

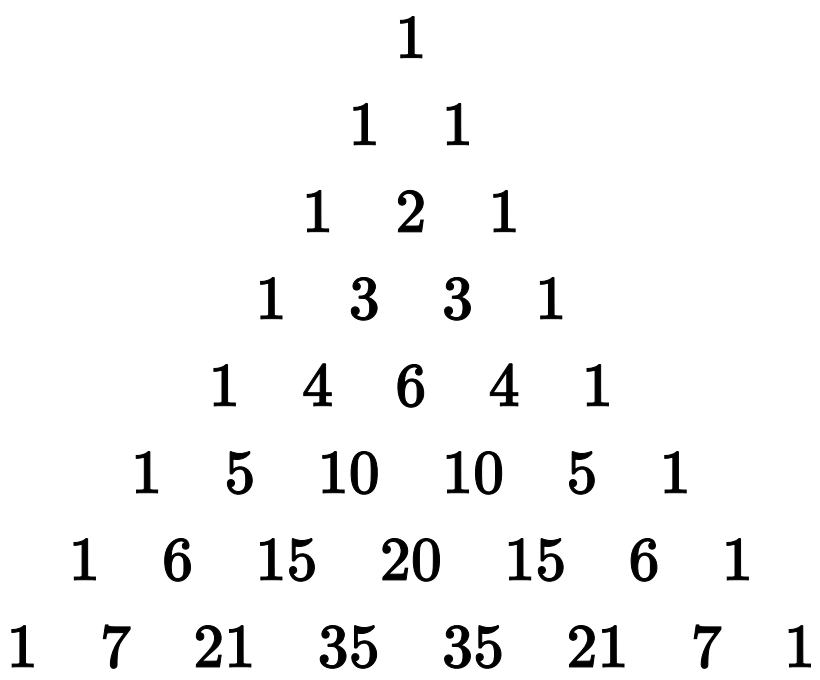
We can calculate rows of Pascal's triangle conveniently with lists, as in code block 7.1.

The key line of this code

```
row = [([0] + row)[i] + (row + [0])[i] for i in range(n+1)]
```

can be understood as follows. Let us take the case $n = 2$ where currently `row = [1, 1]` and we wish to compute `[1, 2, 1]`. The expression `[0] + row` evaluates to `[0, 1, 1]`. The expression `row + [0]` evaluates to `[1, 1, 0]`. The body of the list comprehension then sums these two lists, as before, yielding `[1, 2, 1]`.

[2] Again, the naming is inaccurate, since this pattern was known before Pascal to (among others) Persian mathematicians Al-Karaji and Omar Khayyám, Chinese mathematicians Jia Xian and Yang Hui, and European mathematicians Jordanus de Nemore, Gersonides, and Niccolò Fontana Tartaglia.



Code block 7.1. Pascal's triangle.

```
N = 10  # number of rows to compute
row = [1]  # starting row
rows = []  # this list will contain all rows computed

for n in range(N):
    # Calculate the next row from the previous one
    row = [([0] + row)[i] + (row + [0])[i] for i in range(n+1)]
    rows.append(row)

# Print the output
for row in rows:
    print(row)
```

Exercise 7.1. In our code, we have decided to represent polynomials (i) in the monomial basis (as linear combinations of 1, x, x^2, x^3, etc.)[3] (ii) using Python lists. For example, the polynomial $1 + 3x + 8x^3$ would be represented as `[1, 3, 0, 8]`.

Given a representation of a polynomial in this manner, write a **`for`** loop to generate the representation of its derivative.

[3] The monomial basis is the one you are familiar with from school, and it is fine for calculations on paper or in symbolic algebra packages. But the monomial basis is usually a very poor choice for numerical computation; in a sense one can make precise, the basis functions all get "too similar" to each other, with the consequence that small changes in coefficient values produce enormous changes in the actual polynomial. Instead, stable families of basis functions like Chebyshev or Legendre polynomials underpin most computations with polynomials.

Exercise 7.2. [Hill (2020), Q2.4.4] The Python function `sum` sums all elements of an iterable object. Use `sum` and list comprehension to write one line of Python that computes π using the first 20 entries of the Madhava[4] series:

$$\pi = \sqrt{12}\left(1 - \frac{1}{3 \times 3^1} + \frac{1}{5 \times 3^2} - \frac{1}{7 \times 3^3} + \cdots\right).$$

How many digits of this approximation are correct?

[4] Madhava of Sangamagrama (c. 1350–1425) was a mathematician and astronomer in Kerala, India. He made many discoveries on infinite series expansions for trigonometric functions like sin, and used them to calculate a table of sines that was accurate to 7 digits. He discovered what is often referred to as the Leibniz series for π nearly 300 years before Leibniz, and hence it is often now referred to as the Madhava–Leibniz series. His work also anticipated many of the foundations of calculus subsequently developed by Newton and Leibniz. See `https://en.wikipedia.org/wiki/Madhava_of_Sangamagrama` for more.

7.2 *What assignment means in Python*

Now that we have met a mutable object, we can study more deeply what assignment means in Python. Variable assignment using = really means *assigning labels*.

Consider the following code.

```
(python)  lista = [1, 2, 3]

(python)  listb = lista

(python)  lista.append(4)

(python)  listb
           [1, 2, 3, 4]
```

After the second line, both `lista` and `listb` are *labels* for the same object in memory. Modifying the (mutable) object in place on the third line via `lista` also changes the memory accessed via `listb`, since they are both labels for the same thing. We might visualise this as in figure 7.1.

We can test whether two variable names (two labels) both refer to the same object with **`is`**:

```
(python)  lista is listb
           True
```

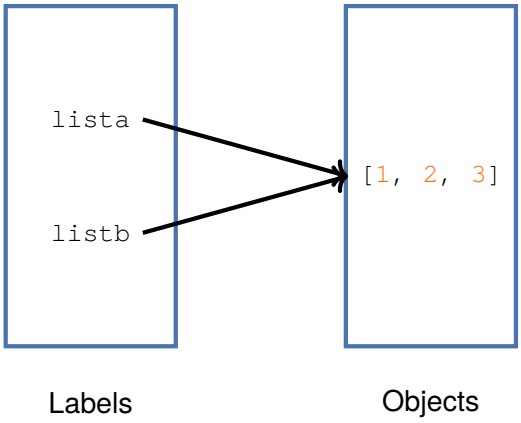


Figure 7.1: In Python, assigning a variable creates a label for an object.

By contrast, if make the list twice, changing `lista` does *not* change `listb`:

```
(python)  lista = [1, 2, 3]
(python)  listb = [1, 2, 3]
(python)  lista == listb
           True
(python)  lista is listb
           False
(python)  lista.append(4)
(python)  listb
           [1, 2, 3]
```

After the second line, the two objects are *equal*, but they are not *identical*. Two different lists have been constructed that happen to have the same elements. This situation might be visualised as in figure 7.2.

7.3 *Tuples*

Let us resume our study of the data structures Python offers. The next data structure we will meet is a *tuple*. A tuple is very much like a list, but immutable; once constructed, it cannot be changed. Whereas lists are constructed using square brackets `[]`, tuples are constructed using round brackets, `()`:

```
(python)  t = ('one', 2, 3.0)
```

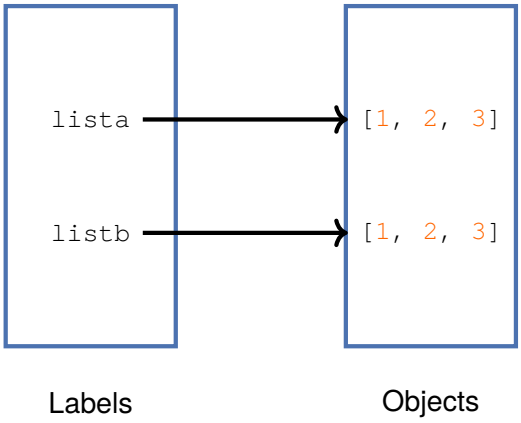


Figure 7.2: Two objects can be *equal* but not *identical*.

```
(python)  t[0]
           'one'
(python)  t[0] = 'four'
           ---------------------------------------------------
           TypeError Traceback (most recent call last)
           <ipython-input-2-b8ea296315bb> in <module>
           ----> 1 t[0] = 'four'
           TypeError: 'tuple' object does not support item assignment
```

Tuples are often used in Python to assign multiple variables. The syntax

```
(python)  (a, b, c) = (1, 2, 3)
```

is preferred[5] to

```
(python)  a = 1; b = 2; c = 3
```

[5] Many people say the code is *more Pythonic*.

Tuples also arise is in passing arguments to functions, and returning outputs from functions. We will discuss this further in section 7.6.

With our understanding of tuples, we can now discuss two useful Python functions, `enumerate` and `zip`.

Imagine we have a list of names of planets in the solar system, and wish to print out that that the first planet from the sun is Mercury, and so on. A naïve way to code this is shown in code block 7.2. This pattern (of needing to enumerate the elements of an iterable object) is so common that Python offers the `enumerate` function. `enumerate` takes in an iterable object and returns another iterable object, where each entry is a tuple `(num, obj)` where `obj` is from the input iterable. An example should make this clearer:

Code block 7.2. Listing the planets by number.

```
planets = ['Mercury',
           'Venus',
           'Earth',
           'Mars',
           'Jupiter',
           'Saturn',
           'Uranus',
           'Neptune']

counter = 0
for planet in planets:
    counter = counter + 1
    print(f"Planet {counter} from the Sun is {planet}.")
```

(python)
```
list(enumerate(planets))
```

```
[(0, 'Mercury'),
 (1, 'Venus'),
 (2, 'Earth'),
 (3, 'Mars'),
 (4, 'Jupiter'),
 (5, 'Saturn'),
 (6, 'Uranus'),
 (7, 'Neptune')]
```

With this, we can give a much more Pythonic version of code block 7.2 in code block 7.3.

Now suppose that our CEO decides that the program should print out 'The first planet from the Sun is Mercury' instead. How can we do this? The Python function `zip` lets you iterate over two or more iterables at the same time. It creates an iterable where each entry is a tuple of the entries of the input iterables:

(python)
```
list(zip([1, 2, 3], ('a', 'b', 'c')))
 [(1, 'a'), (2, 'b'), (3, 'c')]
```

We can use this to satisfy our CEO as in code block 7.4.

Code block 7.3. A better version of listing the planets by number.

```
planets = ['Mercury',
           'Venus',
           'Earth',
           'Mars',
           'Jupiter',
           'Saturn',
           'Uranus',
           'Neptune']

for (counter, planet) in enumerate(planets):
    print(f"Planet {counter} from the Sun is {planet}.")
```

Code block 7.4. Zipping two lists.

```
planets = ['Mercury',
           'Venus',
           'Earth',
           'Mars',
           'Jupiter',
           'Saturn',
           'Uranus',
           'Neptune']

ordinals = ['first',
            'second',
            'third',
            'fourth',
            'fifth',
            'sixth',
            'seventh',
            'eigth']

for (ordinal, planet) in zip(ordinals, planets):
    print(f"The {ordinal} planet from the Sun is {planet}.")
```

7.4 *Dictionaries*

Dictionaries record associations between *keys* and *values*[6]. Dictionaries are created using curly braces `{}`, with the key-value pairs separated

[6] In other languages, these are sometimes called associative arrays or hash tables.

by colons `:`. For example, a phone book might be implemented with a dictionary:

```
(python) phonebook = {"Arya": 3245, "Bran": 9282, "Cate": 6251}
(python) phonebook["Arya"]
          3245
(python) phonebook["Cate"]
          6251
```

We can look up dictionaries using the indexing notation familiar from lists and tuples, but now the keys need not be integers. The keys of a dictionary must be *hashable*[7], but the values may be mutable; the dictionary overall is a mutable object. Key-value pairs can be added, modified, and removed after creation:

[7] *Hashing* is the process of computing a signature of some complicated data structure, usually as an integer. It is these integers that a dictionary uses internally to store and index items. While being hashable and mutable are distinct concepts (an object can be any combination of the two or their negations), for the data structures built in to Python, hashability and immutability are the same. In practice, this means e.g. you can use tuples as keys of a dictionary, but not lists.

```
(python) phonebook["Dany"] = 1862 # add a new entry
(python) phonebook["Cate"] = 7827 # modify existing entry
(python) del phonebook["Bran"] # delete existing entry
```

Here **`del`** is short for delete.

The key-value pairs do not all need to be the same type. If you try to access a key that is not defined in the dictionary, a `KeyError` exception is raised.

Python tries to keep the interface among its data structures consistent. As with lists, you can check for key containment with **`in`**:

```
(python) "Dany" in phonebook
          True
(python) "Eddard" in phonebook
          False
```

Similarly, you can get the number of keys with `len`:

```
(python) len(phonebook)
          3
```

Iterating over a dictionary returns the keys, as in code block 7.5.

As with lists, there are lots of useful methods on dictionaries, including `.items()` (which lets you iterate over (key, value) pairs) and `.get(key)`, which allows you to specify a default value to use if the key is not found. Another point of similarity is that you can conveniently construct dictionaries using *dictionary comprehension*. In

Code block 7.5. Iterating over a dictionary.

```
phonebook = {"Arya": 3245, "Bran": 9282, "Cate": 6251}
for name in phonebook:
    print(f"{name} --> {phonebook[name]}")
```

this code, we use a dictionary as a simple database[8], recording the area and population of the ten largest islands in the world. From this dictionary, we derive another dictionary that stores the population density in code block 7.6. Here the key-value pairs are separated by a

[8] Of course, a *real* database would offer much more functionality, like supporting queries, redundancy and backups, access control, etc.

Code block 7.6. Dictionary comprehension.

```
# Data is (area in square kilometres, population)
islands = {'Greenland': (2130800, 56653),
           'New Guinea': (785753, 11.31e6),
           'Borneo': (743330, 21.26e6),
           'Madagascar': (587041, 22.01e6),
           'Baffin': (507041, 10745),
           'Sumatra': (473481, 50.18e6),
           'Honshu': (225800, 104e6),
           'Victoria': (217291, 1875),
           'Britain': (209331, 60.8e6),
           'Ellesmere': (196235, 191)}

# Build a dictionary containing population density using comprehension
density = {island: pop/area for (island, (area, pop)) in islands.items()}
```

colon `:`, and we use two levels of tuple unpacking to compactly assign the variables. Printing `density` yields

```
{'Greenland': 0.02658766660409236,
 'New Guinea': 14.393836230978437,
 'Borneo': 28.60102511670456,
 'Madagascar': 37.49312228617762,
 'Baffin': 0.021191580168073192,
 'Sumatra': 105.98102141374206,
 'Honshu': 460.58458813108945,
 'Victoria': 0.008628981412023508,
 'Britain': 290.4490973625502,
 'Ellesmere': 0.0009733228017428084}
```

Exercise 7.3. Write a one-line dictionary comprehension to swap the keys and values of a given dictionary.

Exercise 7.4. Exercise 7.1 discussed how one might represent the coefficients of a polynomial in the monomial basis with a list. But what if the coefficients are *sparse* (mostly zeros)? For example, to represent the coefficients of $1 + x^{1000}$ would require storing 999 zeros, which seems silly.

Let us instead represent polynomials with a dictionary. For example, $1 + 3x + 8x^3$ would be represented as `{0: 1, 1: 3, 3: 8}`.

Given a representation of a polynomial in this manner, build the representation of its derivative.

Harder, optional: given representations of two polynomials in this manner, build the representation of their product.

7.5 Sets

In Python, a *set* is an unordered collection of unique items. As with dictionaries, their entries must be hashable. We can build a set using curly braces `{}`, this time without colons:

```
(python)  s = {1, 1, 2, 2.0, 3, 4j}
(python)  s
           1, 2, 3, 4j
```

Here the duplicate entry of `1` is ignored, but why does `2.0` not appear? The reason is that both `2` and `2.0` hash to `2`; when hashes collide, Python determines uniqueness by equality, and in Python `2 == 2.0`:

```
(python)  hash(2)
           2
(python)  hash(2.0)
           2
(python)  2 == 2.0
           True
```

As with dictionaries and lists, you can check for containment with **`in`**:

```
(python)  1 in s
           True
```

```
(python)  4 in s
           False
```

You can get its cardinality with `len`:

```
(python)  len(s)
           4
```

Python sets model sets in mathematics, and support the same key operations: taking unions, intersections, testing for subset relations, etc. Here are some examples:

```
(python)  seta = {1, 2, 3}
(python)  setb = {3, 4, 5}
(python)  setc = {1, 2}
(python)  setc < seta # subset containment
           True
(python)  setc <= seta # subset containment or equality
           True
(python)  seta | setb # union
           1, 2, 3, 4, 5
(python)  seta & setb # intersection
           3
(python)  seta - setb # set difference
           1, 2
(python)  seta ^ setb # symmetric difference, elements in either but not both
           1, 2, 4, 5
(python)  setb.isdisjoint(setc)
           True
```

Sets are mutable; you can add and remove entries after creation:

```
(python)  s = {1, 2, 3}
(python)  s.add(4)
(python)  4 in s
           True
(python)  s.remove(3)
(python)  3 in s
           False
```

As such, you cannot use sets as keys for dictionaries. However, `set`

has an immutable friend, a `frozenset`. This is like a set in every way except that it is immutable, and so they can be used as keys in a dictionary.

As with lists and dictionaries, you can construct sets using *set comprehension*:

```
(python)  {x**2 for x in range(5)}
           0, 1, 4, 9, 16
```

Exercise 7.5. In 1774, Euler made a remarkable discovery, that the expression

$$n^2 + n + 41, \quad n \in [0, 40) \cap \mathbb{N}$$

produces 40 distinct prime numbers. Use a one-line set comprehension to compute these primes.

7.6 *Functions*

It is common in programming to have a set of statements that you would like to re-use again and again. For example, if we have some code to calculate whether an integer is prime or not, we might want to apply that to different inputs, or to use it in other programs. If someone comes along with a faster primality test, we should be able to swap out the old one from our code easily.

In Python, we use *functions* to gather together a specified set of statements so that we can code, test, and use them repeatedly. Functions enable code reuse—for example, if you find yourself copying and pasting code from within your program, you should probably make a function out of it. Functions also make programming more tractable for our limited human minds; functions allow us to break down a big problem into a set of smaller problems that we can think about tackling in isolation, before putting them all together.

In Python, we define a function with the **`def`** statement. A function takes in some inputs, executes some statements, and possibly returns a value with the **`return`** statement. Our first example is code block 7.7. This defines a new variable, called `square`, of type `function`. The function we have defined takes in a single input[9], which inside the function is called `x`[10], does some calculations, and **`return`**s an output. In the calling code, we pass the inputs to the function in round brackets `()`, just as when we call a built-in function in Python. The function carries with it its documentation: users can access the documentation with `help(square)`[11].

[9] As in mathematics, the inputs to a function are often called its *arguments*.

[10] What variable names are used in the code that calls this function are irrelevant; within the *scope* of the function, the input is called `x`.

[11] This inline documentation is called the function's *docstring*, in Python parlance.

Code block 7.7. Defining a function to square its input.

```
def square(x):
    """
    Return the square of the input.
    """
    x_squared = x*x
    return x_squared

y = 3
y_squared = square(y)
print(f"{y} squared is {y_squared}")
```

The problem sheets frequently have instructions along the lines of "write a function that takes as input x and returns y". This means that the function should use the **`return`** statement to pass the computed output y to the calling code.

Functions can take multiple inputs. They always return a single object, but that single object may be a tuple containing multiple outputs. As an example, in code block 7.8 is a function that takes in the coefficients of a quadratic polynomial in the monomial basis and returns its roots.

There are several things to note about code block 7.8. First, the docstring describes the function's purpose, inputs, outputs, and gives the big picture of how the function works; your docstrings should do the same. This is also the first time we see how to raise an exception, with the `raise` statement. If user code calls this with `a == 0`, it is likely a mistake, and it will be obvious to the programmer. We use an `assert`ion to check that it is giving sensible answers for the solutions of $x^2 - 1 = 0$[12]. Lastly, we see on the final line that it is possible to specify arguments out of order when calling a function, so long as we tell Python which is which.

It is sometimes useful to specify *default values* for the inputs of a function. Consider again the representation of polynomials using a list of coefficients in the monomial basis of exercise 7.1. when we write a function to calculate its derivative, it would be convenient for users to be able to specify how many derivatives to take, defaulting to one. Such a function might look as in code block 7.9.

Again, there are several things to note in code block 7.9. The first is that we specify `order=1` when defining the function. If the user does not specify `order`, it defaults to one. The second is that we check whether the `order` argument is an integer using the Python func-

[12] Even better would be to code a *test* for this function, verifying automatically that it gives the correct output on a wide range of inputs. This would give us confidence as we develop the code that all previous functionality remains correct. `py.test` is the de facto standard testing framework for Python, but its use is outside the scope of this course.

Code block 7.8. Defining a function for the roots of a quadratic polynomial

```
import math

def roots(a, b, c):
    """
    Return the two roots of the quadratic polynomial

    a*x**2 + b*x + c

    using the quadratic formula.

    Raises a ValueError if a == 0.
    """

    if a == 0:
        raise ValueError("Not a quadratic polynomial!")

    det = math.sqrt(b**2 - 4*a*c)
    root1 = (-b - det) / (2*a)
    root2 = (-b + det) / (2*a)

    return (root1, root2)

assert roots(1, 0, -1) == (-1, 1)
assert roots(c=-1, b=0, a=1) == (-1, 1)
```

Code block 7.9. Optional arguments and recursion.

```
def diff_poly(p, order=1):
    """
    Differentiate a polynomial p an arbitrary number of times,
    where the polynomial is expressed as a list of coefficients
    in the monomial basis.

    order is the order of the derivative to calculate.
    """

    if order < 0 or not isinstance(order, int):
        raise ValueError("Only for positive integer derivative orders")

    if order == 0:
        # No derivatives to calculate, just return the polynomial
        return p

    # Check for constants, doesn't matter what order is requested
    if len(p) == 1:
        return [0]

    # Calculate first derivative
    dp = [(i+1)*a for (i, a) in enumerate(p[1:])]

    if order == 1:
        return dp
    else:
        return diff_poly(dp, order=order-1)

print(diff_poly([0, 0, 0, 1]))  # order defaults to 1
print(diff_poly([0, 0, 0, 1], order=2))
print(diff_poly([0, 0, 0, 1], order=3))
print(diff_poly([0, 0, 0, 1], order=4))
```

tion `isinstance`. Third, we calculate the first derivative in one line of clear, readable code, using list comprehension, slicing, and the `enumerate` function. The fourth is that *the function calls itself*: if we want to calculate the second derivative, the function calculates the first derivative, then calls `diff_poly` again to calculate the derivative of that. Such a function is called *recursive*.

Let us now see the most general pattern for taking arguments in Python. We have seen that we can pass arguments unlabelled (as with `roots(1, 0, -1)`, or labelled (as with `roots(c=-1, b=0, a=1)`). When defining a function, we can give it an argument `*args` that gathers *all* unlabelled arguments not otherwise assigned in a tuple, and an argument `**kwargs` that gathers *all* labelled arguments not otherwise assigned in a dictionary. This is best seen with an example, code block 7.10.

Code block 7.10. The most general pattern of arguments in Python.

```
def printer(*args, **kwargs):
    """
    Prints any arguments passed to the function, labelled or unlabelled.
    """

    print(f"Unlabelled arguments: {args}")
    print(f"Labelled arguments: {kwargs}")


printer(1, 2.0, 3j, four=4, five=5j)
```

This code prints

```
Unlabelled arguments: (1, 2.0, 3j)
Labelled arguments: {'four': 4, 'five': 5j}
```

Try passing different arguments to this function to fully understand how this works.

The `printer` function we have defined lacks a **`return`** statement. What does Python do if we try to assign its output? In Python, *every* function **`return`**s a value—if the function has no **`return`** statement, then the function implicitly returns a special value **`None`**. **`None`** indicates no value at all, the absence of data. **`None`** is different to every other built-in constant—it is neither **`True`** nor **`False`**.

Let us close with a useful example, of testing whether an integer is prime or not.

Code block 7.11. First attempt at primality testing.

```
import math

def isprime(n):
    """
    Check whether an integer n is prime or not.

    Returns True or False.

    Raises ValueError if n is not an integer.
    """

    if not isinstance(n, int):
        raise ValueError("Only integers can be prime")

    if n < 2:
        return False

    for m in range(2, math.isqrt(n)+1):
        if n % m == 0:
            return False

    return True


for i in range(-1, 21):
    print(f"Is {i} prime? {isprime(i)}")
```

After the preliminaries, this function works by checking all possible divisors of n, up to $\sqrt{n}$ (since any factor greater than $\sqrt{n}$ will have a matching quotient less than $\sqrt{n}$). Here `math.isqrt` returns the integer part of the square root of its argument. If any divisor divides cleanly, the function is not prime; otherwise, the function is prime.

Exercise 7.6. In the code above, we are testing with all potential divisors between 2 and $\sqrt{n}$. This is obviously wasteful, since there is no point testing with 4 if n is not divisible by 2; if n were divisible by 4 it would also be divisible by 2.

Modify the code to only test with odd divisors, maintaining correctness. How much faster does this make `isprime(9999991111111)`?

More generally, the optimal set of divisors to try would be all prime

Primality testing is a fascinating subject at the interface of computational mathematics and number theory. Many of the fastest algorithms available are in fact *probabilistic*, with a provably small probability of failure. The fastest deterministic algorithms are based on deep insights from the theory of elliptic curves. For more details, see `https://mathworld.wolfram.com/PrimalityTest.html`.

numbers up to $\sqrt{n}$. Calculating this is more work than just testing for primality[13], but we can use this insight to improve the code further. By exploiting the fact that all primes greater than 3 are of the form $6n \pm 1, n \in \mathbb{N}$[14], further reduce the number of divisors tested. How much faster again does this make `isprime(99999911111111)`[15]?

[13] Production codes cache a large number of primes, say all the ones with up to a certain number of digits.

[14] If you have not met this fact before, can you prove it?

[15] Algorithm improvements often result from, and motivate, mathematical insight. The best computational mathematics motivates theory with computation, and improves computation with theory.

7.7 *Plotting*

Being able to *visualise* mathematical ideas is invaluable for research and understanding. Computers aid this process tremendously.

In Python, the main library used for plotting 2D graphics is called `matplotlib`[16]. To install this, at the terminal type

```
(terminal)  pip install numpy matplotlib
```

This also installs numpy[17], a library for array computing in Python that we will meet more closely in Chapter 12.

[16] J. D. Hunter. Matplotlib: a 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007

[17] C. R. Harris et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020

Here is a first demonstration. We will plot the function

$$f(x) = \sin x + \sin x^2, \quad x \in [0, 15]$$

in code block 7.12.

Code block 7.12. A first demonstration of plotting.

```
import matplotlib.pyplot as plt
from numpy import linspace, sin

def f(x):
    return sin(x) + sin(x**2)


# Sample the function on an equispaced
# grid of 2000 points on [0, 15]
xs = linspace(0, 15, 2001)
ys = f(xs)  # vectorised; does all points at once

plt.plot(xs, ys)
plt.show()
```

Running this code should bring up a window showing a plot like that in figure 7.3.

In code block 7.12, we first import `matplotlib`'s `pyplot` interface. This interface is designed to closely mimic the extremely successful plotting interface used in MATLAB[18]. We import the `sin` and

[18] There is another (object-oriented) interface for matplotlib, but we will not use it on this course. You might see it in demos online, however.

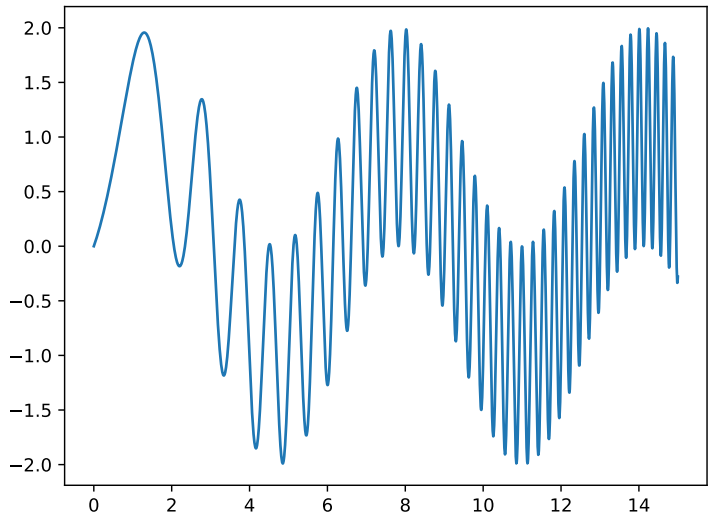


Figure 7.3: The figure displayed by code block 7.12.

`linspace` functions from `numpy`. `linspace` makes a numpy array[19] of equispaced points:

```
(python)  linspace(0, 1, 5)
           array([0. , 0.25, 0.5 , 0.75, 1. ])
(python)  linspace(0, 1, 11)
           array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

We next define the function that we wish to evaluate. Interestingly, we can apply this function to the entire array of x values with one line, `ys = f(xs)`. This is because numpy's `sin` function is *vectorised*: it can evaluate the same operation on many data simultaneously. Exploiting this is much more efficient than the equivalent `ys = [f(x) for x in xs]`[20]. This is why we **`import`**ed `sin` from `numpy` and not from `math`; the built-in `math.sin` does not vectorise.

We next plot the data with `plt.plot(xs, ys)`. This takes in the x-coordinates of the points to plot in one iterable, the y-coordinates in another iterable, and by default draws solid blue straight lines between them. The function `plt.show()` then renders the figure and opens an interactive window displaying it.

Of course, we have committed a cardinal sin: we have not labelled our axes, or given our plot a title! Let us do better in code block 7.13.

[19] For our purposes now, we can treat numpy arrays like lists. We will discuss the differences in chapter 12.

[20] For example, the computer on which this manual is written on has a CPU that supports AVX-512. These instructions can evaluate 8 double-precision floating-point operations simultaneously. Its GPU can compute 6144 single-precision floating-point arithmetic operations simultaneously. In general, when trying to compute at speed, one of the main challenges is *trying to get the data to the processors fast enough*; vectorisation is very useful for this.

Code block 7.13. Labelling axes and title.

```
import matplotlib.pyplot as plt
from numpy import linspace, sin

def f(x):
    return sin(x) + sin(x**2)


# Sample the function on an equispaced
# grid of 2000 points on [0, 15]
xs = linspace(0, 15, 2001)
ys = f(xs)

plt.grid()
plt.plot(xs, ys, '--og', linewidth=0.5, markersize=0.5)
plt.xlabel(r"$x$")
plt.ylabel(r"$f(x)$")
plt.title(r"Plot of $f(x) = \sin{x} + \sin{x^2}$")
plt.show()
```

There are several new elements in this code. First, we use `plt.grid()` to activate a background grid to make the plot easier to read. When we plot our data, we pass the format string `'--og'`. The `'--'` means to draw dashed lines, instead of solid ones. The `'o'` means to plot a circle at every data point. The `'g'` means to colour these in green[21]. We control the line width and marker size using optional arguments to `plt.plot`. We label the x and y axes using `plt.xlabel` and `plt.ylabel`; to these we pass strings with mathematics typeset in TeX notation[22], with the mathematics between dollar signs $ ... $. We specify that these strings are 'raw' strings using the r in front of the string `r""`; this makes the Python interpreter pass the backslashes through, rather than trying to interpret them as special characters. This should show a plot like that of figure 7.4.

[21] See `https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html` for the full specification of format strings.

[22] TeX notation is a means of typesetting equations. Its study beyond the scope of this course; you will learn it for your Part C dissertations, or probably before.

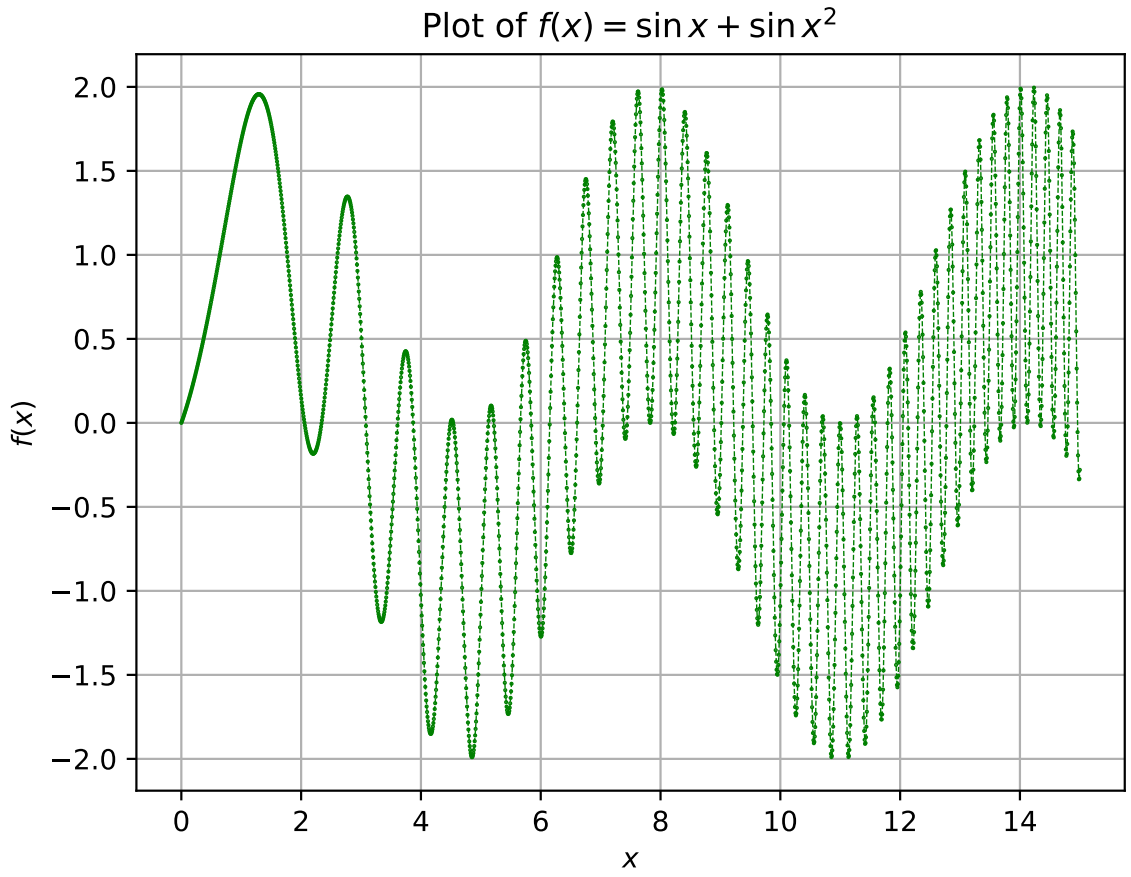


Figure 7.4: The figure displayed by code block 7.13.

8 Intermezzo: the Lander–Parkin counterexample

The material in this chapter is not necessary for the problem sheets, or for the course. It is here strictly for your interest.

Recall the Lander–Parkin counterexample to Euler's Conjecture[1]:

$$27^5 + 84^5 + 110^5 + 133^5 = 144^5.$$

How could we find this, if we did not already know it?

Here is a first attempt in code block 8.1. It searches all combinations of natural numbers $(a_0, a_1, a_2, a_3, b) \in ([1, \mathtt{N}) \cap \mathbb{N})^5$, up to some specified upper limit `N`.

Of course, this code does not actually *find* a counterexample, because `N` is too small, but this will come once the code has been improved. Here we import the `time` module (another Python built-in) to record the start and end times of the loop. We also include a counter, `tests`, which records how many times we evaluate the test for equality. The syntax `{end - start:.2f}` means to print the time difference to two digits after the decimal place[2]. Running this on a laptop yields

```
Total time: 2.67 s
Total number of tests: 3200000
```

which is not surprising as $20^5 = 3200000$. If we multiplied `N` by 10 (so as to actually find the counterexample), the time taken would multiply by about 10^5, and the code would take roughly 3 days. We need to make this go faster.

The first improvement we can make is to not search *every* combination $(a_0, a_1, a_2, a_3, b) \in ([1, \mathtt{N}) \cap \mathbb{N})^5$. After all, for any counterexample to Euler's Conjecture, it must be the case that $b > a_i$ for all i. The code above is wasting large amounts of computational effort on combinations that could never work. So let us instead try to improve this in code block 8.2.

In code block 8.2, the bounds for the inner loops now depend on the values of the outer variables. Instead of searching every point in

[1] L. J. Lander and T. R. Parkin. Counterexample to Euler's conjecture on sums of like powers. *Bulletin of the American Mathematical Society*, 72(6):1079, 1966

[2] You can control the formatting of strings inside curly braces with a colon. For some examples, see `https://builtin.com/data-science/python-f-string`.

Code block 8.1. First attempt at Euler's Conjecture; very slow.

```
def lander_parkin(N):
    tests = 0

    for b in range(1, N):
        for a0 in range(1, N):
            for a1 in range(1, N):
                for a2 in range(1, N):
                    for a3 in range(1, N):
                        # Count how many times we test for equality
                        tests = tests + 1

                        if a0**5 + a1**5 + a2**5 + a3**5 == b**5:
                            # Spread printout over a few lines
                            # to stay inside box
                            msg = f"{a0}**5 + {a1}**5 + " + \
                                  f"{a2}**5 + {a3}**5 " + \
                                  f"== {b}**5"
                            print(msg)
                            return tests

    return tests

N = 21  # upper limit
import time

start = time.time()  # record the start time of the loop
tests = lander_parkin(N)
end = time.time()  # record the end time of the loop

print(f"Total time: {end - start:.2f} s")
print(f"Total number of tests: {tests}")
```

Code block 8.2. Second attempt at Euler's Conjecture; triangular search.

```
def lander_parkin(N):
    tests = 0

    for b in range(1, N):
        for a0 in range(1, b):  # only loop up to b
            for a1 in range(1, a0+1):  # etc
                for a2 in range(1, a1+1):
                    for a3 in range(1, a2+1):
                        tests = tests + 1

                        if a0**5 + a1**5 + a2**5 + a3**5 == b**5:
                            msg = f"{a0}**5 + {a1}**5 + " + \
                                  f"{a2}**5 + {a3}**5 " + \
                                  f"== {b}**5"
                            print(msg)
                            return tests

    return tests

N = 21
import time

start = time.time()
tests = lander_parkin(N)
end = time.time()

print(f"Total time: {end - start:.2f} s")
print(f"Total number of tests: {tests}")
```

a five-dimensional lattice, as we were previously doing, we are now only searching in a five-dimensional triangular corner of it. How much does this improve things by? Quite a lot. On the same laptop, running this code yields

```
Total time: 0.03 s
Total number of tests: 33649
```

for a dramatic reduction in the number of points in the lattice we must check (only about 1% of them for `N = 21`, even less for larger `N`).

There are still some points in the lattice that we are checking that we shouldn't. For example, if $a_0^5 + a_1^5 + a_2^5 + a_3^5 > b^5$, there is no point in incrementing a_3 and checking that value also, since it can only *increase* the left-hand side. Before moving on to other code improvements, let us make this small refinement in code block 8.3.

Running code block 8.3 yields

```
Total time: 0.03 s
Total number of tests: 27674
```

for a small but worthwhile improvement (especially for larger `N`).

Looking at our code, we now notice some repetition. We evaluate `b**5` hundreds or thousands of times for each value of `b`; more generally, we evaluate the fifth power of the same integers over and over again. We should *cache* the result of this computation, storing a map from an integer to its fifth power, trading a modest increase in memory usage for a large increase in speed[3]. It is natural to store this as a list (such that `quintics[i] == i**5`), and to compute this list using a list comprehension:

```
quintics = [x**5 for x in range(N)]
```

which is done in code block 8.4.

For the same number of tests, code block 8.4 is about 3× faster:

```
Total time: 0.01 s
Total number of tests: 27674
```

This code is now fast enough to run in anger—if you set `N = 145`, the code in code block 8.4 finds the Lander–Parkin counterexample in 52.71 seconds on the same computer[4]. Much better than three days!

[3] Code optimisations are often of this flavour, trading one attribute for another.

[4] Of course the code can always be made faster. One obvious step would be to cache the lookups of the `quintics` list inside the loop. But most code doesn't need to be the fastest it could possibly be, it just needs to be fast enough.

Code block 8.3. Third attempt at Euler's Conjecture; exclude more cases that cannot work.

```
def lander_parkin(N):
    tests = 0

    for b in range(1, N):
        for a0 in range(1, b):
            for a1 in range(1, a0+1):
                for a2 in range(1, a1+1):
                    for a3 in range(1, a2+1):
                        # Evaluate left-hand side of expression
                        lhs = a0**5 + a1**5 + a2**5 + a3**5
                        if lhs > b**5:
                            break

                        tests = tests + 1

                        # We've already evaluated lhs, use it again
                        if lhs == b**5:
                            msg = f"{a0}**5 + {a1}**5 + " + \
                                  f"{a2}**5 + {a3}**5 " + \
                                  f"== {b}**5"
                            print(msg)
                            return tests

    return tests

N = 21
import time

start = time.time()
tests = lander_parkin(N)
end = time.time()

print(f"Total time: {end - start:.2f} s")
print(f"Total number of tests: {tests}")
```

Code block 8.4. Fourth attempt at Euler's Conjecture; cache the computation of fifth powers.

```
def lander_parkin(N):
    tests = 0

    # Cache map from x to x**5
    quintics = [x**5 for x in range(N)]

    for b in range(1, N):
        for a0 in range(1, b):
            for a1 in range(1, a0+1):
                for a2 in range(1, a1+1):
                    for a3 in range(1, a2+1):
                        lhs = (
                                quintics[a0]
                              + quintics[a1]
                              + quintics[a2]
                              + quintics[a3]
                              )
                        if lhs > quintics[b]:
                            break

                        tests = tests + 1

                        if lhs == quintics[b]:
                            msg = f"{a0}**5 + {a1}**5 + " + \
                                  f"{a2}**5 + {a3}**5 " + \
                                  f"== {b}**5"
                            print(msg)
                            return tests

    return tests

N = 21
import time

start = time.time()
tests = lander_parkin(N)
end = time.time()

print(f"Total time: {end - start:.2f} s")
print(f"Total number of tests: {tests}")
```

9 Problem sheet 2

1. Eratosthenes of Cyrene (c. 276 BC–194 BC) was a Greek polymath, making major contributions in mathematics, geography, poetry, and astronomy. He was the chief librarian of the Library of Alexandria, the greatest centre of learning in the classical world. He is remembered foremost for making the first accurate estimate of the circumference of the Earth. He did this by examining the shadows cast by rods of known length in Alexandria and Syene; his estimate was within one or two percent of the true value.

   The *Sieve of Eratosthenes* is an algorithm for enumerating all prime numbers up to a given value N. The algorithm proceeds as follows.

   (a) Associate to each number $n = 2, \ldots, N$ a Boolean flag for primality, true or false. Set all flags to true.

   (b) Fetch the first unprocessed number n whose flag is true. (On the first iteration, this will be $n = 2$.) Terminate if $n^2 > N$.

   (c) Mark all multiples of n as composite by setting their flag to false.

   (d) Go to step 2.

   On termination, the primes are those numbers with flag true.

   Eratosthenes has now requested your help. He wants to count the number of prime numbers up to $N = 1,000,000$, but he is too busy measuring the distance from Syene to Alexandria. Write a program for Eratosthenes to calculate the number of primes $p \leqslant 1,000,000$. Your program should define a function `primes(N)` that returns a list of primes up to N.

   [*Hint: it is more convenient to make the prime flag a numpy array with* `prime = numpy.array([True for n in range(N+1)])`*, rather than with a list. This means you can use convenient slicing and striding notation for the update of all composite numbers for a given n.*]

2. Alphonse de Polignac (1826–1863) was a French prince, military officer, and mathematician. He was born to an aristocratic family:

his grandmother had been governess to the children of Marie Antoinette, while his father served as prime minister to Charles X until the overthrow of the Bourbon dynasty in 1830. de Polignac served in the Crimean War as an artillery officer, and published on number theory in his spare time.

In his first year of studies at the École Polytechnique, de Polignac made a famous conjecture: for every positive even integer k, there are infinitely many prime gaps of size k. For $k = 2$, this is known as the twin primes conjecture. The conjecture remains open. Oxford's James Maynard was awarded a Fields Medal in 2022 in part for his work on gaps between the primes.

de Polignac has now demanded your help. (He is an aristocrat, after all.) He wishes to compute all twin primes (pairs $(p, p+2)$ with both prime) with $p < 2,000$, but the Crimean War has diverted his attention. Using your `primes(N)` function, calculate all twin primes with $p < 2,000$.

[*Challenge: given* `p = primes(N)`*, can you do this in one line?*]

[*Print the pairs of twin primes, rather than counting them.*]

3. Johann Elert Bode (1747–1826) was a Holy Roman Imperial astronomer. After the discovery of Uranus by Herschel, Bode realised that Uranus had been mistakenly recorded as a star in several previous almanacs, allowing him to calculate its orbit for the first time. He also proposed the name Uranus for the planet; since Saturn was the father of Jupiter, the planet should be named after the father of Saturn. (Herschel attempted to name it after King George III; favourably disposed, the King awarded Herschel an annual stipend of £200.)

   In 1772, in a footnote in his book *Anleitung zur Kenntniss des gestirnten Himmels*, Bode wrote

   > This latter point seems in particular to follow from the astonishing relation which the known six planets observe in their distances from the Sun. Let the distance from the Sun to Saturn be taken as 100, then Mercury is separated by 4 such parts from the Sun. Venus is $4 + 3 = 7$. The Earth $4 + 6 = 10$. Mars $4 + 12 = 16$. Now comes a gap in this so orderly progression. After Mars there follows a space of $4 + 24 = 28$ parts, in which no planet has yet been seen. Can one believe that the Founder of the Universe had left this space empty? Certainly not. From here we come to the distance of Jupiter by $4 + 48 = 52$ parts, and finally to that of Saturn by $4 + 96 = 100$ parts.

In units of AU (astronomical units, almost but not quite the average distance from the Earth to the Sun), the Titius–Bode law is that the average distance of planet n from the Sun is

$$d(n) = 0.4 + 0.3 \times 2^n, \quad n = -\infty, 0, 1, \ldots, 8.$$

Bode has now requested your help. With the discoveries of Uranus, Ceres, Neptune, and Pluto, he wishes to visualise whether the Titius–Bode law is valid or not. Find a table of distances of planets and dwarf planets to the Sun. Plot the measured distances of all 8 planets, as well as the dwarf planets Ceres and Pluto, ordering them by distance to the Sun. (Recall that Ceres lies in Bode's 'gap' between Mars and Jupiter.) On the same plot, plot the predictions of the Titius–Bode law.

[*Hint: you can plot multiple functions on the same plot with multiple calls to* `plt.plot`*. For each call to* `plt.plot`*, pass an optional argument* `label="Label here"`*. Put all matplotlib calls on adjacent lines to ensure they render correctly in the published version. Use* `plt.legend()` *to show the labels.*]

[*Hint: look up the documentation for* `plt.xticks`*. Use this function to label the ticks on the x-axis Mercury, Venus, etc. rather than* $-\infty, 0, 1, \ldots$.]

[*Hint: the default resolution of plots saved in* `publish` *is rather low. Use* `plt.gcf().set_dpi(300)` *to fix this.*]

4. Karl Weierstrass was a Prussian mathematician who founded modern analysis. As an undergraduate in Bonn, he spent four years neglecting his studies of law in favour of fencing and drinking, and left without a degree. He became a teacher of mathematics in a secondary school; while on sick leave from teaching he wrote a paper on Abelian functions that won him an honourary doctorate from Königsberg, and he eventually was appointed a Professor at the University of Berlin. Among many other mathematical achievements, he formalised the definition of the continuity of a function which you will meet in Hilary term.

   In 1872 Weierstrass devised a function which is *everywhere continuous but nowhere differentiable*. At the time many mathematicians believed that continuous functions might be non-differentiable only on limited sets, and his counterexample tore up several erroneous

proofs that had implicitly made this assumption. Weierstrass' function is given by

$$f(x) = \sum_{k=0}^{\infty} a^k \cos\left(b^k \pi x\right),$$

where $a \in (0,1)$ and b is a positive odd integer satisfying $ab > 1 + 3\pi/2$. Poincaré denounced Weierstrass' work as ''an outrage against common sense''; Hermite described it as a ''lamentable scourge''.

Weierstrass has now requested your help. To convince his contemporaries to overcome their incorrect intuition, he wishes to visualise this function, but ill health means he cannot do the necessary calculations. Write a program for Weierstrass that visualises Weierstrass' function on $[-2,2]$ with default values of parameters $a = 0.3, b = 23$, approximating the infinite sum by default with 100 terms.

10 Introduction to symbolic computing

10.1 What is symbolic computing?

In this chapter we will study symbolic computing, the use of computers to manipulate and solve mathematical expressions. Symbolic computing is also sometimes known as computer algebra. Symbolic computing systems aim to mechanise and automate the kind of manipulations you do with a pen on paper—as we will see, they will allow us to simplify and factor expressions, differentiate functions, calculate limits and indefinite integrals, and symbolically solve algebraic and differential equations.

The symbolic computing system we will use in this course is SymPy[1] (henceforth sympy). Many other computer algebra systems exist, including Mathematica, Maple, GAP, Axiom, FriCAS, Magma, and SageMath[2]. Familiarity with sympy will be useful for all of these.

To install sympy, at the terminal type

(terminal)
```
pip install sympy
```

To get a sense of the distinction between symbolic computing and numerical computing, one might consider taking square roots:

(python)
```
import math
```

(python)
```
print(f"√2**2: {math.sqrt(2)**2:.30f}")
```
```
√2**2: 2.000000000000000044089209850063
```

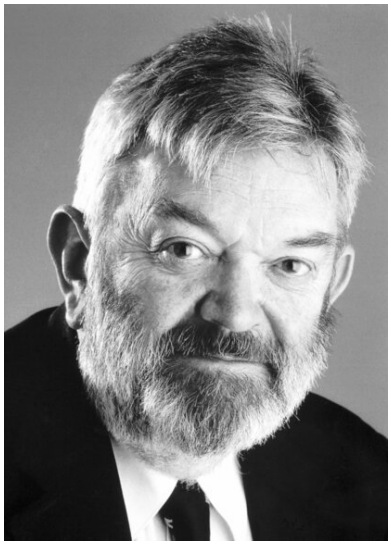
When we calculate a square root with `math.sqrt`, it stores a representation of the number to 15 decimal digits. This means that when we square the output again, the result is not quite 2 (but it is close). By contrast, if we take the square root with sympy, it returns to us a *symbolic* object whose defining feature is that its square is 2:

(python)
```
import sympy as sp
```

(python)
```
print(f"√2**2: {sp.sqrt(2)**2}")
```

[1] A. Meurer et al. SymPy: symbolic computing in Python. *PeerJ Computer Science*, page e103, 2017

[2] The first computer algebra system, Schoonschip, was written by Martinus Veltman in 1963. Veltman went on to win the Nobel Prize in Physics for his work in particle theory.



Martinus Veltman, 1931–2021

```
√2**2: 2
```

where the output is an integer (sympy's own type of integer, but an integer nonetheless)[3]. The output of `sp.sqrt` is not a number: it is a symbol representing a number via its mathematical properties.

Symbolic computing is an extremely powerful assistant in learning and doing mathematics. When exploring a subject, it takes the burden of the sometimes tedious calculations involved. When your calculations on a problem sheet go awry, you can use the computer to find the step with a mistake. Mastering the basics of it will ease your path through your undergraduate degree.

Symbolic computing is an old idea, dating back to the very first conceptions of steam-powered computers. In 1843, in her translator's notes accompanying a paper describing Charles Babbage's Analytical Engine, Ada Lovelace wrote[4]

> Many persons who are not conversant with mathematical studies imagine that because the business of the engine is to give its results in numerical notation, the nature of its processes must consequently be arithmetical and numerical rather than algebraic and analytical. This is an error. The engine can arrange and combine its numerical quantities exactly as if they were letters or any other general symbols; and in fact it might bring out its results in algebraic notation were provisions made accordingly[5].

[3] Here we have imported `sympy` as `sp` to abbreviate our code. We will always explicitly access sympy objects via `sp.`, to clearly distinguish symbolic objects from others.

[4] L. F. Menabrea of Turin, Officer of the Military Engineers. Sketch of the analytical engine invented by Charles Babbage, Esq. *Scientific Memoirs, Selected from the Transactions of Foreign Academies of Science and Learned Societies*, 3:666–731, 1843. Translated by A. King, Countess of Lovelace.

[5] Lovelace wrote the first ever computer program, to compute Bernoulli numbers on the Analytical Engine. The Analytical Engine was never built; the first programmable computer was built nearly a century later, by Konrad Zuse in 1941.

10.2 *Symbols and expressions*

Symbolic computing is built on symbols, unsurprisingly. In sympy we can make a symbol with `sp.Symbol`. When making a symbol, you pass a name that sympy will use to render the symbol:

```
(python)  x = sp.Symbol("x")
(python)  y = sp.Symbol("y")
```

Giving a symbol a variable name different to its internal name is a path to madness; please never do it. With these symbols, we can make *expressions*:

```
(python)  2*x + x + 5
           3*x + 5
(python)  2*x/6
           x/3
(python)  2*x/x
           2
```

Figure 10.1: Ada King, Countess of Lovelace, 1815–1852

We see that sympy automatically simplifies expressions using the same algebraic rules you might on paper. However, it does not do trigonometric simplifications by default:

```
(python)  sp.sin(x)**2 + sp.cos(x)**2
           sin(x)**2 + cos(x)**2
```

To simplify this expression, we need an explicit call to `sp.simplify`:

```
(python)  sp.simplify(sp.sin(x)**2 + sp.cos(x)**2)
           1
```

There are other simplifications that sympy does not do by default. If you give it an expression in factored form, it prefers it, as there is more information in the factored form:

```
(python)  (x-3)*(x+5)
           (x - 3)*(x + 5)
```

To ask sympy to expand the brackets, use `sp.expand`:

```
(python)  sp.expand((x-3)*(x+5))
           x**2 + 2*x - 15
```

To do the opposite, use `sp.factor`:

```
(python)  sp.factor(x**2 + 2*x - 15)
           (x - 3)*(x + 5)
```

This does not always help, however. The algorithm in `sp.factor` returns factors that are irreducible *over the rationals*; factors associated with irrational roots will not be returned. In addition, there exists no algorithm that will exactly find the roots of all polynomials of degree 5 and above. Here are examples where `sp.factor` tells us nothing:

```
(python)  sp.factor(x**2 - 2)
           x**2 - 2
(python)  sp.factor(x**5 - x - 1)
           x**5 - x - 1
```

This latter polynomial is an example of a non-solvable polynomial, one that cannot be solved in radicals[6].

You can also factor expressions with multiple variables:

```
(python)  xv = sp.symbols("x0:2")
(python)  xv
           (x0, x1)
```

Évariste Galois, 1811–1832

[6] The investigation of such problems inspired Évariste Galois to lay the foundations of group theory and Galois theory, solving a problem that had been open for 350 years. He died in a duel at the age of 20. For an example of how sympy can be used to explore Galois theory, see

C. S. Covaci. The Unsolvability of the Quintic: an Insight into Galois Theory. Master's thesis, Universidad Politécnica de Madrid, 2022

```
(python)  expr = xv[0]**2 * xv[1] + xv[0] * xv[1]
(python)  expr
           x0**2*x1 + x0*x1
```

Here we make a tuple of symbols using a similar syntax to `range`—the upper limit is not included. We make an expression with a common factor $x_0 x_1$. `sp.factor` can identify this for us:

```
(python)  sp.factor(expr)
           x0*x1*(x0 + 1)
```

Sympy includes various useful mathematical constants, like π (`sp.pi`), i (`sp.I`), and e (`sp.E`). Unlike their counterparts in the `math` module, these are symbols with the right mathematical properties, not approximations:

```
(python)  sp.I**2
           -1
(python)  sp.E**(sp.I * sp.pi)
           -1
```

encoding Euler's identity $e^{i\pi} = -1$. Sympy has a constant representing ∞, which will be useful later in limits and integrals, given by `sp.oo`. Sympy also has symbolic versions of all the mathematical functions you are familiar with, like `sin`, `atan`, `exp`, `log`, as well as some you will meet in your undergraduate studies, like the Gamma function $\Gamma(z)$, Bessel functions, Airy functions, spherical harmonics, the Meijer-G function, and many others[7].

[7] For a list, see `https://docs.sympy.org/latest/modules/functions/special.html`.

10.3 *Assumptions and evaluation*

By default, sympy treats all symbols as representing complex numbers in $\mathbb{C}$. However, there are many simplifications that only hold if a variable is real, positive, integer, etc. For example, the relation

$$\log(\exp(x)) = x$$

only holds for $x \in \mathbb{R}$. As a result, sympy does not simplify such an expression:

```
(python)  z = sp.Symbol("z", complex=True) # default
(python)  sp.log(sp.exp(z))
           log(exp(z))
```

but if we tell sympy a variable is real, it will:

```
(python)  r = sp.Symbol("r", real=True)

(python)  sp.log(sp.exp(r))
           r
```

You can also tell sympy that a variable is positive, which enables further simplifications:

```
(python)  p = sp.Symbol("p", positive=True)

(python)  sp.sqrt(r**2)
           Abs(r)
(python)  sp.sqrt(p**2)
           p
```

or tell sympy that a variable is an integer:

```
(python)  n = sp.Symbol("n", integer=True)

(python)  (-1)**(2*p)
           (-1)**(2*p)
(python)  (-1)**(2*n)
           1
```

In general, when solving complicated problems with sympy, it is best practice to give it as many assumptions as possible. As of writing, the list of attributes you can assume is: `positive`, `nonpositive`, `nonnegative`, `extended_nonzero`, `real`, `finite`, `hermitian`, `nonzero`, `commutative`, `zero`, `extended_real`, `infinite`, `extended_nonpositive`, `imaginary`, `negative`, `extended_nonnegative`, `extended_negative`, `extended_positive`, `complex`, `integer`, `irrational`, `algebraic`, `rational`, `transcendental`, `noninteger`.

To evaluate an expression, we use its `.subs` method:

```
(python)  x = sp.Symbol("x")

(python)  expr = x**17 + x**6 - x**3

(python)  expr.subs({x: 6})
           16926659491176
```

Here `.subs` takes in a dictionary mapping the symbols to be substituted to their values. This does not change the original expression. You can also substitute expressions in for symbols:

```
(python)  expr.subs({x: x**2})
           x**34 + x**12 - x**6
```

Substituting values does not always yield a number:

```
(python)  expr = sp.sin(x**3 * sp.pi) + sp.exp(sp.cos(x))
(python)  expr.subs({x: 1.5})
           sin(1.375*pi) + 1.07329912758172
```

To numerically evaluate this, use `sp.N`:

```
(python)  sp.N(expr.subs({x: 1.5}))
           0.149419595070430
```

By default this returns 15 decimal places. You can change this with the second argument:

```
(python)  sp.N(expr.subs({x: 1.5}), 40)
           0.1494195950704302157183206492089248124188
```

Note that this returns sympy's own (arbitrary-precision) floating-point datatype. This may or may not be understood by other Python packages; if you run into trouble interoperating with another package, cast the output of `sp.N` to Python's native floating-point datatype using `float`.

A neat feature of sympy is that you can use it to make a Python function that evaluates your expression with `sp.lambdify`[8]. This can then be used as normal Python, e.g. for plotting with matpotlib.

```
(python)  f = sp.lambdify(x, expr)
(python)  f(1.5)
           0.14941959507043046
```

Figure 10.2: Alonzo Church, 1903–1995

[8] This name refers to λ-calculus, a formal system of mathematical logic for expressing computation introduced by Alonzo Church in 1936. Alongside his doctoral student Alan Turing, Church is considered one of the founders of computer science.

10.4 *Solving algebraic equations*

We can use sympy to solve algebraic equations using `sp.solveset`. We write all terms of the equation on the left-hand side, to make an expression whose roots we wish to find[9]. For example, to find the solutions of $x^2 = 5$, we might do

[9] Alternatively, you can specify a right-hand side with `sp.solveset(sp.Eq(lhs, rhs), x)`.

```
(python)  x = sp.Symbol("x")
(python)  sp.solveset(x**2 - 5, x)
```

```
{-sqrt(5), sqrt(5)}
```

The first argument is the expression to find the roots of; the second argument is the variable to solve for[10]. The solution sets may not always be finite. For example, solving $\cos x = \sin x$ with

[10] The variable to solve for is optional if there is only one free symbol.

```
(python)  sp.solveset(sp.cos(x) - sp.sin(x), x)
```

yields a set which in mathematical notation[11] is

$$\left\{2n\pi + \frac{5\pi}{4} \;\middle|\; n \in \mathbb{Z}\right\} \cup \left\{2n\pi + \frac{\pi}{4} \;\middle|\; n \in \mathbb{Z}\right\}.$$

[11] A *very* useful feature of sympy is that the `sp.print_latex` function to any sympy object makes LaTeX code for rendering it.

As with `sp.factor`, there are fundamental mathematical limitations to equation solvers in any symbolic computing system: Richardson's theorem asserts that finding a complete set of solutions for an equation is undecidable[12]. As an example where sympy fails to tell us anything useful, consider solving $\cos x = x$:

[12] D. Richardson. Some undecidable problems involving elementary functions of a real variable. *Journal of Symbolic Logic*, 33(4):514–520, 1969

```
(python)  sp.solveset(sp.cos(x) - x, x)
          ConditionSet(x, Eq(-x + cos(x), 0), Complexes)
```

which in mathematical notation is

$$\{x \mid x \in \mathbb{C} \wedge -x + \cos(x) = 0\},$$

or in other words the solution set returned is tautological ($\wedge$ means 'and'); it just says that the solution is the set of complex numbers satisfying the equation. In Prelims *Constructive Mathematics* you will learn how to prove that this equation has in fact a unique real solution, and study algorithms for approximating its solution with lightning speed (and many other equations that simply cannot be handled symbolically).

Sympy can also solve systems of linear and nonlinear equations. To solve a system of linear equations, use `sp.linsolve`:

```
(python)  (x, y, z) = sp.symbols("x, y, z")
```

```
(python)  sp.linsolve([3*x + 5*y - 3*z - 1, x + y + z - 3], (x, y, z))
          {(7 - 4*z, 3*z - 4, z)}
```

Here we have supplied two equations in three unknowns, so there is at least a one-dimensional family of solutions[13]. To solve a system of nonlinear equations, use `sp.nonlinsolve`:

[13] This follows from the rank-nullity theorem, which you will meet in *M1: Linear Algebra*.

```
(python)  sp.nonlinsolve([x**2 - 2*y**2 -2, x*y - 2], (x, y))
          {(-2, -1), (2, 1), (-sqrt(2)*I, sqrt(2)*I), (sqrt(2)*I, -sqrt(2)*I)}
```

Lastly, we mention the `sp.solve` function. This has largely been superceded by `sp.solveset`, since it does not generally return all solutions to an equation. Considering the same example as before,

```
(python)  sp.solve(sp.cos(x) - sp.sin(x), x)
           [pi/4]
```

we see that `sp.solve` yields only partial information. However, it is worth knowing, since it is better than `sp.solveset` at algebraic manipulations[14]. If you have an expression involving various symbols and want to write one symbol in terms of the others, you should use `sp.solve`. For example,

[14] In particular, `sp.solve` is better than `sp.solveset` when it comes to algebraic manipulations involving sympy *functions*, which we discuss below in section 10.7.

```
(python)  sp.solve(3*x + sp.sin(y)**2 * sp.exp(z), x)
           [-exp(z)*sin(y)**2/3]
```

10.5 *Differentiation and integration*

Sympy understands calculus, not just algebra. It can therefore be very useful for solving problems on your problem sheets in other courses, or verifying your answers. To differentiate an expression, use `sp.diff`:

```
(python)  x = sp.Symbol("x")
(python)  expr = sp.sin(sp.exp(x)) + 3*sp.cos(x**2)
(python)  sp.diff(expr, x)
           -6*x*sin(x**2) + exp(x)*cos(exp(x))
```

You can also use the equivalent method `expr.diff(x)`. You can differentiate an expression multiple times:

```
(python)  sp.diff(expr, x, 2)
           -12*x**2*cos(x**2) - exp(2*x)*sin(exp(x)) + exp(x)*cos(exp(x)) - 6*sin(x**2)
```

There is a fair chance you would make a mistake if you had to calculate the 20^{th} derivative of this expression, but sympy handles it in a second: `sp.diff(expr, x, 20)` yields

```
3145728*x**20*cos(x**2) +
298844160*x**18*sin(x**2) -
11430789120*x**16*cos(x**2) -
228615782400*x**14*sin(x**2) +
2600504524800*x**12*cos(x**2) +
17163329863680*x**10*sin(x**2) -
64362486988800*x**8*cos(x**2) -
```

```
128724973977600*x**6*sin(x**2) +
120679663104000*x**4*cos(x**2) +
40226554368000*x**2*sin(x**2) +
exp(20*x)*sin(exp(x)) -
190*exp(19*x)*cos(exp(x)) -
15675*exp(18*x)*sin(exp(x)) +
741285*exp(17*x)*cos(exp(x)) +
22350954*exp(16*x)*sin(exp(x)) -
452329200*exp(15*x)*cos(exp(x)) -
6302524580*exp(14*x)*sin(exp(x)) +
61068660380*exp(13*x)*cos(exp(x)) +
411016633391*exp(12*x)*sin(exp(x)) -
1900842429486*exp(11*x)*cos(exp(x)) -
5917584964655*exp(10*x)*sin(exp(x)) +
12011282644725*exp(9*x)*cos(exp(x)) +
15170932662679*exp(8*x)*sin(exp(x)) -
11143554045652*exp(7*x)*cos(exp(x)) -
4306078895384*exp(6*x)*sin(exp(x)) +
749206090500*exp(5*x)*cos(exp(x)) +
45232115901*exp(4*x)*sin(exp(x)) -
580606446*exp(3*x)*cos(exp(x)) -
524287*exp(2*x)*sin(exp(x)) +
exp(x)*cos(exp(x)) -
2011327718400*cos(x**2)
```

Sympy can also compute partial derivatives of expressions involving multiple symbols.

```
(python)  y = sp.Symbol("y")

(python)  expr = sp.sin(x*16 * sp.exp(y)) + sp.gamma(y**2) * sp.erf(x)

(python)  sp.diff(expr, x, 1)
           16*exp(y)*cos(16*x*exp(y)) + 2*exp(-x**2)*gamma(y**2)/sqrt(pi)
(python)  sp.diff(expr, x, 2, y, 2) # mixed partial derivatives
           8*(8192*x**2*exp(4*y)*sin(16*x*exp(y)) -
           2*x*y**2*exp(-x**2)*gamma(y**2)*polygamma(0, y**2)**2/sqrt(pi) -
           2*x*y**2*exp(-x**2)*gamma(y**2)*polygamma(1, y**2)/sqrt(pi) -
           2560*x*exp(3*y)*cos(16*x*exp(y)) -
           x*exp(-x**2)*gamma(y**2)*polygamma(0, y**2)/sqrt(pi) -
           128*exp(2*y)*sin(16*x*exp(y)))
```

Sympy can calculate both indefinite and definite integrals, for both single-variable and multi-variable expressions. Here are some examples. We first consider integrating the probability density function of

the Gaussian distribition

$$\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}x^2\right) \mathrm{d}x$$

with

```
(python)  sp.integrate(1/sp.sqrt(2*sp.pi) * sp.exp(-x**2/2), (x, -sp.oo, +sp.oo))
           1
```

Here the second argument is a tuple describing the variable to integrate against, the lower limit of the integral, and the upper limit of the integral (recall that `sp.oo` represents ∞). The limits of our definite integrals can also be symbols. For example, the error function erf often arises in statistics; if a random variable Y is normally distributed with mean 0 and standard deviation $1/\sqrt{2}$, the probability that Y falls within $[-x, x]$ is given by $\operatorname{erf} x$. It is defined as

$$\operatorname{erf} x = \frac{2}{\sqrt{\pi}} \int_0^x \exp\left(-t^2\right) \mathrm{d}t.$$

Sympy can identify this:

```
(python)  t = sp.Symbol("t")
```

```
(python)  sp.integrate(2/sp.sqrt(sp.pi) * sp.exp(-t**2), (t, 0, x))
           erf(x)
```

Indefinite integrals can also be handled, by leaving out the limits of integration[15]. This can handle imposing integrals such as

$$\int \frac{x^2 + 2x + 1 + (3x+1)\sqrt{x + \log x}}{x\sqrt{x + \log x}(x + \sqrt{x + \log x}} \mathrm{d}x$$

with

[15] Symbolic computing packages generally use the Risch algorithm for finding antiderivatives. Describing the algorithm takes over 100 pages. Generations of calculus students worldwide depend on it.

R. H. Risch. The problem of integration in finite terms. *Transactions of the American Mathematical Society*, 139(0):167–189, 1969

```
(python)  top = x**2 + 2*x + 1 + (3*x+1)*sp.sqrt(x + sp.log(x))
```

```
(python)  bot = x*sp.sqrt(x + sp.log(x)) * (x + sp.sqrt(x + sp.log(x)))
```

```
(python)  sp.integrate(top / bot, x)
           2*sqrt(x + log(x)) + 2*log(x + sqrt(x + log(x)))
```

Note that the constant of integration is omitted. Sympy's indefinite integral algorithm is not perfect, however[16]. Chebyshev[17] calculated the elementary antiderivative of

$$\frac{x}{\sqrt{x^4 + 10x^2 - 96x - 71}},$$

but sympy cannot find it:

[16] Sympy does not implement the full Risch algorithm. In fact, there is no known full implementation of the entire algorithm in any package. For a discussion, see https://mathoverflow.net/questions/374089/does-there-exist-a-complete-implementation-of-the-risch-algorithm.

[17] P. L. Chebyshev. *Oeuvres de P. L. Tchébychef*, volume 1. Commissionaires de l'Académie Impériale des Sciences, 1899–1907

```
(python)  sp.integrate(x/sp.sqrt(x**4 + 10*x**2 - 96*x - 71), x)
           Integral(x/sqrt(x**4 + 10*x**2 - 96*x - 71), x)
```

Mathematica, however, can. Intriguingly, if the constant term 71 is changed to 72, it is not possible to represent its antiderivative in terms of elementary functions[18].

[18] G. Zolotareff. Sur la méthode d'intégration de M. Tchébychef. *Mathematische Annalen*, 5(4):560–580, 1872

We close with an example of multivariate integration. Consider a tetrahedron with its apex at the origin and edges of length ℓ along the x-, y-, and z-axes. By integrating 1 over the tetrahedron, we can calculate its volume with

$$\int_0^\ell \int_0^{\ell-x} \int_0^{\ell-x-y} 1 \,\mathrm{d}z \,\mathrm{d}y \,\mathrm{d}x,$$

which evaluates to $\ell^3/6$. In code this becomes

```
(python)  z = sp.Symbol("z")
```

```
(python)  l = sp.Symbol("l")
```

```
(python)  sp.integrate(1, (z, 0, l-x-y), (y, 0, l-x), (x, 0, l))
           l**3/6
```

Exercise 10.1. Consider a curve parameterised by $t \in [0, 1]$, given by

$$(x, y, z) = (0, t, t^2). \tag{10.5.1}$$

Calculate the arclength L of this curve with

$$L = \int_0^1 \sqrt{(\mathrm{d}x/\mathrm{d}t)^2 + (\mathrm{d}y/\mathrm{d}t)^2 + (\mathrm{d}z/\mathrm{d}t)^2}\,\mathrm{d}t. \tag{10.5.2}$$

Exercise 10.2. Evaluate the integral[19]

$$f(t) = \int_1^t x^{10} \exp x \,\mathrm{d}x. \tag{10.5.3}$$

[19] Doing this by hand might require applying integration by parts many times.

Exercise 10.3. A falling object o encounters a moving platform p accelerating upwards. Their heights for a given time $t \geqslant 0$ are

$$h_o(t) = h_0 - v_0 t - \frac{1}{2} g t^2, \tag{10.5.4}$$

$$h_p(t) = \quad v_p t + \frac{1}{2} q t^2, \tag{10.5.5}$$

where $v_0, v_p \geqslant 0$ are the initial velocities, and $g, q > 0$ are the (constant) accelerations. Find the initial velocity v_0 such that when the object and platform collide, they are moving at the same speed.

10.6 *Limits, sequences, and series*

Sympy can evaluate limits like

$$\lim_{x\to\infty} x \sin\left(\frac{1}{x}\right) = 1, \quad \lim_{x\to\infty} x \exp(-x) = 0,$$

with

```
(python)  sp.limit(x*sp.sin(1/x), x, sp.oo)
           1
(python)  sp.limit(x*sp.exp(-x), x, sp.oo)
           0
```

This uses the Gruntz algorithm[20] for calculating the symbolic limits. By default, this evaluates the limit *from the right*: to specify whether you want a right-sided limit or left-sided limit, use `'+'` or `'-'` respectively. Here we calculate

$$\lim_{x\to 0^+} \frac{\mathrm{d}}{\mathrm{d}x}|x| = 1, \quad \lim_{x\to 0^-} \frac{\mathrm{d}}{\mathrm{d}x}|x| = -1.$$

[20] D. Gruntz. *On computing limits in a symbolic manipulation system*. PhD thesis, ETH Zürich, 1996

```
(python)  r = sp.Symbol("r", real=True)
(python)  sp.limit(sp.diff(sp.Abs(r), r), r, 0, '+')
           1
(python)  sp.limit(sp.diff(sp.Abs(r), r), r, 0, '-')
           -1
```

Sympy can calculate the limits of sequences. Here is sympy calculating the answers to some of your Analysis I homework:

```
(python)  n = sp.Symbol("n")
(python)  sp.limit_seq((n**2 + n + 1) / (n + 1), n) # Sheet 3
           oo
(python)  sp.limit_seq((1 + 1/n)**n, n)
           E
(python)  sp.limit_seq(((1 - sp.I)*n) / (n + sp.I), n)
           1 - I
(python)  sp.limit_seq(n**2 / sp.factorial(n), n) # Sheet 4
           0
```

Sympy can also evaluate series, although the algorithm is far from robust[21]. Here is the one piece of Analysis I homework it can do:

[21] If you wanted to get started with open-source mathematical software development, this could be a good place to start contributing to sympy; it appears sympy lacks strategies that are known by Prelims students.

```
(python)  n = sp.Symbol("k")
(python)  sp.Sum((2*k + 1)/((k+1)*(k+2)**2), (k, 0, sp.oo)).doit() # Sheet 5
           -4 + pi**2/2
```

Here the `.doit()` method is required to evaluate the sum, rather than to represent it symbolically.

10.7 *Solving differential equations*

Sympy can solve (some) ordinary differential equations. For example, let us consider the solution of

$$xf''(x) + f'(x) = x^3 \tag{10.7.1}$$

subject to

$$f(1) = 0, \quad f'(2) = 1. \tag{10.7.2}$$

To represent an unknown function, we define a `sp.Function` object:

```
(python)  x = sp.Symbol("x")
(python)  f = sp.Function("f")(x)
```

Here we indicate to sympy that the function depends on x. We can represent the derivatives, integrals, or evaluations of this unknown function symbolically:

```
(python)  f.diff(x)
           Derivative(f(x), x)
(python)  f.integrate(x)
           Integral(f(x), x)
(python)  f.subs({x: 0})
           f(0)
```

To represent the differential equation (10.7.1), we use a `sp.Eq` object:

```
(python)  eq = sp.Eq(x*f.diff(x, 2) + f.diff(x), x**3)
(python)  eq.lhs
           x*Derivative(f(x), (x, 2)) + Derivative(f(x), x)
(python)  eq.rhs
           x**3
```

The syntax for constructing the equation is `sp.Eq(lhs, rhs)`, and we can use the attributes `.lhs` and `.rhs` to extract these afterwards. Let us now pass this to sympy's differential equation solver, `sp.dsolve`:

```
(python)  sp.dsolve(eq, f)
           Eq(f(x), C1 + C2*log(x) + x**4/16)
```

which returns a `sp.Eq` representing

$$f(x) = C_1 + C_2 \log(x) + \frac{x^4}{16}, \tag{10.7.3}$$

the general form of the solution, with two unknown constants of integration C_1 and C_2. If we wished to set these by hand, we could do so as follows:

```
(python)  sol = sp.dsolve(eq, f).rhs

(python)  tuple(sol.free_symbols)
           (x, C1, C2)
(python)  (_, C1, C2) = tuple(sol.free_symbols)

(python)  sol.subs({C1: 3, C2: 5})
           x**4/16 + 5*log(x) + 3
```

On the first line, we get the expression for the general solution via the right-hand side of the returned equation. On the second line, we inspect its free symbols (calling `tuple` to convert a set, which we cannot index, to a tuple, which we can). On the third line, we use tuple unpacking to assign variables to the unknown constants of integration C_1 and C_2; assigning to _ is a convention which denotes a variable we do not care about. On the fourth line, we substitute $C_1 = 3$ and $C_2 = 5$ to get a particular solution of the differential equation.

Of course, as you know from *Introductory Calculus*, the reason why these unknown constants of integration arise is because we did not specify the initial conditions (10.7.2). To do so, we build a dictionary mapping expressions to values that should be satisfied by the solution:

```
(python)  ics = {f.subs({x: 1}): 0, f.diff(x).subs({x: 2}): 1}
```

We can now pass these to `sp.dsolve` as an optional argument:

```
(python)  sp.dsolve(eq, f, ics=ics).rhs
           x**4/16 - 2*log(x) - 1/16
```

which returns the solution to (10.7.1)–(10.7.2)

$$f(x) = \frac{x^4}{16} - 2\log(x) - \frac{1}{16}. \tag{10.7.4}$$

Exercise 10.4. Verify with sympy that (10.7.4) satisfies (10.7.1)–(10.7.2).

Exercise 10.5. Plot the solution (10.7.4) using `sp.lambdify`, `linspace` from the `numpy` library, and matplotlib.

10.8 *Coda: rendering sympy objects in published documents*

It is possible to render a sympy object in a document published as described in chapter 5. To do so, call the `render` function that `publish.py` offers. It can be used like so:

Code block 10.1. Demonstrating `publish.render`.

```
from publish import *
import sympy as sp

# # Basic usage of publish.render
# The function makes LaTeX output from sympy objects
# in published documents.

x = sp.Symbol("x")
# Make some expression
expr = sp.E + sp.pi + sp.gamma(x)
render(expr)

# # Optional name argument
# You can also give an optional name argument
# to publish.render. If this is supplied, it
# will render "name = " the expression supplied.

# Make a polynomial
alpha = x**3 + 3*x + 10
render(alpha, name=r"\alpha(x)")

publish()
```

This should yield a `.html` file like the image below.

```
from publish import *
import sympy as sp
```

Basic usage of publish.render

The function makes LaTeX output from sympy objects in published documents.

```
x = sp.Symbol("x")
# Make some expression
expr = sp.E + sp.pi + sp.gamma(x)
render(expr)
```

$$\Gamma\left(x\right) + e + \pi$$

Optional name argument

You can also give an optional name argument to publish.render. If this is supplied, it will render "name = " the expression supplied.

```
# Make a polynomial
alpha = x**3 + 3*x + 10
render(alpha, name=r"\alpha(x)")
```

$$\alpha(x) = x^{3} + 3x + 10$$

11 *Problem sheet 3*

1. Giuseppe Luigi Lagrangia (1736–1813) was a Piedmontese mathematician, physicist, and astronomer. Born in Turin, he first studied law before switching to mathematics after reading a paper by Oxford's Edmund Halley on the use of algebra in optics. While serving as a mathematics instructor at a Piedmontese artillery school, he invented the calculus of variations, a central mathematical tool in understanding the universe. In 1765 he was persuaded to move to Berlin, where he wrote his *magnum opus*, the Mécanique Analytique, which established the Lagrangian formulation of Newtonian mechanics. In 1786 the Prussian king died, and Lagrangia moved to an apartment in the Louvre. As a result he was caught up in the French Revolution; Lagrangia was specifically exempted by name in the decree of October 1793 that ordered all foreigners to leave France, and wound up proposing the use of decimal subdivision in the metric system. He became a firm favourite of Napoleon, who appointed him a senator. Lagrangia was the first signatory of the annexation of Piedmont to France in 1802. He died in 1813, two years before Napoleon's final defeat by the Irish general Wellington at Waterloo.

   Roughly speaking, Lagrangian mechanics formulates problems in terms of tradeoffs between energies, whereas the Newtonian mechanics you study in Prelims *M4 Dynamics* formulates problems in terms of forces. One of the many benefits of this approach is that it allows for the use of *generalised coordinates*: you can express the state of the system in terms of whatever variables are convenient. For example, when describing the motion of a pendulum, in a Newtonian framework one must enforce the constraint that $x(t)^2 + y(t)^2 = \text{constant}$, but in a Lagrangian framework this is enforced naturally by describing the state of the system with the angle $\theta(t)$. No matter the coordinates used to describe the system, Lagrangian mechanics allows you to derive the associated equations of motion. You will learn more if you take the third-year option *B7.1 Classical Mechanics*.

Consider the orbit of the earth (of mass $m \in \mathbb{R}_+$) around the sun (of mass $M \in \mathbb{R}_+$), which is fixed to lie at the origin. It is natural to describe the system in terms of the generalised (polar) coordinates $r(t) \geqslant 0$ and $\theta(t)$. The associated kinetic energy of the system is given by

$$T(r, \theta, \dot{r}, \dot{\theta}) = \frac{1}{2} m \left(\dot{x}^2 + \dot{y}^2 \right),$$

where we consider $x(t), y(t)$ to be functions of (r, θ) given by

$$x = r \cos\theta, \quad y = r \sin\theta.$$

The potential energy of the system is given by the Kepler potential

$$V(r, \theta, \dot{r}, \dot{\theta}) = -\frac{GMm}{r},$$

where $G \in \mathbb{R}_+$ is the gravitational constant. With these, we define the Lagrangian as the difference between these energies,

$$L(r, \theta, \dot{r}, \dot{\theta}) = T(r, \theta, \dot{r}, \dot{\theta}) - V(r, \theta, \dot{r}, \dot{\theta}).$$

Lagrange has now asked for your help. He wishes to derive the equations of motion for the system (called the Euler–Lagrange equations[1]), given by

$$\frac{\mathrm{d}}{\mathrm{d}t} \frac{\partial L}{\partial \dot{\theta}} = \frac{\partial L}{\partial \theta}, \tag{$\star$}$$

$$\frac{\mathrm{d}}{\mathrm{d}t} \frac{\partial L}{\partial \dot{r}} = \frac{\partial L}{\partial r}, \tag{$\dagger$}$$

but is otherwise occupied surviving the French Revolution.

[1] Lagrange wrote *Ce sont ces équations qui serviront à déterminer la courbe décrite par le corps M et sa vitesse à chaque instant ....*

J.-L. Lagrange. Applications de la méthode exposée dans le mémoire précédent à la solution de différents problèmes de dynamique. *Mélanges de Philosophie et de Mathématiques de la Société Royale de Turin*, 2:196–298

(a) Verify that the Lagrangian for the system is given by

$$L(r, \theta, \dot{r}, \dot{\theta}) = \frac{1}{2} m \left(\dot{r}^2 + r^2 \dot{\theta}^2 \right) + \frac{GMm}{r}. \tag{$\ddagger$}$$

(b) Observe that ($\ddagger$) only depends on $\dot{\theta}$; it is independent of θ. This means that the right-hand side of ($\star$) is zero, and that $\partial L / \partial \dot{\theta}$ must be a constant, $\ell > 0$. (In fact, ℓ is the angular momentum.) Use this to derive a symbolic expression for $\dot{\theta}$ in terms of ℓ, m, and r.

(c) Calculate the Euler–Lagrange equation ($\dagger$). Using the expression derived in (b), verify that it reduces to

$$\ddot{r} = \frac{\ell^2}{m^2 r^3} - \frac{GM}{r^2}. \tag{$\blacklozenge$}$$

[*Hint: for* (*b*)*, you should use* `sp.solve` *instead of* `sp.solveset`*.*]

[*Comment: equation* (◆) *cannot be solved in closed form for* $r(t)$*. However, it is possible to derive an equation for* $r(\theta)$ *from* (◆)*, which is possible to solve analytically; this derived equation is known as the* orbit equation*. The orbit equation hidden in* (◆) *reveals that the orbits of the planets are given by conic sections: circles, ellipses, parabolas, and hyperbolas. This was first understood in 1710 by Johann Bernoulli; Newton included it in the second edition of the* Principia *in 1713, without attribution. Details of the derivation can be found in e.g. section I.2.2 of Hairer, Lubich & Wanner* (2006)[2]*.*]

[2] E. Hairer, C. Lubich, and G. Wanner. *Geometric Numerical Integration*, volume 31 of *Springer Series in Computational Matheematics*. Springer-Verlag, 2006

2. Wolfgang Pauli (1900–1958) was an Austrian physicist. He finished high school in 1918; he wrote his first scientific paper two months later, on Einstein's theory of general relativity, and received his PhD in 1921. He was one of the founders of quantum mechanics, formulating the Pauli exclusion principle and the concept of spin. He worked in Zürich from 1928–1940; the Anschluss of Germany and Austria caused him difficulties in Switzerland, and he fled to the United States. He was awarded the Nobel Prize in Physics in 1933. He was a relentless perfectionist, once famously describing a theory as 'not even wrong'. Among many other contributions, he was the first to successfully describe the hydrogen atom using quantum mechanics; he submitted his paper on the subject ten days before Schrödinger's.

   The wave function for the non-relativistic hydrogen atom in spherical polar coordinates (r, θ, ϕ) is given by

$$\Psi_{n\ell m} = R_{n\ell}(r) Y_\ell^m(\theta, \phi),$$

   where $n = 1, 2, \ldots$ is the principal quantum number (describing the shell the electron occupies), $\ell = 0, 1, \ldots n-1$ is the azimuthal quantum number (describing the magnitude of the angular momentum), and $m = -\ell, \ldots, \ell$ is the magnetic quantum number (describing the projection of the angular momentum onto the z-axis). Here Y_ℓ^m is a spherical harmonic function of degree ℓ and order m, while the radial part of the wave function is given by

$$R_{n\ell}(r) = \sqrt{\left(\frac{2}{na}\right)^3 \frac{(n-\ell-1)!}{2n[(n+\ell)!]}} e^{-r/na} \left(\frac{2r}{na}\right)^\ell \left[L_{n-\ell-1}^{2\ell+1}(2r/na)\right].$$

   Here $L_n^\alpha(r)$ is an associated Laguerre polynomial and a is the Bohr radius (a positive real physical constant whose value is not required).

   Pauli has now asked for your help. He wishes to compute the mean μ and standard deviation σ of the distance of the electron to the

nucleus for a given quantum state (n, ℓ) via

$$\mu_{n\ell} = \int_0^\infty R_{n\ell}^2 r^3 dr \quad \text{and} \quad \sigma_{n\ell} = \sqrt{\left(\int_0^\infty R_{n\ell}^2 r^4 dr\right) - \left(\int_0^\infty R_{n\ell}^2 r^3 dr\right)^2},$$

but he is too busy formulating the Pauli exclusion principle.

(a) Write a program for Pauli that computes

$$f(n, \ell, k) = \int_0^\infty R_{n\ell}^2 r^k dr,$$

for given (i.e. specified, not symbolic) integer values of n, ℓ, k, in terms of a. Verify that for the ground state $(n, \ell) = (1, 0)$, $\mu_{10} = 3a/2$ and $\sigma_{10} = \sqrt{3}a/2$.

(b) Using your code, plot the mean distance μ_{n0} in units of a for $n = 1, \ldots, 8$, with error bars given by the associated standard deviations. On the same plot, plot μ_{n1} with error bars given by the associated standard deviations for $\ell = 1, n = 2, \ldots, 8$. Are the higher-$\ell$ states closer to the nucleus, or further away, on average?

[*Hint: the associated Laguerre polynomials $L_n^\alpha(r)$ are available in sympy as*

`sympy.assoc_laguerre(n, alpha, r)`.]

[*Hint: you can plot data with error bars with*

`matplotlib.pyplot.errorbar`.

You might use the keyword arguments `yerr`, `fmt`, `color`, `ecolor`, `label`, *and* `capsize`.]

[*Comment: the spherical harmonics $Y_\ell^m(\theta, \phi)$ are available in sympy as* `sympy.Ynm(l, m, theta, phi)`, *but you do not need this for the question.*]

12 *Introduction to numerical computing*

What does it mean to solve a mathematical problem?

One understanding is that a problem is solved when we have a closed-form expression for the objects we seek to know. We want a *formula*, derived by pen-and-paper calculations or via a symbolic computing engine. For example, we consider that the problem of finding the roots of a quadratic equation is solved by the quadratic formula. A more advanced example is that the Schrödinger equation of quantum mechanics is exactly solvable for the case of the hydrogen atom, with one electron orbiting one proton: we know the formula for the wave function, and may thereby compute everything we wish to know about the system.

This conception of mathematical solution soon meets its limits, however. We can solve for the roots of a quadratic, cubic, or quartic equation, but no general formula exists for the roots of polynomials of degree five or higher[1]. We can solve for the wave function of the hydrogen atom, but no analytical solution exists for the wave function of the helium atom, or for any other heavier element. In fact, most mathematical problems do not permit solutions with simple formulae.

So must we give up? Must we abandon all hope of computing the roots of polynomials of degree five, or more complicated equations? Will we never understand helium, never mind carbon or iron or uranium? Should we restrict ourselves to that tiny subset of problems where the answer is summarised in a formula? Or should we limit our ambitions, and merely hope to prove that solutions exist, or make other qualitative statements about them?

No! Absolutely not. We can solve all of these problems, and far more beyond, with stunning accuracy and lightning speed. We do this by expanding our conception of what it means to solve a mathematical problem: we seek *algorithms*, not formulae, for computing the desired solutions. A formula says what the solution is; an algorithm says how to compute the solution[2]. An algorithm specifies a sequence of computations to perform for given inputs that yield the desired outputs, but because these computations will typically involve iteration and conditionals, we cannot express usually the outputs of the algorithm

[1] This result is known as the Abel–Ruffini theorem; Ruffini gave an incomplete proof in 1799, with Abel proving it rigorously in 1824.

[2] In other words, formulae are *declarative* while algorithms are *imperative*. For more on this distinction, see

H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2 edition, 1996

as a simple formula of the inputs. We might say that a mathematical problem is solved if an algorithm is known to compute its solutions to arbitrary accuracy with a reasonable amount of effort.

In numerical mathematics, we discover, develop, study, and analyse algorithms for solving mathematical problems. This is a tradition with a glorious history; among its contributors are Archimedes, Heron, Liu Hui, al-Khwārizmī[3], Madhava, al-Kāshī, Newton, Euler, Lagrange, Legendre, Jacobi, Gauss, Adams, Richardson, Turing, von Neumann, Kantorovich, Courant, and Lanczos.

In this chapter, we will study the use of various classical algorithms of numerical analysis to solve mathematical problems arising in mathematics or the sciences. We begin with the `numpy` module[4], which is the foundation of the scientific software ecosystem in Python. Numpy offers a powerful n-dimensional array datatype, allowing for the extremely efficient implementation of operations on vectors, matrices, and tensors[5].

[3] Muḥammad ibn Mūsā al-Khwārizmī (c. 780–847) was a Persian mathematician, astronomer, and geographer who worked at the House of Wisdom in Baghdad. His book *The Compendious Book on Calculation by Completion and Balancing* founded the field of algebra; its Arabic name *al-Jabr* is the root of the word. His textbook *On the Calculation with Hindu Numerals* introduced the Indo–Arabic numeral system and the decimal position system we use today. His name was itself the source for the word *algorithm*.

[4] C. R. Harris et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020

[5] We already met numpy briefly when plotting with `matplotlib`, in chapter 7.

12.1 *Vectors*

Let us start by representing a vector $x \in \mathbb{R}^n$ as a numpy array:

```
(python)  import numpy as np
(python)  a = np.array([1, 10, 100, 1000])
(python)  a
           array([ 1, 10, 100, 1000])
```

This is the first way to make a numpy array, via a `list`. Unlike a list, but like a mathematical vector, each entry in an array *must* have the same type. Numpy automatically infers the type from the given data:

```
(python)  a.dtype
           dtype('int64')
```

Here `'int64'` denotes 64-bit integers, i.e. integers $z \in \mathbb{Z}$ satisfying $-2^{63} \leqslant z \leqslant 2^{63} - 1$[6]. If a mix of types is given, numpy attempts to promote the objects to the broadest type. For example, passing a single `float` in the list coerces all elements to a floating-point datatype:

[6] One bit is used to store the sign. Each integer type also has an *unsigned* friend, which assumes the integer is nonnegative. For example, the `np.uint64` datatype represents integers satisfying $0 \leqslant z \leqslant 2^{64} - 1$.

```
(python)  b = np.array([1, 10, 100, 1000.0])
(python)  b.dtype
           dtype('float64')
(python)  b
```

```
array([ 1., 10., 100., 1000.])
```

You can specify the datatype, if more control is needed[7]:

[7] You can find a list of all numpy datatypes at https://numpy.org/doc/stable/reference/arrays.scalars.html#sized-aliases.

```
(python) a = np.array([1, 10, 100, 1000], dtype=np.int32)
(python) a.dtype
         dtype('int32')
```

Since each entry in an array is the same type, it occupies the same amount of memory. This allows operations on numpy arrays to be much faster than on lists. However, you need to be aware of the data type you are using and ensure that your calculations remain within its bounds. For example, if you created an array with 8-bit integers to save memory with

```
(python) c = np.array([1, 25, 50], dtype=np.int8)
```

then adding 100 would give the wrong result:

```
(python) c + 100
         array([ 101, 125, -106], dtype=int8)
```

Here $50 + 100 = 150 > 127 = 2^7 - 1$, which is the largest number storable in an `np.int8`[8].

[8] As you can see, arithmetic on fixed-size integers is subtle. The value -106 is returned because

$$-106 \equiv 150 \pmod{128}.$$

On the other hand, your code can be much faster if it has to move substantially less data around memory. As with many things in programming, speed and danger are on two sides of the same coin.

You can also make numpy arrays with various convenience functions, like `np.arange` (which makes a numpy array akin to that of `range`), `np.linspace` (which makes an array with a specified number of divisions between the beginning and end), `np.zeros` and `np.ones`:

```
(python) np.arange(1, 12, 3)
         array([ 1, 4, 7, 10])
(python) np.arange(0, 11, 2.5)
         array([ 0., 2.5, 5., 7.5, 10.])
(python) np.linspace(0, 1, 11)
         array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
(python) np.zeros(4)
         array([0., 0., 0., 0.])
(python) np.ones(4)
         array([1., 1., 1., 1.])
```

We see that `np.arange` allows for non-integer steps, unlike `range`.

You can query the dimension of an array with its `ndim` attribute:

```
(python)  a.ndim
           1
```

Here the return value 1 indicates we are dealing with a vector[9]. You can query the shape of the array with its `shape` attribute:

[9] A matrix would return 2.

```
(python)  a.shape
           (4,)
```

This returns a tuple with `len(a.shape) == a.ndim`[10]. You can query the number of bytes required to store the array with its `nbytes` attribute:

[10] A matrix would return a tuple with two entries, the number of rows and columns.

```
(python)  np.arange(1, 5, dtype=np.int8).nbytes
           4
(python)  np.arange(1, 5, dtype=np.int32).nbytes
           16
(python)  np.arange(1, 5, dtype=np.int64).nbytes
           32
```

Like Python lists, numpy arrays fully support indexing and slicing:

```
(python)  a[2]
           100
(python)  a[-1]
           1000
(python)  a[1:3]
           array([ 10, 100])
```

Unlike a list, we can set all elements of a slice with a convenient syntax:

```
(python)  b[1:3] = 0

(python)  b
           array([ 1., 0., 0., 1000.])
```

In other words, taking a slice gives a *view* of the underlying data, so that modifying the slice modifies the underlying data. Attempting the same on a list raises a `TypeError`.

Numpy arrays offer the arithmetical operations you would expect for vectors. For example, we can add them together componentwise:

```
(python)  a + np.arange(1, 5)
           array([ 2, 12, 103, 1004])
```

or multiply them by a scalar:

```
(python)  3*a
          array([  3,  30, 300, 3000], dtype=int32)
```

Recall that for lists, these two operations would concatenate or repeat lists, respectively. We can also add a scalar value:

```
(python)  a + 5
          array([   6,   15,  105, 1005], dtype=int32)
```

Under the hood, if you add two numpy arrays of different shapes, it attempts to *broadcast* one to the shape of the other. In this example, it broadcasts the scalar 5 to a vector of the right length with all entries 5, and adds that. If you try to add two numpy arrays that cannot be broadcast to the same shape, numpy raises a `ValueError`.

Numpy gives a convenient syntax for other componentwise operations. For example, multiplying two arrays with `*` does so componentwise[11]:

[11] This is sometimes known as the *Hadamard product*.

```
(python)  a * np.arange(1, 5)
          array([   1,   20,  300, 4000])
```

Similarly, division between arrays or exponentiating an array is understood componentwise:

```
(python)  np.arange(1, 5) / a
          array([1.  , 0.2 , 0.03 , 0.004])
(python)  np.arange(1, 5)**2
          array([ 1,  4,  9, 16])
```

If you wished, you could exponentiate one array with exponents from another.

Inequalities are also understood componentwise:

```
(python)  a >= 50
          array([False, False,  True,  True])
```

These Boolean arrays can be used to select entries of a given array satisfying some criterion. You can use this to extract the entries of `a` that are greater than or equal to 50:

```
(python)  a[a >= 50]
          array([  100, 1000], dtype=int32)
```

or to extract the entries divisible by 4:

```
(python)  a[a % 4 == 0]
          array([  100, 1000], dtype=int32)
```

As we saw previously, numpy also includes every operation from

the `math` module, with the twist that they act componentwise[12]:

[12] In numpy terminology, these are called *universal* functions. You can create your own with `np.vectorize`.

```
(python) np.sin(a)
          array([ 0.84147098, -0.54402111, -0.50636564, 0.82687954])
(python) np.log10(a)
          array([0., 1., 2., 3.])
```

Using these functions is *much* faster than applying the function from `math` with a **`for`** loop. This is because with the numpy function the loop is in a compiled language that avoids all the overhead of Python's interpretation and safety checks. In general, when coding with numpy, we want to use as few **`for`** loops as possible; write your code so that all loops are done inside numpy.

Exercise 12.1. Let `x` and `y` be two numpy arrays representing the x- and y- coordinates of a discretely sampled function in one dimension. Say that x_i is *near a root* if the sign of y_i is different to the sign of y_{i+1}[13]. Write one line of Python to identify the entries of `x` that are near a root.

[13] In other words, (x_i, x_{i+1}) form an interval suitable for applying the bisection rootfinding algorithm of code block 4.14.

12.2 *Matrices*

To represent a matrix A[14], you can construct a `np.array` with a list of lists specifying its rows:

[14] As with vectors, the built-in datatypes represent (subsets of) the ring $\mathbb{Z}$ and the fields $\mathbb{R}$ and $\mathbb{C}$. In mathematics, we often wish to represent matrices over finite fields; you can do so with the `galois` package, which is an add-on for numpy.

```
(python) A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
(python) A
          array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])
```

Matrices are also the same `np.array` class, and have the same methods like `ndim`, `shape`, `nbytes` etc.:

```
(python) A.ndim
          2
(python) A.shape
          (3, 3)
(python) A.nbytes
          72
```

The total number of entries in a matrix can be queried with the `size` attribute:

```
(python) A.size
          9
```

Matrices can be indexed along each axis independently:

```
(python) A[0, :] # first row, all cols
          array([1, 2, 3])
(python) A[1, :] # second row, all cols
          array([4, 5, 6])
(python) A[:, 0] # first col, all rows
          array([1, 4, 7])
(python) A[1, 1] # second row, second col
          5
```

Matrices can also be sliced along each axis independently:

```
(python) A[1:, :-1] # from second row on, until last column
          array([[4, 5],
                 [7, 8]])
```

A numpy array can be reshaped to interpret it in different ways. For example, to create a 10×10 matrix with the numbers $1, \ldots, 100$, we can start with the vector of these numbers, and then reshape it:

```
(python) B = np.arange(1, 101).reshape(10, 10)
(python) B
          array([[  1,   2,   3,   4,   5,   6,   7,   8,   9,  10],
                 [ 11,  12,  13,  14,  15,  16,  17,  18,  19,  20],
                 [ 21,  22,  23,  24,  25,  26,  27,  28,  29,  30],
                 [ 31,  32,  33,  34,  35,  36,  37,  38,  39,  40],
                 [ 41,  42,  43,  44,  45,  46,  47,  48,  49,  50],
                 [ 51,  52,  53,  54,  55,  56,  57,  58,  59,  60],
                 [ 61,  62,  63,  64,  65,  66,  67,  68,  69,  70],
                 [ 71,  72,  73,  74,  75,  76,  77,  78,  79,  80],
                 [ 81,  82,  83,  84,  85,  86,  87,  88,  89,  90],
                 [ 91,  92,  93,  94,  95,  96,  97,  98,  99, 100]])
```

As with slicing, this reshaping creates a *view* of the underlying data; changing the reshaped object changes the underlying one. This can be very useful for accessing certain entries of an array with slicing. To make a copy, use the `np.copy` function.

You can apply various aggregating functions along a given axis. For example, to sum along the rows of `B` (i.e. compute the column sums), we can do

```
(python)  B.sum(axis=0)
           array([460, 470, 480, 490, 500, 510, 520, 530, 540, 550])
```

Other aggregating functions available include `np.mean`, `np.max`, and `np.min`:

```
(python)  B.mean(axis=1) # average each row
           array([ 5.5, 15.5, 25.5, 35.5, 45.5, 55.5, 65.5, 75.5, 85.5, 95.5])
(python)  B.min(axis=0) # min of each column
           array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

As with one-dimensional arrays, the usual arithmetical operations are understood componentwise:

```
(python)  A + 1
           array([[ 2,  3,  4],
                  [ 5,  6,  7],
                  [ 8,  9, 10]])
```

To make the identity matrix, use the `np.eye` function:

```
(python)  I = np.eye(3)
(python)  I
           array([[1., 0., 0.],
                  [0., 1., 0.],
                  [0., 0., 1.]])
```

One of the main operations we do with matrices is matrix-vector multiplication. In Python, this is denoted with the `@` symbol:

```
(python)  A @ np.arange(1, 4)
           array([14, 32, 50])
```

This symbol also denotes matrix-matrix multiplication:

```
(python)  (A @ I == A).all()
           True
(python)  A @ A
           array([[ 30,  36,  42],
                  [ 66,  81,  96],
                  [102, 126, 150]])
```

When applied between two vectors, `@` denotes the dot product[15]:

```
(python)  np.arange(3) @ np.arange(1, 4)
           8
```

[15] All of these uses are consistent: `@` really represents *tensor contraction* over the last index of the first argument and the first index of the second argument.

Finally, to transpose a matrix, use its `.T` attribute:

(python)
```
A.T
 array([[1, 4, 7],
        [2, 5, 8],
        [3, 6, 9]])
```

Similarly, the `.H` attribute of a matrix A returns its conjugate transpose A^* (which is usually what is required if dealing with complex-valued matrices).

Exercise 12.2. The Pauli matrices

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},$$

form a basis for the Hermitian 2×2 matrices, and hence for complex observable quantities in quantum mechanics. They occur in the Pauli equation which accounts for the interaction of the spin of a particle with an external electromagnetic field.

Verify that each Pauli matrix is Hermitian ($\sigma = \sigma^*$), unitary ($\sigma^{-1} = \sigma^*$), and hence involutary ($\sigma = \sigma^{-1}$).

12.3 *Numerical linear algebra*

Numpy offers the fundamental data structure for scientific computing. Its sister package, scipy, offers many algorithms for scientific and technical computing[16]. To install it, at the terminal type

(terminal)
```
pip install scipy
```

Scipy offers algorithms for optimisation, linear algebra, integration, interpolation, Fourier transforms, ODE solvers, and much more besides. We will meet some of this functionality as we go on.

First let us study the use of scipy for numerical linear algebra. Scipy puts its algorithms for linear algebra in a submodule, `linalg`. For example, to solve a linear system $Ax = b$, use `sc.linalg.solve`:

(python)
```
A = np.array([[3, 4, 1], [-1, 8, 6], [5, 6, 7]])
```

(python)
```
b = np.arange(3)
```

(python)
```
import scipy as sc
```

(python)
```
x = sc.linalg.solve(A, b)
```

[16] P. Virtanen et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020

```
(python)  x
          array([ 0.0617284 , -0.13580247,  0.35802469])
```

There are special routines like `sc.linalg.solve_triangular` or `sc.linalg.solve_toeplitz` if you know that your matrix has the relevant structure[17].

You can also compute the inverse of a matrix with `sc.linalg.inv`[18]:

```
(python)  sc.linalg.inv(A)
          array([[ 0.12345679, -0.13580247,  0.09876543],
                 [ 0.22839506,  0.09876543, -0.11728395],
                 [-0.28395062,  0.01234568,  0.17283951]])
```

Determinants are a major topic of Prelims *Linear Algebra II* in Hilary term. They are computed with[19]:

```
(python)  sc.linalg.det(A)
          162.0
```

To compute the eigenvalues of a matrix, use `sc.linalg.eigvals`:

```
(python)  sc.linalg.eigvals(A)
          array([ 1.83954294+2.81570752j,  1.83954294-2.81570752j, 14.32091412+0.j])
```

If you want the eigenvectors too, use `sc.linalg.eig`. You can control whether it computes the left eigenvectors, right eigenvectors, or both by passing the optional arguments `left=False` and `right=True`.

When working with matrices, different matrix factorisations are crucial[20]. Scipy can compute all of the major factorisations, such as the LU factorisation (for solving linear systems), the QR factorisation (for solving eigenvalue and least-squares problems), and the singular value decomposition (for computing pseudoinverses, determining range and nullspace, low-rank approximations, and many other purposes):

```
(python)  sc.linalg.lu(A) # LU decomposition
(python)  sc.linalg.qr(A) # QR decomposition
(python)  sc.linalg.svd(A) # Singular values
```

The LU factorisation factors a matrix A into

$$A = LU$$

where L is lower-triangular and U is upper-triangular. This reduces

[17] A *Toeplitz matrix* is one where the entries are constant along diagonals.

[18] Computing the inverse of a matrix is very rarely the right thing to do. If you have to solve a sequence of linear systems with the same matrix, it is much faster to compute an appropriate matrix factorisation. This is what Gaussian elimination does (it computes the so-called LU factorisation), but as far as I can work out no one tells you this until second year *A7: Numerical Analysis*.

[19] Again, computing determinants is very rarely the right thing to do in practice. The computational cost of the naïve formula scales like $n!$ for an $n \times n$ matrix. To improve on this, one first computes an LU factorisation, then calculates the determinant of that. The cost of this approach scales like n^3 for an $n \times n$ matrix.

[20] You will learn more if you take the Part A option *A7: Numerical Analysis* or the Part C option *C6.1 Numerical Linear Algebra*.

Alan Turing, 1912–1954.

the solution of $Ax = b$ to two solutions with triangular matrices, each of which may be done efficiently using substitution. The LU factorisation is produced by recording the calculations involved in Gaussian elimination; this view of things was introduced by Turing in a landmark 1948 paper[21].

[21] A. M. Turing. Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1(1):287–308, 1948

The QR factorisation factors a matrix A into

$$A = QR$$

where Q is unitary (i.e. $Q^{-1} = Q^*$) and R is upper-triangular. The QR factorisation is used in solving linear least-squares problems; it converts the optimisation problem into an upper-triangular solve using R, which can be done quickly using substitution. The QR factorisation formalises the process of Gram–Schmidt orthogonalisation procedure you encounter in Prelims *Linear Algebra II*[22]. The QR factorisation also lies at the heart of the so-called QR algorithm, the standard algorithm for computing eigenvalue decompositions of matrices, introduced in a crucial 1961 paper by Francis[23].

[22] Gram–Schmidt is not *numerically stable*, so QR factorisations are in fact computed in other ways, e.g. using Householder reflections.

[23] J. G. F. Francis. The QR Transformation: A Unitary Analogue to the LR Transformation—Part 1. *The Computer Journal*, 4(3):265–271, 1961

The singular value decomposition (SVD) factors a matrix A into

$$A = U\Sigma V^*,$$

where U and V are unitary, and Σ is diagonal. The SVD is perhaps the most fundamental and important of all matrix decompositions: it reveals the four fundamental subspaces associated with a matrix[24], provides the optimal low-rank approximations of a matrix, and allows for the straightforward computation of its Moore–Penrose pseudoinverse[25]. It is of central importance in statistics and data science. It was independently discovered around the same time by several mathematicians, among them Oxford's Savilian Professor of Geometry, James Joseph Sylvester[26], who also introduced the term 'matrix'.

[24] G. S. Strang. The fundamental theorem of linear algebra. *The American Mathematical Monthly*, 100(9):848–855, 1993

[25] Oxford's Roger Penrose invented this independently while an undergraduate student; see

R. Penrose. A generalized inverse for matrices. *Mathematical Proceedings of the Cambridge Philosophical Society*, 51(3):406–413, 1955

[26] J. J. Sylvester. A new proof that a general quadric may be reduced to its canonical form (that is, a linear function of squares) by means of a real orthogonal substitution. *Messenger of Mathematics*, 19:1–5, 1889

Exercise 12.3. The matrix

$$F = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

can be used to produce the Fibonacci sequence by repeated multiplication: the top-left element of F^n is the $(n+1)^{\text{th}}$ Fibonacci number.

Use the eigenvalue decomposition of F to efficiently calculate the 1100^{th} Fibonacci number.

James Joseph Sylvester, 1814–1897.

12.4 *Approximating integrals*

The method you are taught in school for evaluating a definite integral

$$I[f,a,b] = \int_a^b f(x)\,\mathrm{d}x$$

involves identifying an antiderivative $F(x)$ such that $F' = f$, and applying the fundamental theorem of calculus to express

$$I[f,a,b] = F(b) - F(a).$$

In 1833, Liouville proved that this strategy is severely limited: the antiderivatives of elementary functions[27] cannot generally be expressed as elementary functions[28]. Among the many examples, the antiderivatives of all of the following functions are non-elementary: $\sqrt{1-x^4}, 1/\log x, \exp -x^2, \sin x^2, \sin x/x, \exp -x/x, \exp\exp x$.

Instead, one approximates $I[f,a,b]$ numerically, using a *quadrature* scheme[29]. This is an ancient idea; Babylonian mathematicians calculated the position of Jupiter by (in modern words) numerically integrating the time-velocity graph. The basic structure of a quadrature scheme is to approximate the integral by a weighted sum

$$I[f,a,b] \approx \sum_{i=1}^{n} w_i f(x_i)$$

Different quadrature schemes differ in their choice of the n *quadrature weights* w_i and *quadrature nodes* x_i. The most prominent family of schemes was proposed by Carl Friedrich Gauss in 1815[30]: by a wonderfully clever choice, a scheme with n nodes can exactly evaluate integrals of polynomials of degree $2n-1$ or less. This is far from the end of the story of quadrature—further topics include multidimensional integrals, integrals on unbounded domains, integrals of highly oscillatory functions, and singular integrals, error estimation, and adaptive quadrature schemes.

To compactly specify our integrands, we now introduce the final Python keyword of this course: the **`lambda`** statement. This is an equivalent way to define a function, but without giving it a name (a so-called *anonymous* function). For example, the code

(python)
```
square = lambda x: x*x
```

is completely equivalent to the function definition using the **`def`** statement in code block 7.7. With a **`lambda`** statement, the expression on the right is automatically returned by the function defined.

Scipy interfaces the QUADPACK package for automatic integration[31]. It can be used as follows:

[27] These are constants, polynomials, exponentials and logarithms, trigonometric functions, hyperbolic functions, and their compositions.

[28] J. Liouville. Premier mémoire sur la détermination des intégrales dont la valeur est algébrique. *Journal de l'École Polytechnique*, XIV:124–148, 1833

[29] This name comes from the geometric interpretation: one is attempting to find a square with the same area as the given integral.

[30] C. F. Gauss. *Methodvs nova integralivm valores per approximationem inveniendi*. Dieterich, 1815

[31] R. Piessens, E. de Doncker-Kapenga, C. W. Überhuber, and D. K. Kahaner. *QUADPACK: a subroutine package for automatic integration*, volume 1 of *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, 1983

```
(python)  sc.integrate.quad(lambda x: np.exp(-x), 0, np.inf)
           (1.0000000000000002, 5.842606742906004e-11)
```

The routine returns a tuple; the first element is the estimate of $I[f,a,b]$, while the second is an estimate for the quadrature error. While the functions listed above do not have elementary antiderivatives, scipy integrates them handily:

```
(python)  sc.integrate.quad(lambda x: np.sqrt(1-x**4), 0, 1)
           (0.8740191847640391, 3.733313658216275e-10)
(python)  sc.integrate.quad(lambda x: 1/np.log(x), 10, 20)
           (3.7397004728455228, 4.151901570154684e-14)
(python)  sc.integrate.quad(lambda x: np.sin(x)/x, 0, np.pi)
           (1.851937051982466, 2.0560631552673694e-14)
```

and so on. If we attempt to integrate the last function $\sin x/x$ over an interval spanning 0, however, QUADPACK fails:

```
(python)  sc.integrate.quad(lambda x: np.sin(x)/x, -np.pi, np.pi)
           (nan, nan)
```

with a `RuntimeWarning` raised to let us know something has gone wrong. The problem here is that the integrand has a removable singularity at $x = 0$. We can tell QUADPACK to break the integral apart at this point with

```
(python)  sc.integrate.quad(lambda x: np.sin(x)/x, -np.pi, np.pi, points=[0])
           (3.703874103964933, 4.1121263105347394e-14)
```

which then succeeds.

Scipy also offers routines for double and triple integrals, as `sc.integrate.dblquad` and `sc.integrate.tplquad` respectively. Higher dimensional integrals are computed with the `sc.integrate.nquad` routine. We will not discuss these further.

Exercise 12.4. There is no closed-form formula for the perimeter of an ellipse. Given an ellipse of semi-major axis $a > 0$ and eccentricity $e \in [0,1]$, the perimeter is given by

$$L = a \int_0^{2\pi} \sqrt{1 - e^2 \sin^2\phi}\, \mathrm{d}\phi.$$

Compute the distance travelled by the Earth in one orbit, using the values $a = 149598261$km and $e = 0.01671123$. What is the error incurred in the distance travelled if one instead approximates the Earth's motion around the sun with a circle of radius $r = 149597870.7$km? Express the error as a percentage of the elliptical value for the distance.

12.5 *Least squares and curve-fitting*

A very common task in scientific computing is to fit the parameters of a model to experimental data. This is done with least-squares, as invented by Gauss for the determination of the orbit of Ceres.

First, let us consider fitting polynomials to given data. This may be done with numpy's `np.polynomial.Polynomial.fit` function[32]. Given data $\{x_i\}_{i=1}^n$, $\{y_i\}_{i=1}^n$, this function finds the polynomial $p(x)$ in the vector space Π_k of polynomials of degree k such that[33]

$$p = \operatorname*{arg\,min}_{q \in \Pi_k} \sum_{i=1}^{n} |q(x_i) - y_i|^2.$$

[32] Numpy offers powerful facilities for dealing with polynomials in its `np.polynomial` module, including numerically stable representations, polynomial algebra, root-finding, etc. We will only see a little of this here.

[33] Here, the arg min means 'find the argument $q(x) \in \Pi_k$ that minimises the expression'.

Consider the task of an astronaut on Mars wishing to estimate the acceleration due to gravity on a potential site for a base. One way to estimate this is to drop a weight and measure the distance $d(t)$ fallen as a function of time $t > 0$, then fit the data to the polynomial

$$d(t) = d_0 + v_0 t + \frac{1}{2} g t^2.$$

This problem is solved in code block 12.1.

Code block 12.1. Fitting a quadratic polynomial to data.

```
import numpy as np

# Recorded data
t = np.linspace(0.1, 1, 10)
d = np.array([0.03, 0.062, 0.169, 0.31, 0.468,
              0.67, 0.903, 1.183, 1.516, 1.869])

# Fit a quadratic polynomial
# The .convert() method is called to express the
# polynomial in the familiar monomial basis.
fit = np.polynomial.Polynomial.fit(t, d, deg=2).convert()

# Extract coefficients
(d0, v0, ghalf) = fit.coef
g = ghalf * 2

print(f"Fitted data: d(t) = {d0} + {v0}*t + 1/2 * {g} * t**2")
```

This returns the estimate $g \approx 3.79$ m s^{-2}, reasonably close to the average Martian acceleration due to gravity of 3.72076 m s^{-2} given the limited precision of the data.

This can also be used to estimate parameters in exponential relationships, by taking logarithms. The number of particles $N(t)$ of an isotope undergoing radioactive decay satisfies

$$N(t) = N_0 \exp -t/\tau,$$

where N_0 is the initial number of particles and τ is related to the half-life $t_{1/2}$ via $t_{1/2} = \tau \log 2$. Taking logarithms, we find

$$\log N = \log N_0 - \frac{t}{\tau}.$$

If we fit a linear polynomial between t and $\log N$, the slope estimates the reciprocal of τ, and hence $t_{1/2}$.

Consider code block 12.2, which estimates the half-life of the ${}^{233}_{97}\mathrm{Bk}$ isotope of Berkelium[34]. The code estimates the half-life as 20.7 s, close to the best estimated value of 21 s.

Exercise 12.5. Modify code blocks 12.1 and 12.2 to plot the data and the fitted polynomial.

It is also possible to fit parameters of functions other than polynomials, using scipy's `sc.optimize.curve_fit` function. This function takes in a function that maps inputs to outputs, the known inputs, the known outputs, and initial guesses for the unknown parameters. These initial guesses are crucial; the minimisation problems solved in the least-squares fitting may no longer be convex, and there may be multiple minimisers. The choice of initial guess will determine both which minimiser will be found, and how quickly it will be found[35]. A good deal of cleverness goes into devising reasonable initial guesses; when Gauss applied least squares to identify the orbit of Ceres, he first found the unique ellipse passing through a minimally determining subset of the observations, then used that as initial guess for fitting to the full data.

As an example, consider the task of fitting an ellipse with axes parallel to the x- and y- axes to given data $\{\theta_i, r_i\}_{i=1}^n$. The formula for the radius r of such an ellipse as a function of angle θ is given by

$$r(\theta; a, b) = \frac{ab}{\sqrt{a^2 \sin^2\theta + b^2 \cos^2\theta}},$$

where $a > b > 0$ are the semi-major axis and semi-minor axis lengths, respectively. This is presented in code block 12.3. We generate synthetic data for $\theta \in [0, \pi/2]$ for the 'true' parameter values $(a, b) = (3, 1)$ and add normally-distributed random noise to model observation errors. We then initialise the curve fit with an initial guess $(a_0, b_0) = (3/2, 3/2)$, i.e. a circle of radius $3/2$. The curve fit successfully

[34] George Berkeley (1685–1753) was an Irish philosopher, bishop, slaveholder, and mathematician. He rejected the Newtonian idea of absolute space and time; Karl Popper described him a precursor of Mach and Einstein's relativity. In 1734, he launched a blistering attack on the shaky foundations of calculus in his book *The Analyst*; this had a major influence on the development of analysis, as mathematicians such as Bayes, Maclaurin, Cauchy, Riemann and Weierstrass strove to place calculus on a rigorous foundation. The city of Berkeley in California and its university are named after him; in 2023 Trinity College Dublin announced they would remove his name from their main library, because of his advocacy of slavery. He is buried in the cathedral of Christ Church, Oxford.

George Berkeley, 1685–1753

[35] You can learn more in Part C *C6.2 Continuous Optimisation*.

Code block 12.2. Estimating the half-life of $^{233}_{97}$Bk.

```
import numpy as np

# Recorded data: times in seconds
t = np.array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.,10.,11.,12.,13.,14.,
     15.,16.,17.,18.,19.,20.,21.,22.,23.,24.,25.,26.,27.,28.,29.,30.,31.,
     32.,33.,34.,35.,36.,37.,38.,39.,40.,41.,42.,43.,44.,45.,46.,47.,48.,
     49.,50.])

# Recorded data: number of particles
N = np.array([100000, 96597, 93400, 90396, 87325, 84345, 81515, 78746,
          76133, 73610, 71221, 68934, 66718, 64502, 62332, 60264, 58326,
          56448, 54618, 52860, 51142, 49498, 47902, 46277, 44690, 43224,
          41735, 40309, 38980, 37615, 36441, 35294, 34046, 33039, 31926,
          30891, 29866, 28907, 27956, 27039, 26173, 25344, 24486, 23685,
          22898, 22100, 21364, 20649, 19992, 19368, 18722])

logN = np.log(N)

fit = np.polynomial.Polynomial.fit(t, logN, deg=1).convert()
(c, m) = fit.coef
print(c, m)

tau = -1/m
t_half = tau * np.log(2)
print(f"Estimated half life of Berkelium-233: {t_half:.3f}")
```

returns parameters $(a^\star, b^\star) \approx (3, 1)$[36]. The code prints out the optimised parameters, the standard deviation returned from the curve fit, and plots the data, initial guess, and fitted ellipse. On a sample run, the code printed

[36] The exact values returned will depend on the random observation errors added.

```
Fitted parameters: [2.9930133  1.00044137] ± [0.0046581  0.00168654]
```

and renders an image like figure 12.1.

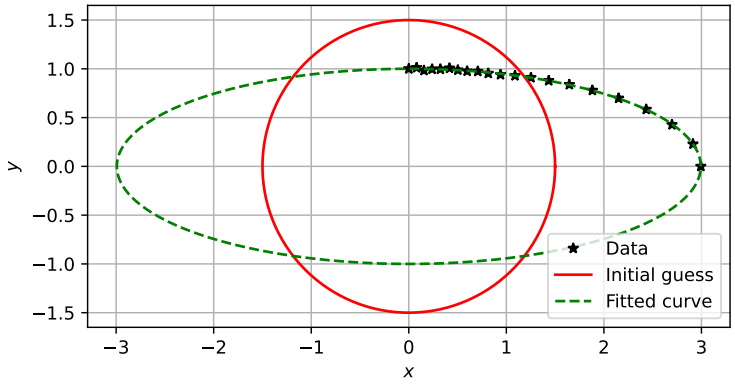


Figure 12.1: The data, initial guess, and fitted ellipse of code block 12.3.

Code block 12.3. Fitting an axis-aligned ellipse to partial noisy data.

```
import numpy as np
import scipy as sc
import matplotlib.pyplot as plt

# Equation for an ellipse, radius as a function of angle.
def f(theta, a, b):
    r = a*b / np.sqrt(a**2 * np.sin(theta)**2 + b**2 * np.cos(theta)**2)
    return r

# Make synthetic noisy data.
# We hide the true parameters in the scope of a function so that
# we cannot accidentally access them in our fitting code.
def make_data():
    a = 3  # semi-major axis
    b = 1  # semi-minor axis

    theta = np.linspace(0, 0.5*np.pi, 21)
    r = f(theta, a, b)
    r = r + np.random.randn(len(theta))*0.01  # noise
    return (theta, r)

(theta, r) = make_data()
plt.grid()
plt.plot(r*np.cos(theta), r*np.sin(theta), '*k', label="Data")
plt.gca().set_aspect('equal')  # make ellipses look like ellipses

# Specify and plot initial guess:
# a circle of radius 1.5
(a0, b0) = (1.5, 1.5)
sample = np.linspace(0, 2*np.pi, 1001)
guess = f(sample, a0, b0)
plt.plot(guess*np.cos(sample), guess*np.sin(sample), '-r', label="Guess")

# Now do the fit
(popt, pcov) = sc.optimize.curve_fit(f, theta, r, (a0, b0))
print(f"Fitted parameters: {popt} ± {np.sqrt(np.diag(pcov))}")

# Plot fitted curve
fit = f(sample, *popt)
plt.plot(fit*np.cos(sample), fit*np.sin(sample), '--g', label="Fit")

plt.legend(loc="lower right")
plt.xlabel(r"$x$")
plt.ylabel(r"$y$")
plt.show()
```

12.6 *Solving differential equation initial value problems*

Finally, we consider how scipy may be used to solve initial value problems associated with systems of ordinary differential equations, i.e. systems of the form

$$\frac{\mathrm{d}y_1}{\mathrm{d}t} = f_1(t, y_1, \ldots, y_n), \tag{12.6.1a}$$
$$\frac{\mathrm{d}y_2}{\mathrm{d}t} = f_2(t, y_1, \ldots, y_n), \tag{12.6.1b}$$
$$\cdots$$
$$\frac{\mathrm{d}y_n}{\mathrm{d}t} = f_n(t, y_1, \ldots, y_n), \tag{12.6.1c}$$

subject to the initial data

$$y_1(t_0) = d_1, \quad y_2(t_0) = d_1, \cdots, \quad y_n(t_0) = d_n. \tag{12.6.2}$$

Equations (12.6.1)–(12.6.2) are usually summarised in vector notation as

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}), \tag{12.6.3a}$$
$$\mathbf{y}(t_0) = \mathbf{y}_0. \tag{12.6.3b}$$

The restriction to systems of first-order equations is not burdensome, because any system of higher-order equations can be rewritten in this form.

Scipy computes numerical approximations to the solution of (12.6.3) with the `sc.integrate.solve_ivp` function. This function takes in three mandatory arguments: a function `f(t, y)` representing (12.6.3a), a tuple containing the initial and final times of the simulation, and the data for the initial condition (12.6.3b). The `solve_ivp` function returns an object `soln` with several attributes of interest, including `soln.success` (a Boolean indicating success or failure), `soln.t` (the time points used for the numerical integration), and `soln.y` (the values of the solution at those time points). It also has many optional arguments for controlling the numerical solution procedure; we will meet these as they arise.

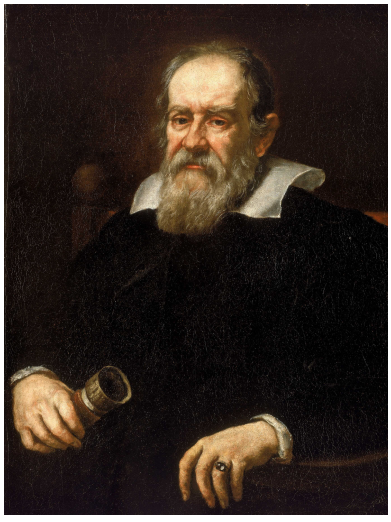
As an example, consider the equation for a simple pendulum[37],

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{l}\theta, \quad \theta(0) = \theta_0, \quad \dot{\theta}(0) = 0, \tag{12.6.4}$$

where g is the acceleration due to gravity and l is the pendulum length. The analytical solution of this problem is

$$\theta(t) = \theta_0 \cos\left(\sqrt{\frac{g}{l}}t\right). \tag{12.6.5}$$

[37] Galileo famously realised that the simple pendulum was isochronous (the period of a pendulum of a given length is constant) while observing a lamp at the centre of the nave of the cathedral in Pisa.



Galileo Galilei, 1564–1642

To cast this into the form (12.6.3), we set $\theta = \theta_1$ and introduce a new variable θ_2:

$$\dot{\theta_1} = \theta_2, \tag{12.6.6a}$$

$$\dot{\theta_2} = -\frac{g}{l}\theta_1. \tag{12.6.6b}$$

The code comparing the numerical solution and the analytical solution is given in code block 12.4.

Code block 12.4. Numerical solution of the simple pendulum.

```
import numpy as np
import scipy as sc
import matplotlib.pyplot as plt

g = 9.81        # acceleration due to gravity
l = 1           # length of the pendulum
ic = [0.1, 0]   # initial conditions
ival = (0, 20) # interval of interest

def f(t, theta):
    return [theta[1], -g/l * theta[0]]

soln = sc.integrate.solve_ivp(f, (0, 20), ic, dense_output=True)

# Get time grid for plotting
t = np.linspace(*ival, 1001)
# Evaluate numerical and analytical solution of IVP
ntheta = soln.sol(t)[0]
atheta = ic[0] * np.cos(np.sqrt(g/l) * t)

plt.grid()
plt.plot(t, ntheta, color='k', label="Numerical solution")
plt.plot(t, atheta, color='b', label="Analytical solution")
plt.xlabel(r"$t$")
plt.ylabel(r"$\theta(t)$")
plt.legend()
plt.show()
```

The two solution approaches are indistinguishable to the eye; the figure the code produces is given in figure 12.2. The `dense_output=True` argument to `sc.integrate.solve_ivp` is necessary to create the `soln.sol` function: this is a callable that evaluates (in a vectorised way) the numerical solution at requested times, by interpolation of

the computed trajectory. Alternatively, one can pass the argument `t_eval` to specify an array of times at which to evaluate the solution.

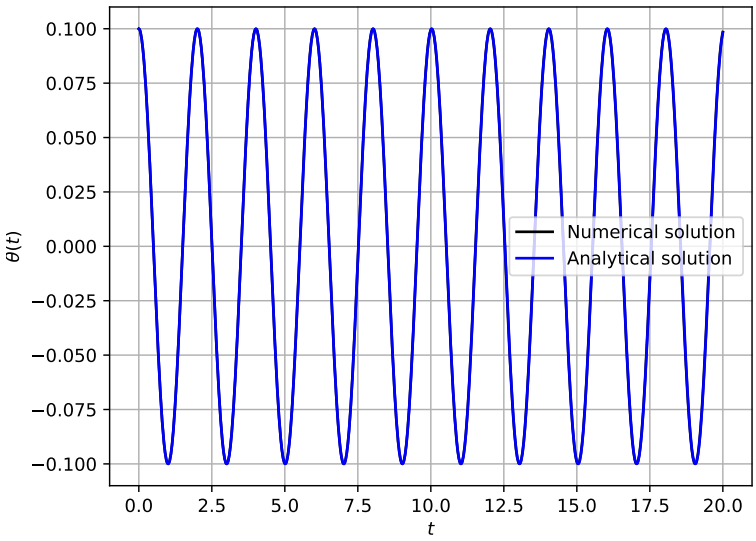


Figure 12.2: Comparing the numerical and analytical solutions of the simple pendulum (12.6.4). The two solutions lie on top of each other. This figure is rendered by code block 12.4.

One convenient feature of the numerical solution is that we can code *events* for the solver to track. These are functions $h(t, y)$ of the state; the solver monitors these functions, and calculates very accurately the times at which each function $h(t, y) = 0$ (as well as the state at these times)[38]. This allows us to monitor properties of the solution, calculate diagnostics, etc. Briefly, one passes a tuple of these monitor functions as the `events` optional argument. The output trajectory object then has the attributes `soln.t_events` (for the times at which the monitor functions were zero) and `soln.y_events` (for the associated states). For example, suppose we wished to monitor when the pendulum is vertical (so that $\theta = 0$). We might code an event as in code block 12.5.

[38] The solver does so using Brent's root-finding algorithm, as implemented in `sc.optimize.brentq`. Brent's algorithm involves bisection, which we met briefly in chapter 4.

R. P. Brent. *Algorithms for Minimisation without Derivatives*. Prentice Hall, 1973

Code block 12.5. Example of event detection.

```
import scipy as sc

g = 9.81        # acceleration due to gravity
l = 1           # length of the pendulum
ic = [0.1, 0]   # initial conditions
ival = (0, 20) # interval of interest

def f(t, theta):
    return [theta[1], -g/l * theta[0]]

def h(t, theta):
    return theta[0]

soln = sc.integrate.solve_ivp(f, (0, 20), ic, events=(h,))

print("Times at which the pendulum is vertical: ")
print(soln.t_events[0])  # Events of the first functional, i.e. h
```

This prints

```
Times at which the pendulum is vertical:
[ 0.50146316  1.50422274  2.50690771  3.50962485  4.51235431  5.51511827
  6.51781387  7.52052444  8.52324324  9.52600688 10.52872046 11.53142346
 12.53413417 13.5368929  14.53962698 15.54232196 16.54503088 17.54778287
 18.55053026 19.55321358]
```

As with integrals, most differential equations cannot be solved in closed form in terms of elementary functions. The simple pendulum equation (12.6.4) is derived from the more accurate model

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{l}\sin\theta, \quad \theta(0) = \theta_0, \quad \dot{\theta}(0) = 0, \tag{12.6.7}$$

under the *small-angle* assumption $\theta \ll 1$, so that $\sin\theta \approx \theta$. Solving (12.6.7) is necessary if this assumption does not hold, but the equation does not have a closed-form solution. The small-angle assumption is often made in introductory texts in order to make the hand-calculations tractable, whether it is physically justified or not[39]. Let us solve (12.6.7) numerically for $\theta_0 = 150^\circ$, and compare it to the analytical solution of (12.6.4). We make the very minor modifications to code block 12.4 in code block 12.6.

As seen in figure 12.3, the results are strikingly different[40]. Computational mathematics gives us the power to understand and solve mathematical problems that we otherwise could not.

[39] This is a common pattern, and not something often emphasised. In many of the problems you will solve on problem sheets or exams, the questions have been carefully and delicately rigged so that your calculations on paper are tractable. Perturb the problem a little, or undo one assumption, and this property will no longer hold.

[40] For a graphical exploration of the nonlinear pendulum, see Chapter 9 of

L. N. Trefethen, Á. Birkisson, and T. A. Driscoll. *Exploring ODEs*. Society for Industrial & Applied Mathematics, 2018

Code block 12.6. Numerical solution of the simple pendulum.

```
import numpy as np
import scipy as sc
import matplotlib.pyplot as plt

g = 9.81              # acceleration due to gravity
l = 1                 # length of the pendulum
ic = [5*np.pi/6, 0]   # initial conditions
ival = (0, 20)        # interval of interest

def f(t, theta):
    return [theta[1], -g/l * np.sin(theta[0])]

soln = sc.integrate.solve_ivp(f, (0, 20), ic, dense_output=True)

# Get time grid for plotting
t = np.linspace(*ival, 1001)
# Evaluate numerical solution, and analytical solution of linearised problem
ntheta = soln.sol(t)[0]
atheta = ic[0] * np.cos(np.sqrt(g/l) * t)

plt.grid()
plt.plot(t, ntheta, color='k', label="Nonlinear solution")
plt.plot(t, atheta, color='b', label="Linearised solution")
plt.xlabel(r"$t$")
plt.ylabel(r"$\theta(t)$")
plt.legend()
plt.show()
```

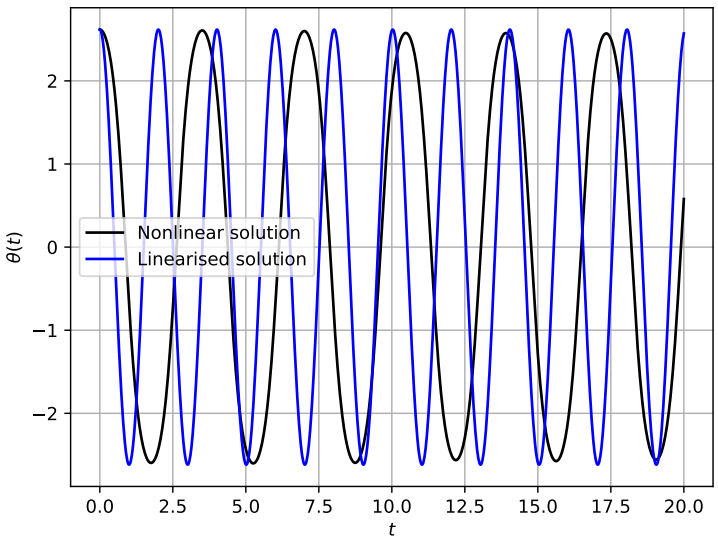


Figure 12.3: Comparing the solutions of the nonlinear and linearised pendulum, (12.6.7) and (12.6.4), for $\theta_0 = 5\pi/6$.

13 Coda: simulating the solar system

In 1901, sponge divers off the coast of the small Greek island of Antikythera discovered the shipwreck of a Roman-era cargo vessel that sank in approximately 70 BC. The shipwreck contained treasures such as marble statues, vases, coins, jewellery, as well as a mysterious hunk of corroded metal to which no one paid much attention. The following year, however, the archaeologist Valerios Stais discovered that the hunk of metal contained a gear—moreover, a precision gear, with teeth about a millimetre long. In fact, the hunk of metal—now called the *Antikythera Mechanism*—was an analogue computer, the first computer known to history. Driven by a hand crank, it used an intricate mechanism of bronze gears and cogs to predict the positions of the Sun, moon, eclipses, and the known planets, according to the astronomical theories then available in Greece. Its discovery forced us to rethink the entire history of technology; it was far more sophisticated than what we thought possible for the ancients. Its like would not be seen again until the astronomical clocks devised by Richard of Wallingford in the 14th century[1].

Let us now turn to make our own computations of the solar system, exploiting the millennia of mathematical, scientific, and engineering advances between ourselves and the Greek designer of the Antikythera Mechanism, whose name is lost to history.

Our basic strategy will be to write the equations of motion for the Sun and major planets of the solar system, and use scipy to solve the resulting differential equations. Before we do so, however, there are several points we must address.

First, we wish to write the equations of motion in a general and flexible way. As we have seen in chapter 12, scipy's ODE integrators require us to specify a system of first-order differential equations; for this, the Newtonian and Lagrangian formulations of classical mechanics are not immediately suitable. Instead, we will derive the equations of motion from the *Hamiltonian* formulation of classical mechanics, devised by the Irish mathematician and polymath William Rowan Hamilton in 1833. Whereas Newtonian mechanics formulates problems in terms of forces, and Lagrangian mechanics in terms

[1] Richard of Wallingford was born in Wallingford, 21 km from Oxford. He was orphaned as a boy and raised by monks. He studied at the University of Oxford for fifteen years and became a monk himself. While serving as abbot of St Albans, he built perhaps the first astronomical clock since classical times, which calculated the motions of the Sun, moon, and planets. A manuscript describing the clock is preserved in the Bodleian Library. For more details, see

J. North. *God's clockmaker: Richard of Wallingford and the invention of time*. Hambledon Continuum, London, England, 2004

Richard of Wallingford, 1292–1336

Figure 13.1: Front and rear of the largest fragment of the Antikythera Mechanism, displaying the largest surviving gear.

of tradeoffs $T - V$ between the kinetic T and potential energies V, Hamiltonian mechanics derives the equations of motion from the total energy of the system, $T + V$, called the Hamiltonian[2]. Another key difference is that where Lagrangian mechanics uses generalised velocities (the time derivatives of the coordinates employed), Hamiltonian mechanics employed generalised momenta. For our purposes, the main advantage is that where the Newtonian and Lagrangian formulations yield n second-order differential equations, the Hamiltonian formulation yields $2n$ first-order differential equations[3].

Let the subscript $i = 0, \dots N - 1$ denote the bodies to simulate. Let $q_i(t) \in \mathbb{R}^3$ be the Cartesian position vector for body i at time t, and let $p_i(t) = m_i \dot{q}_i(t) \in \mathbb{R}^3$ be its associated momentum, with mass $m_i > 0$. Then the kinetic energy of the system is given by

$$T = \sum_i \frac{p_i^T p}{2m_i}, \tag{13.0.1}$$

and the potential energy of the system is

$$V = \sum_i \sum_{j>i} -\frac{G m_i m_j}{\|q_i - q_j\|}. \tag{13.0.2}$$

With these, we form the Hamiltonian

$$H = T + V \tag{13.0.3}$$

and derive the equations of motion via

$$\dot{q}_i = \frac{\partial H}{\partial p_i}, \quad \dot{p}_i = -\frac{\partial H}{\partial q_i}. \tag{13.0.4}$$

We will employ sympy for these manipulations, and use `sp.lambdify` to create the right-hand side for the general formulation of the ODE (12.6.3).

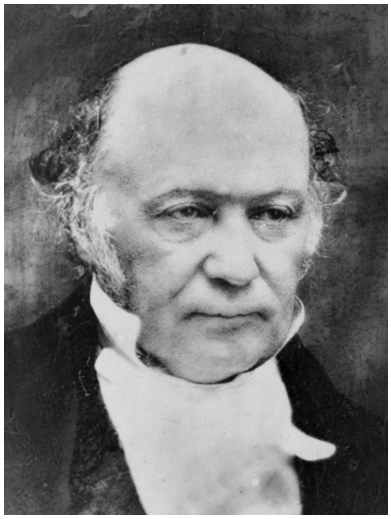
Second, any computation can only be as good as the data furnished to it. Since Newton's second law is a second-order differential equation, we will require the initial positions and velocities of the bodies concerned. To acquire these, we will employ the `astroquery` package[4], which queries the Horizons database maintained by the Solar System Dynamics group of NASA's Jet Propulsion Laboratory[5]. The package can be installed with

(terminal) `pip install astroquery`

as usual. For example, to get the position of the Earth-moon barycentre[6] at midnight on the author's birthday relative to the solar system barycentre, one could employ the code in code block 13.1.

[2] W. R. Hamilton. XV. On a general method in dynamics; by which the study of the motions of all free systems of attracting or repelling points is reduced to the search and differentiation of one central relation, or characteristic function. *Philosophical Transactions of the Royal Society*, 124:247–308, 1834

[3] Hamiltonian mechanics has another crucial advantage over previous formulations: it exposes the essential geometric structure of classical mechanics. Indeed, Hamiltonian mechanics had a significant impact on the development of geometry itself. The Hamiltonian formulation of classical mechanics was central to the subsequent development of quantum mechanics; the Hamiltonian arises as an operator in the Schrödinger equation. As previously noted, you will learn more if you take the third-year option *B7.1 Classical Mechanics*.



William Rowan Hamilton, 1805–1865

[4] A. Ginsburg et al. `astroquery`: an astronomical web-querying package in Python. *The Astronomical Journal*, 157(3):98, 2019

[5] J. D. Giorgini. Status of the JPL Horizons Ephemeris System. In *IAU General Assembly*, volume 29, page 2256293, 2015

[6] When simulating the solar system, one must understand the distinction between the barycentre of a planetary system and the planet itself. The barycentre is the centre of mass of the planet and its moons; it is this which we wish to track on its orbit around the Sun. The position of the planet itself will have small-scale oscillations as it is pulled this way and that by its moons.

Code block 13.1. Querying the JPL Horizons database.

```
from astroquery.jplhorizons import Horizons
from astropy.time import Time

date = Time('1985-03-17 00:00:00').jd
earth_moon_barycentre = 3  # code for Earth-moon barycentre
origin = '500@0'  # code for solar system barycentre
query = Horizons(id=earth_moon_barycentre, location=origin, epochs=date)
vec = query.vectors()

print(f"x: {vec['x'][0]} y: {vec['y'][0]}, z: {vec['z'][0]}")
```

The code 3 for the Earth-moon barycentre is specific to the Horizons database. The code for the barycentre of the Mercury system is 1, Venus is 2, etc. Code 10 refers to the Sun, while code 0 refers to the solar system barycentre[7].

Third, the default integrator used by `sc.integrate.solve_ivp` will not suffice. The default integrator is an explicit Runge–Kutta scheme of order 5(4) proposed by Dormand & Prince[8]. Briefly, this timesteps using a method whose error is proportional to the fifth power $(\Delta t)^5$ of the timestep Δt, and adaptively controls Δt by estimating the error using an embedded lower-order scheme of order 4. This is a good default choice, but we would like more accuracy than this can efficiently deliver, so we will switch to an explicit Runge–Kutta scheme of order 8(5, 3) written by E. Hairer[9,10].

Without further ado, the code is presented in code blocks 13.2–13.5. The results are shown in figure 13.2.

[7] For more details, see the Horizons manual at `https://ssd.jpl.nasa.gov/horizons/manual.html#select`.

[8] J. R. Dormand and P. J. Prince. A family of embedded Runge–Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, 1980

[9] See section (II.5) of E. Hairer, G. Wanner, and S. P. Nørsett. *Solving Ordinary Differential Equations I*, volume 8 of *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, 2nd edition, 1993

[10] Ernst Hairer is the father of Fields medallist Martin Hairer.

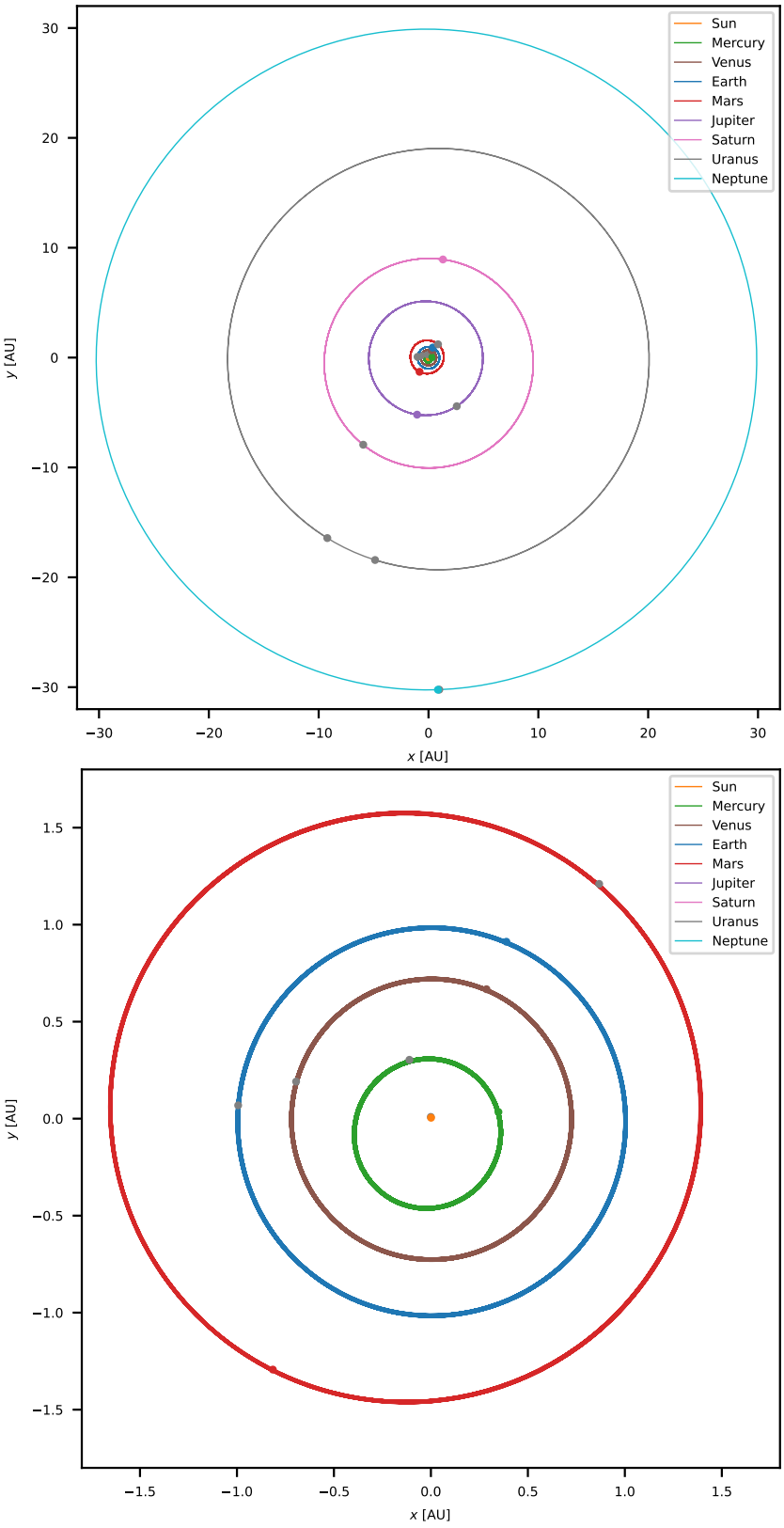


Figure 13.2: A simulation of the solar system over one Neptunian year, code blocks 13.2–13.5. The lower figure zooms in to the rocky inner planets.

Code block 13.2. Code for simulating the solar system: step zero.

```
import sympy as sp
import numpy as np
import scipy as sc
import matplotlib.pyplot as plt

from astroquery.jplhorizons import Horizons
from astropy.time import Time
import time

#
# Step 0. Gather data for the bodies to be simulated.
#

# Mapping from body to JPL Horizons ID
ids = {"Sun": 10,
       "Mercury": 1,
       "Venus": 2,
       "Earth": 3,
       "Mars": 4,
       "Jupiter": 5,
       "Saturn": 6,
       "Uranus": 7,
       "Neptune": 8}

# Masses in kg
masses = {"Sun": 1.9884e30,
          "Mercury": 3.301e23,
          "Venus": 4.867e24,
          "Earth": 5.97e24,
          "Mars": 6.417e23,
          "Jupiter": 1.898e27,
          "Saturn": 5.683e26,
          "Uranus": 8.681e25,
          "Neptune": 1.024e26}

# Colours to use in plotting: (red, green, blue)
colours = {"Sun":     (1.0000, 0.4982, 0.0549),
          "Mercury": (0.1725, 0.6274, 0.1725),
          "Venus":   (0.5490, 0.3372, 0.2941),
          "Earth":   (0.1215, 0.4666, 0.7058),
          "Mars":    (0.8392, 0.1529, 0.1568),
          "Jupiter": (0.5803, 0.4039, 0.7411),
          "Saturn":  (0.8901, 0.4666, 0.7607),
          "Uranus":  (0.4980, 0.4980, 0.4980),
          "Neptune": (0.0901, 0.7450, 0.8117)}

bodies = list(ids.keys())
N = len(bodies)
```

Code block 13.3. Code for simulating the solar system: step one.

```
#
# Step 1. Derive equations of motion from the Hamiltonian formulation.
#

pv = sp.symbols(f"p1:{3*N+1}", real=True)  # momenta
qv = sp.symbols(f"q1:{3*N+1}", real=True)  # positions

# Massage these scalars into a nicer vector format
momenta = {}
positions = {}
for (i, body) in enumerate(bodies):
    momenta[body] = sp.Matrix([pv[3*i+0], pv[3*i+1], pv[3*i+2]])
    positions[body] = sp.Matrix([qv[3*i+0], qv[3*i+1], qv[3*i+2]])

# Gravitational constant in AU, Earth days, kg
G = 1.4878e-34

# Construct kinetic energy
V = 0
for body in bodies:
    p = momenta[body]
    m = masses[body]
    V = V + p.dot(p)/(2*m)

# Construct potential energy
T = 0
for (i, body_a) in enumerate(bodies):
    m_a = masses[body_a]
    q_a = positions[body_a]

    for body_b in bodies[i+1:]:
        m_b = masses[body_b]
        q_b = positions[body_b]

        T = T - G * m_a * m_b / (q_a - q_b).norm()

# Write the Hamiltonian
H = V + T

# Derive equations of motion
equations = [-sp.diff(H, q) for q in qv] + [+sp.diff(H, p) for p in pv]

# Lambdify to construct right-hand side of ODE
t = sp.Symbol("t", real=True, nonnegative=True)
f = sp.lambdify([t, pv + qv], equations)
```

Code block 13.4. Code for simulating the solar system: step two.

```
#
# Step 2. Construct initial data by querying Horizons database.
#

ic = np.zeros(6*N)

# Start time of our simulation
epoch = Time('1985-03-17 00:00:00').jd

for (i, body) in enumerate(bodies):
    # Query Horizons database for coordinates relative to
    # barycentre of the solar system
    query = Horizons(id=ids[body], location='500@0', epochs=epoch)
    vec = query.vectors()

    # Multiply by mass to compute momentum
    ic[3*i+0] = vec['vx'][0]*masses[body]
    ic[3*i+1] = vec['vy'][0]*masses[body]
    ic[3*i+2] = vec['vz'][0]*masses[body]
    ic[3*i+3*N+0] = vec['x'][0]
    ic[3*i+3*N+1] = vec['y'][0]
    ic[3*i+3*N+2] = vec['z'][0]

    # Plot initial position of the planet in grey
    plt.plot(vec['x'][0], vec['y'][0], 'o', color="gray")
```

Code block 13.5. Code for simulating the solar system: step three.

```
#
# Step 3. Simulate and plot.
#

t0 = 0
t1 = 60191  # Just over one Neptune year
t_eval = np.linspace(t0, t1, t1+1)  # store simulation data every day

start = time.time()
trajectory = sc.integrate.solve_ivp(f, (t0, t1), ic, t_eval=t_eval,
                                    method='DOP853', rtol=1e-8, atol=1e-8)
end = time.time()
print(f"Simulation complete. Successful: {trajectory.success}. ", end="")
print(f"Time taken: {end - start:.2f} s.")

for (i, body) in enumerate(bodies):
    x = trajectory.y[3*N + 3*i + 0, :]
    y = trajectory.y[3*N + 3*i + 1, :]

    # Plot trajectory
    plt.plot(x, y, label=body, color=colours[body])

    # Plot final position of the planet
    plt.plot(x[-1], y[-1], 'o', color=colours[body])

    # Annotate final position with letter
    plt.annotate(body[0], (x[-1] + 0.08, y[-1] + 0.08), color=colours[body])

plt.legend()
plt.xlabel(r"$x$ [AU]")
plt.ylabel(r"$y$ [AU]")
plt.gca().set_aspect('equal')
plt.show()
```

14 Problem sheet 4

1. Geoffrey Ingram Taylor (1886–1975) was an English mathematician and physicist. He came from a mathematical family; his Irish mother was the daughter of George Boole. He studied mathematics and physics at Cambridge under Whitehead, Hardy, and Thomson, and eventually won a faculty position at Cambridge in mathematical meteorology. He made many fundamental contributions to fluid mechanics, such as in Taylor–Couette flow, the Taylor–Green solution of the incompressible Navier–Stokes equations, and the Rayleigh–Taylor instability; the latter arises in everything from clouds to nuclear explosions to supernovae. He was a keen outdoorsman; he loved to sail, participated in oceanographic cruises, learned to fly aircraft, and to parachute jump. He conducted research work for His Majesty's Government during both world wars; in the second world war he independently invented the concept of a nuclear fission bomb, and participated in the Manhattan Project, where he studied the propagation of blast waves. As part of this he directly observed the Trinity nuclear test on July 16 1945 that ushered in the atomic age.

   In 1950 Taylor published a famous paper[1] estimating the initial energy E released by the Trinity nuclear test, which was highly classified at the time, using timed photographs of the expansion of the spherical ball of fire from four unclassified sources. Solving the equations describing the conservation of momentum, mass, and energy, and the equation of state, he calculated that the radius of the ball $R(t)$ should be related to E, the density of air ρ_{air}, and time t through

   $$R(t) \propto E^{\frac{1}{5}} \rho_{\text{air}}^{-\frac{1}{5}} t^{\frac{2}{5}},$$

   and via experiment estimated that the constant of proportionality was about 1. (It is often erroneously reported that he derived this via dimensional analysis alone. This is not true. However, this relationship *is* recoverable via dimensional analysis if you assume that the radius only depends on these quantities.)

   Taylor has now asked for your help. Taking $\rho_{\text{air}} = 1.25 \text{ kg m}^{-3}$,

[1] G. I. Taylor. The formation of a blast wave by a very intense explosion. II. The atomic explosion of 1945. *Proceedings of the Royal Society A*, 201(1065):175–186, 1950

write a Python program to estimate the yield of the Trinity nuclear explosion from the data in table 14.1, gathered by Taylor in 1949. State your answer both in Joules and in kilotons of TNT, where the energy released by 1 kiloton of TNT is 4.184×10^{12} J.

[*Hint: a modern re-analysis by Hanson et al. (2016)*[2] *gives an estimate of* $E = 22.1 \pm 2.7$ kt.]

[2] S. K. Hanson, A. D. Pollington, C. R. Waidmann, W. S. Kinman, A. M. Wende, J. L. Miller, J. A. Berger, W. J. Oldham, and H. D. Selby. Measurements of extinct fission products in nuclear bomb debris: determination of the yield of the Trinity nuclear test 70 y later. *Proceedings of the National Academy of Sciences of the United States of America*, 113(29):8104–8108, 2016

2. Pierre–Simon, Marquis de Laplace (1749–1827) was a French mathematician. Raised in a bourgeois family in Normandy, he first studied to become a priest at the University of Caen, but quickly realised his vocation was in mathematics rather than theology. He won an appointment to teach at the École Militaire in Paris, where he was Napoleon's examiner in 1785. He made fundamental contributions to the study of probability, devised the correct mathematical theory of the tides, discovered spherical harmonics, finite differences, potential theory, the Laplace transform, and proposed the idea of black holes. The stability of the solar system was a lifelong obsession; Newton had believed that periodic divine intervention was necessary to guarantee the stability of the solar system, but Laplace was determined to prove him wrong. In pursuing this Laplace wrote his magnum opus, the *Traité de Mécanique Céleste*, published in five volumes from 1798 to 1825. The book was infamous for its invocation *Il est aisé à voir que* ... (it is easy to see that ...) whenever Laplace had mislaid the details of a derivation or proof. A favourite of Napoleon, he was appointed a count of the Empire in 1806; his relationship with the Emperor cooled considerably after his son fought in Napoleon's disastrous 1812 campaign in Russia. In 1814 he voted in the Senate to restore the Bourbon monarchy; he fled Paris when Napoleon briefly retook power in 1815. He was appointed a marquis in 1817 and died in 1827.

   In Volume IV of the *Traité de Mécanique Céleste*, Laplace discusses the orbits of the Galilean moons of Jupiter, Io, Europa, Ganymede and Callisto[3]. These moons are of a similar size to the rocky planets; Ganymede is larger than Mercury, while Callisto is very slightly smaller. The moons were discovered by Galileo and Simon Marius; they were the first objects discovered in the solar system since the classical planets of antiquity. They were also the first discovered to orbit a planet other than the Earth, dealing a fatal blow to the geocentric model of the solar system.

[3] P. S. Laplace. *Traité de Mécanique Céleste*, volume IV. Chez Courcier, Paris, 1805

   Laplace realised that the three inner moons (Io, Europa, and Ganymede) were in a *1:2:4 orbital resonance*: the orbital period of Europa is almost exactly twice that of Io, while the orbital period of Ganymede is almost exactly four times that of Io. (Callisto is not yet in resonance, but numerical simulations indicate it will enter a

1:2:4:8 resonance in about 1.5 billion years[4].)

Laplace has now asked for your help. He wishes to solve the equations of motion for the orbits of Io, Europa, Ganymede, and Callisto around Jupiter, but is otherwise occupied proving the Central Limit Theorem. Write a Python program for Laplace to simulate the orbits of Jupiter and the Galilean moons, and use your code to verify the 1:2:4 Laplacian resonance. Plot the orbits taken over 170 Earth days (approximately 10 orbits of Callisto).

[4] G. Lari, M. Saillenfest, and M. Fenucci. Long-term evolution of the Galilean satellites: the capture of Callisto into resonance. *Astronomy & Astrophysics*, 639:A40, 2020

[*Hint: the ID numbers of these bodies for the NASA/JPL Horizons database are given in table 14.2. Fetch their initial positions and velocity relative to the barycentre of the Jovian system using* `location='500@5'`.]

[*Hint: Use the* `events` *functionality of* `sc.integrate.solve_ivp` *to compute the orbital periods of the Galilean moons. Write a function for each moon that detects when its y-coordinate is the same as Jupiter's, and from this compute the mean orbital periods.*]

t (ms)	R (m)	t (ms)	R (m)	t (ms)	R (m)
0.1	11.1	1.36	42.8	4.34	65.6
0.24	19.9	1.50	44.4	4.61	67.3
0.38	25.4	1.65	46.0	15.0	106.5
0.52	28.8	1.79	46.9	25.0	130.0
0.66	31.9	1.93	48.7	34.0	145.0
0.80	34.2	3.26	59.0	53.0	175.0
0.94	36.3	3.53	61.1	62.0	185.0
1.08	38.9	3.80	62.9		
1.22	41.0	4.07	64.3		

Table 14.1: Radius of the Trinity nuclear test as a function of time.

Body	Horizons ID
Jupiter	599
Io	501
Europa	502
Ganymede	503
Callisto	504

Table 14.2: ID numbers of the relevant bodies in the NASA/JPL Horizons database.

General advice on computational projects

In Hilary term you will undertake two of the three computational projects listed below. Each project is worth up to twenty marks, which are split between mathematical content, programming skill, and clarity of exposition. The marks will count towards the Preliminary examinations; they carry the weight of one third of a paper. The deadlines for the projects are

- 12:00 (noon), Monday, week 6: online submission of first project;
- 12:00 (noon), Monday, week 9: online submission of second project.

The projects are to be submitted online via Inspera. The projects can be done in any order (e.g. one may submit project C in week 6 and project A in week 9). It is recommended that students familiarise themselves with Inspera and all data necessary for submission (e.g. single sign-on and examination candidate number) well in advance of the deadlines. The deadlines are strict and penalties for late submission apply.

Your submissions should consist of one or more `.py` files and a `.html` file produced from the main `.py` file via `publish`, as described in chapter 5. The examiners will primarily scrutinise the published `.html` file, but may modify and execute your code. The files for your submission should be gathered into exactly one `.zip` or `.tar.gz` file for upload.

A key difference from the problem sheets is that the project reports should be more expository. The mathematics behind and intent of the code written for the project must be made as clear as possible, since marks are awarded for mathematical understanding.

When you complete your projects, you must not upload them on the internet e.g. on web forums or in public code repositories. This is to assist in the prevention of plagiarism.

All projects must be your own unaided work. You will be asked to make a declaration to this effect when you submit them. The University's plagiarism policy applies in full, with potential penalties for plagiarism ranging from deduction of marks to expulsion from the University.

Frequently asked questions

Do I need to do any background reading for the projects?

No. References to the literature are included for any students who may be interested in learning more, but are not required to solve the problems.

Can we use TeX for our documentation?

It is not expected at this stage that students know TeX/LaTeX. However, if you are familiar with it, you are encouraged to use TeX notation for writing equations. TeX notation written in comments is rendered appropriately by `publish.py`. TikZ diagrams are not supported; any diagrams required by the projects should be rendered with matplotlib.

Are we allowed to use Jupyter Notebooks?

Again, it is not expected that students are familiar with Jupyter Notebooks. It is possible to use Jupyter Notebooks for your code development if you prefer, but your code should ultimately be submitted as a `.py` file. You can convert Jupyter Notebooks to plain Python with `jupyter nbconvert`.

How important is code optimisation?

The code should not be egregiously inefficient (e.g. having drastically worse scaling in time or memory than a straightforward implementation). Beyond that, do not invest too much effort into it; code clarity is more important than running as fast as possible. As long as the code runs in reasonable time, it suffices. The reference solutions for the projects each take no more than ten minutes or so to execute on modest hardware.

Should we determine the complexity of our algorithms in time and/or memory?

You do not need to do this unless explicitly requested.

How should I structure my code?

A good general structure for your code is to first write out the solution idea in comments, then give the (commented) implementation, then show the code is correct with examples. In many cases the question will indicate what examples to run your code on.

A *Primality testing*

(*This project relates to material in the Trinity term Prelims course* M1: Groups and Group Actions*, and in the Part A option* ASO: Number Theory.)

Computing whether a natural number is prime, and identifying its factors, are core tasks in number theory and in cryptography. As Carl Friedrich Gauss wrote in article 329 of his magnum opus *Disquisitiones Arithmeticae* (1801) (translated by Arthur A. Clarke, 1965[1]):

> The problem of distinguishing prime numbers from composite numbers and of resolving the latter into their prime factors is known to be one of the most important and useful in arithmetic. It has engaged the industry and wisdom of ancient and modern geometers to such an extent that it would be superfluous to discuss the problem at length. Nevertheless we must confess that all methods that have been proposed thus far are either restricted to very special cases or are so laborious and prolix that even for numbers that do not exceed the limits of tables constructed by estimable men, i.e. for numbers that do not require ingenious methods, they try the patience of even the practiced calculator. ... Further, the dignity of the science itself seems to require that every possible means be explored for the solution of a problem so elegant and so celebrated.

[1] C. F. Gauss. *Disquisitiones Arithmeticae*. Yale University Press, 1965. Translation by A. A. Clarke

The two tasks Gauss mentions—primality *testing* and number *factorisation*—are rather different. An algorithm is known for primality testing (due to Agrawal, Kayal, and Saxena[2]) that deterministically gives the correct answer with a runtime that is a polynomial function of the number of digits of the input. By contrast, the existence or nonexistence of a polynomial-time algorithm for factorising a number remains a major open problem in mathematics, and indeed most of the cryptography in practical use today relies centrally on its computational difficulty.

[2] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004

In this project you will investigate algorithms for testing whether a number is prime or not. For more details on this subject, see the book of Crandall and Pomerance[3].

[3] R. Crandall and C. Pomerance. *Prime Numbers: a Computational Perspective*. Springer-Verlag New York, second edition, 2010

A.1 Trial division

Trial division, the algorithm we met in Code block 7.11 and Exercise 7.6, was first described in Fibonacci's *Liber Abaci* (1202)[4]. In Chapter 5, Fibonacci writes

> If it is even, then he recognises its composition. However if odd, then it will be composite or prime ...Odd numbers truly are composed of odds alone. Whence the components of them by odds are investigated, for which we take the beginning. Therefore when in the figure of first place of any odd number there is the number 5, one will know 5 to be a factor. However, if another odd figure will appear in the first place, then one indeed takes the residue of number when divided by 9; if a zephir [i.e. zero] results, then 9 is a factor, and if 3 or 6 is the residue, then 3 is a factor; however if the residue will show none of these, one divides by 7; and if there will be an excess, then one again divides the number by 11; and if there is an excess, then he divides again by 13, and always he goes on dividing in order by prime numbers until he will find a prime number by which he can divide, and thence he will come to the square root; if he will be able to divide by none of them, then one will judge the number to be prime.

[4] L. Pisano (Fibonacci). *Liber Abaci*. Springer Science & Business Media, 2003. Translation by L. E. Sigler

We will employ some convenient notation for this question. Define $a : b$ as

$$a : b := [a, b] \cap \mathbb{Z}. \tag{A.1.1}$$

We denote numbers known to be prime by p, and the number whose primality we wish to determine by n.

Question A.1. Modify your code for Exercise 7.6 (which implements an efficient variant of trial division) to return `(flag, ndivisions)`, where `flag = True` if the input is prime and `False` otherwise, and `ndivisions` is the count of the number of divisions performed. Print the output of the function applied to all $n \in 2 : 20$. How many divisions are performed to test the primality of 9999991111111?

Question A.2. Compute the number of divisions performed for all numbers $n \in 2 : 10^5$. By means of a plot, verify that trial division takes about $\sqrt{n}/3$ divisions in the worst case to test a number n for primality.

Expressing $\sqrt{n}/3$ in terms of the logarithm of n, we see that the work involved in trial division is exponential in the number of digits. This exponential dependence on the number of digits motivates the search for more efficient algorithms.

A.2 The Fermat test

The trial division algorithm is based on the definition of primality, i.e. that n has no factors other than one and n. Ideally, we would base a test for primality on alternative condition that is equivalent to primality, i.e. that is necessary and sufficient for primality. Finding such conditions is tricky. Instead, we will base our next algorithm on a condition that is merely *necessary* for primality—all prime numbers satisfy the condition, but some composite numbers may also. The condition is inspired by Fermat's Little Theorem, which you will meet in Hilary Term *Groups and Group Actions*.

Theorem A.2.1 (Fermat, 1640)**.** *Let $p \in \mathbb{N}$ be prime. Let $a \in \mathbb{N}$ such that $\gcd(a, p) = 1$ (i.e. a is not a multiple of p). Then*

$$a^{p-1} \equiv 1 \mod p. \tag{A.2.1}$$

This inspires the following procedure. Let n be the number we wish to test for primality.

1. Choose $a \in 2 : (n-2)$.
2. Calculate the greatest common divisor $\gcd(a, n)$;[5] if the greatest common divisor is not 1, then n is composite.
3. Calculate $a^{n-1} \mod n$; if it is not congruent to 1, then n is composite.
4. If it is congruent to 1, then the test is inconclusive.

[5] The greatest common divisor can be efficiently computed using Euclid's algorithm, as in code block 4.9.

We refer to conducting this procedure for a single a as a *Fermat trial*; a *Fermat test* is to do this for a set of candidate values of a.

Of course, since a Fermat test relies on a condition that is only *necessary* for primality, it can only prove that n is *not prime*—in other words, this is actually a compositeness test. Nevertheless, if n passes enough trials, it might give us confidence that n is probably prime.

Question A.3. Write a function to implement the Fermat trial for given n and a.

Write another function to apply the Fermat test with all a in a given list; if no list is supplied, use as default value all $a \in 2 : (n-2)$ in ascending order. This latter function should return a tuple `(flag, ntrials)` where `flag = False` if the Fermat test has shown n to not be prime and `True` otherwise[6], and where `ntrials` is the number of Fermat trials performed. Print the output of the function applied to the natural numbers $n \in 2 : 20$, using in each case all $a \in 2 : (n-2)$ in ascending order.

[6] In other words, a number with flag `True` might still be composite.

[*Hint: the greatest common divisor can be computed using* `math.gcd`.]

[*Hint: in Python, the* `pow` *function takes an optional third argument.* `pow(x, y, z)` *calculates* $x^y \mod z$.]

Question A.4. Compute the first 5 odd numbers n where the Fermat test proves compositeness with just one trial, i.e. with $a = 2$.

Question A.5. For how many odd $n \in 3 : 10,000$ does the Fermat test prove compositeness with at most five trials (using $a \in 2 : \min(6, n-2)$)? What proportion of odd composite numbers in $3 : 10,000$ does this represent?

Question A.6. A *Carmichael number*, also called an absolute Fermat pseudoprime, is a composite number which passes the Fermat trial for any $a \in 2 : (n-1)$ with $\gcd(a, n) = 1$. Compute the Carmichael numbers up to $10,000$.

[*Hint: the first Carmichael number is* 561.]

In 1994, Alford, Granville & Pomerance proved that there are infinitely many Carmichael numbers[7]; for large enough n, there are at least $n^{2/7}$ Carmichael numbers in $1 : n$. This fact limits the utility of the standalone Fermat test; for the Fermat test to work on a Carmichael number, only those bases that share a factor with n will detect its compositeness, and choosing a few a's will likely not help us. However, in the words of Carl Pomerance, ''using the Fermat congruence is so simple that it seems a shame to give up on it just because there are a few counterexamples'' [8]. It is often used in combined algorithms to quickly test for compositeness with a handful of choices of a before subjecting n to more complicated algorithms.

[7] W. R. Alford, A. Granville, and C. Pomerance. There are infinitely many Carmichael numbers. *Annals of Mathematics*, 139(3):703, 1994

[8] F. Bornemann. PRIMES is in P: a breakthrough for ''Everyman''. *Notices of the American Mathematical Society*, 50(5):545–553, 2003

A.3 *Miller–Rabin primality test*

The Miller–Rabin test is a refinement of the Fermat test. Like the Fermat test, we will computationally determine whether a specific property that must hold for primes holds for the n in question. A deterministic version was introduced by Miller in 1976[9], with its correctness dependent on the (unproven) extended Riemann hypothesis; Rabin introduced a probabilistic version in 1980[10].

[9] G. L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976

[10] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980

To motivate the Miller–Rabin trial, suppose $p > 4$ is prime. Let $a \in 2 : (p-2)$ with $\gcd(a, p) = 1$. From Fermat's Little Theorem, we know that

$$a^{p-1} - 1 \equiv 0 \mod p. \tag{A.3.1}$$

Since $p - 1$ is even, the left-hand side is the difference of two squares, so we can write

$$\left(a^{\frac{p-1}{2}} - 1\right)\left(a^{\frac{p-1}{2}} + 1\right) \equiv 0 \mod p. \tag{A.3.2}$$

If $(p-1)/2$ is still even we can expand the left-most term further, as

$$\left(a^{\frac{p-1}{4}} - 1\right)\left(a^{\frac{p-1}{4}} + 1\right)\left(a^{\frac{p-1}{2}} + 1\right) \equiv 0 \mod p. \tag{A.3.3}$$

Repeating this process yields

$$\left(a^{\frac{p-1}{2^s}} - 1\right)\left(a^{\frac{p-1}{2^s}} + 1\right) \cdots \left(a^{\frac{p-1}{2}} + 1\right) \equiv 0 \mod p \tag{A.3.4}$$

for some s such that $p - 1 = 2^s d$ with d odd. The left-hand side is divisible by p; we wish to assert that p must divide one of the factors. To do so we invoke Euclid's lemma, given as Proposition 30 in Book VII of Euclid's Elements (approximately 300 B.C., translated by Richard Fitzpatrick, 2008[11]):

> If two numbers make some number by multiplying one another, and some prime number measures the number so created from them, then it will also measure one of the original numbers.

[11] Εκλεδης. Στοιχεα. 300 B.C. Translation by R. Fitzpatrick. Independently published

In modern language, we would express this as

Lemma A.3.1. *If a prime p divides the product ab of two integers a and b, then p must divide at least one of a or b.*

By the primality of p, we can therefore conclude that at least one of the following conditions must hold:

1. $a^d \equiv 1 \mod p$,
2. $a^{2^r d} \equiv -1 \equiv p - 1 \mod p$ for some $r \in 0 : (s-1)$,

where again $p - 1 = 2^s d$.

Now let $n > 4$ be the number whose primality we wish to test. Write $n - 1 = 2^s d$. For a given $a \in 2 : (n-2)$, the Miller–Rabin trial for $n > 4$ proceeds as follows. If one of the following conditions holds:

1. $a^d \equiv 1 \mod n$,
2. $a^{2^r d} \equiv -1 \equiv n - 1 \mod n$ for some $r \in 0 : (s-1)$,

then the trial is inconclusive. If none of these conditions hold, then the trial yields the conclusion that n is composite.

As with the Fermat test, the Miller–Rabin *test* consists of one or more Miller–Rabin trials with different choices of a. Choosing several bases at random gives a probabilistic primality test; Miller gave a clever choice of deterministic bases that guarantees correctness, subject to the extended Riemann hypothesis.

Question A.7. Write a function to implement the Miller–Rabin trial for given n and a.

Write another function to apply the Miller–Rabin test with all a in a given list; if no list is supplied, use as default value the single trial $a = 2$. This latter function should return a tuple `(flag, ntrials)` where `flag = False` if the Miller–Rabin test has shown n to not be prime and `True` otherwise[12], and where `ntrials` is the number of Miller–Rabin trials performed. Print the output of the function applied to the natural numbers $n \in 5 : 20$, using in each case only $a = 2$.

[12] In other words, a number with flag `True` might still be composite.

Question A.8. Using only the single trial with base $a = 2$, what is the minimal odd composite number n for which the test does not conclude that n is composite?

Question A.9. Using only the trials $a \in \{2, 3\}$, what is the minimal odd composite number n for which the test does not conclude that n is composite?

Adding a single additional trial to the Miller–Rabin test greatly extends the range of natural numbers for which the test is guaranteed to be accurate. For example, using $a \in \{2, 3, 5\}$ is guaranteed to give the correct answer for $n < 25,326,001$; using $a \in \{2, 3, 5, 7\}$ is guaranteed to give the correct answer for $n < 3,215,031,751$; using $a \in \{2, 3, 5, 7, 11\}$ is guaranteed to give the correct answer for $n < 2,152,302,898,747$[13].

[13] C. Pomerance, J. L. Selfridge, and S. S. Wagstaff. The pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35(151):1003–1026, 1980; and G. Jaeschke. On strong pseudoprimes to several bases. *Mathematics of Computation*, 61(204):915–926, 1993

Question A.10. Using trials $a \in \{2, 3, 5, 7, 11, 13, 17\}$[14], how much faster or slower is the Miller–Rabin test than trial division to verify the primality of $n = 9999991111111$?

[14] With these bases, the Miller–Rabin test is guaranteed to be correct for this n.

A.4 Concluding remarks

The current state of the art for deterministic primality testing is to combine one Miller–Rabin trial (with base $a = 2$) with another test we have not discussed, the *Lucas probable prime test*. This combination is known as the Baillie–Pomerance–Selfridge–Wagstaff (or Baillie–PSW) test[15], and is the algorithm behind the primality testing algorithms in Mathematica, Maple, PARI/GP, SageMath, and other symbolic algebra systems. The reason for the popularity of this algorithm is that no composite number is known that (falsely) passes the Baillie–PSW test. The construction of such a composite number, or a proof that no such number exists, would solve a major open question in computational number theory.

[15] R. Baillie and S. S. Wagstaff. Lucas pseudoprimes. *Mathematics of Computation*, 35(152):1391–1417, 1980; and C. Pomerance, J. L. Selfridge, and S. S. Wagstaff. The pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35(151):1003–1026, 1980

B The Kepler problem

(*This project relates to material in Prelims* M4: Dynamics, A7: Numerical Analysis, *and* B7.1: Classical Mechanics.)

The glorious triumph of Newton's twin discoveries of calculus and Newtonian mechanics was that it allowed us to make *physical predictions* by *solving differential equations*. Newton's second law provides an initial value problem for a second-order differential equation that, if solved, describes the motion of the given system for all future time. One of the first examples of the first written treatment of calculus, Problema II, Solutio Casus II, Ex. I of Newton (1671)[1], is to solve

$$\dot{y} := \frac{\mathrm{d}y}{\mathrm{d}t} = 1 - 3t + y + t^2 + ty \tag{B.0.1}$$

which Newton does by means of an infinite series. Indeed, in the bitter dispute with Leibniz over the discovery of calculus, Newton wrote[2]

> ...and by the Answer of Mr. Leibnitz to the first of those Letters, it is as certain that he had not then found out the Reduction of Problems either to differential Equations or to converging Series.

Of course, most differential equation initial value problems cannot be solved exactly, and so numerical methods for their approximate solution were (and remain) of pressing concern[3]. In this project you will investigate numerical algorithms for computing approximate solutions of ordinary differential equation initial value problems. For more details on this subject, see the books of Hairer, Wanner & Nørsett[4], or Hairer, Lubich & Wanner[5].

We will focus our investigations on the two-body Kepler problem, modelling the orbit of a single planet around a star. As discussed in Chapter 9, the two-body problem is amenable to symbolic analysis that does not extend to three or more bodies. However, it is best to study the qualitative and quantitative accuracy of our numerical algorithms on the simplest possible case; if an algorithm does not work for two bodies, we cannot reasonably expect it to work for three or more.

[1] I. Newton. *The Method of Fluxions and Infinite Series*. Henry Woodfall; and sold by John Nourse, 1671. Translated from the Author's Latin Original Not Yet Made Publick. To which is Subjoin'd, a Perpetual Comment Upon the Whole Work, By John Colson. Published in 1736.

[2] I. Newton. An account of the book entituled Commercium Epistolicum Collinii et Aliorum, de Analysi Promota. *Philosophical Transactions of the Royal Society of London*, 342:173–224, 1715

[3] Even for ordinary differential equation initial value problems, important mathematical and algorithmic advances continue to this day.

[4] E. Hairer, G. Wanner, and S. P. Nørsett. *Solving Ordinary Differential Equations I*, volume 8 of *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, 2nd edition, 1993

[5] E. Hairer, C. Lubich, and G. Wanner. *Geometric Numerical Integration*, volume 31 of *Springer Series in Computational Matheematics*. Springer-Verlag, 2006

B.1 *Equations of motion and invariants*

Following section I.2 of Hairer et al.[6], we choose one of the two bodies to be the origin of our coordinate system. Since two-body motion remains in a plane (a fact you will prove in Dynamics), we consider the position $\mathbf{q} = (q_1, q_2)$ and momentum $\mathbf{p} = (p_1, p_2)$ as two-dimensional vector-valued functions of time. Setting all physical constants to one for convenience, the Hamiltonian for the system is

$$H(\mathbf{p}, \mathbf{q}) = \frac{1}{2}\left(p_1^2 + p_2^2\right) - \frac{1}{\sqrt{q_1^2 + q_2^2}}. \tag{B.1.1}$$

[6] E. Hairer, C. Lubich, and G. Wanner. *Geometric Numerical Integration*, volume 31 of *Springer Series in Computational Matheematics*. Springer-Verlag, 2006

Question B.1. Symbolically calculate with sympy the resulting system of ordinary differential equations. (Recall (13.0.4).)

As a Hamiltonian system, the equations of motion you derive in Question B.1 must structurally preserve the Hamiltonian, the total energy of the system. The Kepler problem has other invariants, however[7]. Kepler's second law is equivalent to the statement that the angular momentum is conserved, which is

$$\mathbf{L}(\mathbf{p}, \mathbf{q}) = \begin{pmatrix} p_1 \\ p_2 \\ 0 \end{pmatrix} \wedge \begin{pmatrix} q_1 \\ q_2 \\ 0 \end{pmatrix}. \tag{B.1.2}$$

[7] In fact, the Kepler problem is *maximally superintegrable*—it has as many invariants of motion as is possible to have. The concept of a system being integrable is rather subtle; Oxford's Nigel Hitchin, emeritus Savilian Professor of Geometry, gives a useful introduction in the first few pages of

N. J. Hitchin, G. B. Segal, and R. S. Ward. *Integrable systems: Twistors, loop groups, and Riemann surfaces*, volume 4 of *Oxford Graduate Texts in Mathematics*. Clarendon Press, 1999

Question B.2. Prove by symbolic substitution with sympy that the Hamiltonian is conserved, i.e. that its value does not change over time.

Question B.3. Prove by symbolic substitution with sympy that the angular momentum is conserved.

These two invariants are also conserved by n-body problems. The case $n = 2$ has one further invariant, the so-called Laplace–Runge–Lenz (LRL) vector:

$$\mathbf{A}(\mathbf{p}, \mathbf{q}) = \begin{pmatrix} p_1 \\ p_2 \\ 0 \end{pmatrix} \wedge \begin{pmatrix} 0 \\ 0 \\ q_1 p_2 - q_2 p_1 \end{pmatrix} - \frac{1}{\sqrt{q_1^2 + q_2^2}} \begin{pmatrix} q_1 \\ q_2 \\ 0 \end{pmatrix}. \tag{B.1.3}$$

Because of its more complicated form, this invariant is much less well known, and has thus been rederived independently many times (by, among others, Hermann, Bernoulli, Laplace, Hamilton, and Gibbs).

The LRL vector points from the star being orbited to the point of closest approach, the periapsis. The LRL vector was crucial to Pauli's quantum mechanical analysis of the hydrogen atom, which is a two-body problem with a central force governed by an inverse square law, discussed in Chapter 11.

Question B.4. Prove by symbolic substitution with sympy that the LRL vector is conserved.

The conservation of H, $\mathbf{L}$, and $\mathbf{A}$ encode the crucial geometric property of Kepler's problem, that the planet should orbit in an ellipse. (Roughly speaking, H and $\mathbf{L}$ encode the shape of the ellipse, while $\mathbf{A}$ encodes its orientation.) An important way to study the effectiveness of numerical algorithms for solving differential equations is to consider how they preserve the geometric properties encoded in the equations; algorithms that honour the underlying geometry are studied in the field of *geometric numerical integration*. In our case, we are particularly interested to what extent numerical algorithms yield discretisations that conserve H, $\mathbf{L}$, and $\mathbf{A}$, given in (B.1.1), (B.1.2), and (B.1.3); if these invariants are conserved, then the discrete solution will be the correct ellipse, but if they are not, the approximate trajectory will deviate from this. Measuring the conservation error therefore gives insight into whether a particular numerical approximation is unphysical or not.

B.2 *Euler's method*

The simplest possible method for solving initial value problems is the *forward Euler method*. Suppose we are solving

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}), \tag{B.2.1a}$$

$$\mathbf{y}(t_0) = \mathbf{y}_0. \tag{B.2.1b}$$

Let $t_1 = t_0 + \Delta t$ with $\Delta t > 0$ small. Euler's suggestion is to make the approximation that $f(t, \mathbf{y})$ is constant over $[t_0, t_1]$, with value $f(t_0, \mathbf{y}_0)$. Integrating the equation over this interval then yields that

$$\mathbf{y}(t_1) = \mathbf{y}(t_0) + \Delta t f(t_0, \mathbf{y}_0). \tag{B.2.2}$$

More generally, denoting

$$t_n = t_0 + n\Delta t, \quad \mathbf{y}_n \text{ our approximation for } \mathbf{y}(t_n), \tag{B.2.3}$$

the forward Euler scheme is to iteratively compute

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t f(t_n, \mathbf{y}_n). \tag{B.2.4}$$

The forward Euler method was proposed by Euler in *Institutiones Calculi Integralis* (the Foundations of Integral Calculus) in 1768[8]. In Volume I, Section II, Chapter 7 (translated by Ian Bruce in 2010), Euler writes

[8] L. Euler. *Institutiones Calculi Integralis*. Academia Imperialis Scientiarum, 1768. Translation by I. Bruce. Independently published

> **Concerning the approximate integration of differential equations.**
>
> **Problem 85**. To assign an approximate value to the complete integral of any differential equation.
>
> **Solution**. Let x and y be two variables, between which the differential equation is proposed, and this equation shall have a form of this kind, so that $\frac{dy}{dx} = V$ with V being some function of x and y. Now since the complete integral is desired, this has to be interpreted thus, so that while x is given a certain value, for example $x = a$, the other variable y is given a certain value, for example $y = b$. Hence in the first place we are to treat the question, so that we can find the value of y, when the value of x is attributed a value differing a little from a, on putting $x = a + \omega$ so that we may find y. But since shall be the smallest possible amount, the value of y will differ minimally from b ; from which, while x only is changed from a as far as to $a + \omega$, the quantity V is allowed to be looked on as being constant. Whereby on putting $x = a$ and $y = b$ there is made $V = A$ and from this very small change we will have $\frac{dy}{dx} = A$ and thus on integrating, $y = b + A(x - a)$, clearly with a constant of this kind to be added so that on putting $x = a$ there becomes $y = b$. Hence we may put in place $x = a + \omega$ and there becomes $y = b + A$.
>
> Hence just as here from the values given initially $x = a$ and $y = b$ we find approximately the following $x = a + \omega$ and $y = b + A$, thus from these in a like manner it is allowed to progress through another very short interval, as long as it arrives finally at values however far from the starting value.

This is a natural first idea; if Δt is small enough, the error in the value of $\mathbf{y}(t_\star)$ for some fixed $t_\star$ converges linearly as Δt is reduced (i.e. if you halve the timestep, the error in the approximation also halves). However, the forward Euler scheme does not pay heed to the geometric structure of our problem, with disastrous physical consequences, as we shall soon see.

Question B.5. Write a function to execute the forward Euler scheme to approximate the solution of the Kepler problem with initial conditions

$$\mathbf{p}(t = 0) = [0, 2], \quad \mathbf{q}(t = 0) = [0.4, 0], \tag{B.2.5}$$

for a specified timestep and number of timesteps.

On a single figure, plot the trajectory computed with forward Euler with

1. 50 steps of timestep $\Delta t = 0.1$;

2. 100 steps of timestep $\Delta t = 0.05$;

3. 200 steps of timestep $\Delta t = 0.025$.

For comparison, on the same figure plot also the true orbit, given in parametric form by

$$q_1 = -0.6 + \cos s, \quad q_2 = 0.8 \sin s, \quad s \in [0, 2\pi]. \tag{B.2.6}$$

Comment on your results.

Question B.6. For the finest discretization, on separate figures plot the computed values of H, L, and θ, where

$$L := \|\mathbf{L}\|, \quad \theta := \arg \mathbf{A} \tag{B.2.7}$$

as a function of time, where arg returns the azimuthal angle from the positive x-axis of the vector projected to the x-y plane. Are the invariants conserved by the forward Euler discretisation? What are the implications of this for the numerical approximation to the planet's orbit?

These results motivate the search for more sophisticated algorithms.

B.3 *Explicit midpoint method*

Over an interval $[t_n, t_{n+1}]$, the forward Euler method approximates the slope of the tangent to the solution $\mathbf{y}(t)$ via its (known) value at the left end-point. A natural objection to this procedure is that it *violates the symmetry inherent in the equations*: whereas Newton's laws of motion are symmetric forward or backward in time[9], the forward Euler method is not symmetric in time. More precisely, exchanging $t_n \leftrightarrow t_{n+1}$, $\mathbf{y}_n \leftrightarrow \mathbf{y}_{n+1}$, and $\Delta t \leftrightarrow -\Delta t$ in (B.2.4), we do not recover (B.2.4)[10].

This consideration of symmetry prompts us to seek a numerical method where the slope of our approximation matches the right-hand side of the ordinary differential equation *at the midpoint of the interval*, i.e. to find $\mathbf{y}_{n+1}$ such that

$$\frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{\Delta t} = f\left(\frac{t_n + t_{n+1}}{2}, \frac{\mathbf{y}_n + \mathbf{y}_{n+1}}{2}\right). \tag{B.3.2}$$

The numerical method defined by (B.3.2) is clearly symmetric in time by construction. However, it has a substantial disadvantage: to compute $\mathbf{y}_{n+1}$, we need to solve (B.3.2), which in general is a nonlinear

[9] Indeed, the invariance of the laws of physics in time is precisely what leads to the conservation of energy: Noether's Theorem, one of the most beautiful in all of mathematics, asserts that every differentiable symmetry of a physical system has a corresponding conservation law.

E. Noether. Invariante Variationsprobleme. *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, pages 235–257, 1918

[10] Instead, we recover

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t f(t_{n+1}, \mathbf{y}_{n+1}). \tag{B.3.1}$$

This scheme is known as *backward Euler*.

equation. (Notice that the right-hand side depends on the unknown $\mathbf{y}_{n+1}$.) The method defined by (B.3.2) is described as *implicit*, in contrast to *explicit* methods like forward Euler (B.2.4), where one can directly compute $\mathbf{y}_{n+1}$ from $\mathbf{y}_n$. This method is therefore known as *implicit midpoint*.

Moreover, implicit midpoint has a more fundamental geometric property: the associated discrete system is *symplectic*, just as our Hamiltonian system is. Explaining symplecticity is beyond the scope of this project, but for our purposes the following rough idea will suffice. Imagine an ordinary differential equation with $\mathbf{y} \in \mathbb{R}^2$. The initial data are thus an element of $\mathbb{R}^2$. Imagine taking a (measurable) set A of many different initial conditions in $\mathbb{R}^2$, and integrating the equations of motion up to an arbitrary fixed time $T > 0$ for each $a \in A$. This procedure will give another set of points $B \subset \mathbb{R}^2$. Symplecticity means that the *area is preserved* under this procedure: the area of B is exactly the area of A, for any terminal time T. Symplecticity is crucial to the geometry of Hamiltonian mechanics; implicit midpoint preserves symplecticity, whereas forward Euler does not.

Implicit midpoint is therefore a very powerful and popular integrator[11]. However, in this project we will confine our attention to explicit schemes, as implementing implicit schemes requires a good knowledge of the numerical solution of nonlinear equations, which you will study in Trinity Term *Constructive Mathematics*. An explicit alternative is to instead approximate

$$\frac{\mathbf{y}_n + \mathbf{y}_{n+1}}{2} \approx \mathbf{y}_n + \frac{\Delta t}{2} f(t_n, \mathbf{y}_n), \tag{B.3.3}$$

i.e. we estimate the average of the initial and final states with a half-step of forward Euler. This yields the *explicit midpoint* method:

$$\frac{\mathbf{y}_{n+1} - \mathbf{y}_n}{\Delta t} = f\left(\frac{t_n + t_{n+1}}{2}, \mathbf{y}_n + \frac{\Delta t}{2} f(t_n, \mathbf{y}_n)\right). \tag{B.3.4}$$

Unlike implicit midpoint, explicit midpoint is neither symmetric nor symplectic, but it is nevertheless a substantial improvement over forward Euler: on halving the timestep, the error in $\mathbf{y}_n$ decreases by a factor of 4, instead of the factor of 2 yielded by forward Euler.

[11] It is especially popular for problems where implicit integrators are necessary for other reasons, such as the parabolic partial differential equations you will meet in Part A *A1: Differential Equations I*.

Question B.7. Write a function to execute the explicit midpoint scheme to approximate the solution of the Kepler problem. Make an analogous plot as in Question B.5, with the same initial conditions, timestep Δt, and number of steps.

Question B.8. Compare on a plot a run of forward Euler with a run

of explicit midpoint. Make the comparison fair by ensuring that both schemes require the same number of evaluations of the right-hand side f. Comment on your results.

Question B.9. Comment on the conservation properties of explicit midpoint. (No plot necessary.)

B.4 *Newton–Störmer–Verlet method*

Is it possible to devise an explicit scheme for our problem that is both symmetric and symplectic?

Our intuition for the explicit and implicit midpoint schemes was that we wished the *slope* of our approximation at the midpoint of the time interval $[t_n, t_{n+1}]$ to match that specified by the ODE. This makes sense for a generic first-order system of ODE. But we are not solving just any first-order system of ODE—we are solving a first-order reformulation of a problem that is fundamentally second-order (recall Newton's second law). In other words, our equations are of the particular form

$$\dot{\mathbf{p}} = g(\mathbf{q}), \tag{B.4.1a}$$

$$\dot{\mathbf{q}} = \mathbf{p}, \tag{B.4.1b}$$

which is the first-order reformulation of

$$\ddot{\mathbf{q}} = g(\mathbf{q}). \tag{B.4.2}$$

Since g tells us *the second derivative of* $\mathbf{q}$, this instead suggests we should find a numerical approximation that matches the *second derivative* at $\mathbf{q}_n$ with the right-hand side evaluated there. That is, given $\mathbf{q}_{n-1}$ and $\mathbf{q}_n$, we compute the next value $\mathbf{q}_{n+1}$ such that the second derivative of the quadratic function that passes through $(t_{n-1}, \mathbf{q}_{n-1})$, $(t_n, \mathbf{q}_n)$, and $(t_{n+1}, \mathbf{q}_{n+1})$ is $g(\mathbf{q}_n)$). After some algebra, the scheme that results is

$$\mathbf{q}_{n+1} - 2\mathbf{q}_n + \mathbf{q}_{n-1} = (\Delta t)^2 g(\mathbf{q}_n). \tag{B.4.3}$$

The method (B.4.3) has been invented many times, with different names used in different communities[12]. Its most common name is the Verlet method, invented by Loup Verlet in 1967[13] in the context of molecular dynamics. It is sometimes called the Störmer method, as Carl Störmer[14] used higher-order variants of it to compute the motion of ionised particles in the Earth's magnetic field to understand the *aurora borealis*[15]. Loup Verlet subsequently became interested in the history of science and discovered the method that had made

[12] This historical account is drawn from E. Hairer, C. Lubich, and G. Wanner. Geometric numerical integration illustrated by the Störmer–Verlet method. *Acta Numerica*, 12:399–450, 2003

[13] L. Verlet. Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard–Jones molecules. *Physical Review*, 159(1):98, 1967

[14] C. Störmer. Sur les trajectoires des corpuscules électrisés. *Archives des Sciences Physiques et Naturelles*, 24:5–18, 113–158, 221–247, 1907

[15] For more details, see Section III.10 of E. Hairer, G. Wanner, and S. P. Nørsett. *Solving Ordinary Differential Equations I*, volume 8 of *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, 2nd edition, 1993

him famous had been employed by Newton in 1687 to prove Kepler's second law, the conservation of angular momentum in the two-body Kepler problem—in Theorem I of Book I of the *Principia*. We therefore refer to it here as the Newton–Störmer–Verlet method.

It turns out that the direct implementation of (B.4.3) suffers from a numerical instability in the presence of (inevitable) rounding errors[16]. A more stable equivalent implementation arises for the first-order system (B.4.1). Given $\mathbf{p}_0$ and $\mathbf{q}_0$, the Newton–Störmer–Verlet scheme is to compute

[16] For more details, see pg. 472 of Hairer et al. (1993).

$$\mathbf{p}_{n+\frac{1}{2}} = \mathbf{p}_n + \frac{\Delta t}{2} g(\mathbf{q_n}), \tag{B.4.4}$$

$$\mathbf{q}_{n+1} = \mathbf{q}_n + \Delta t \mathbf{p}_{n+\frac{1}{2}}, \tag{B.4.5}$$

$$\mathbf{p}_{n+1} = \mathbf{p}_{n+\frac{1}{2}} + \frac{\Delta t}{2} g(\mathbf{q_{n+1}}). \tag{B.4.6}$$

Like explicit and implicit midpoint, Newton–Störmer–Verlet is second-order accurate: halving the timestep quarters the error. But unlike explicit midpoint, it is both symmetric and symplectic, with crucial qualitative advantages in the simulation of the class of problems to which it applies.

Question B.10. Write a function to execute the Newton–Störmer–Verlet scheme to approximate the solution of the Kepler problem. Make an analogous plot as in Question B.5, with the same initial conditions, timestep Δt, and number of steps.

Question B.11. Compare on a plot a run of explicit midpoint with a run of Newton–Störmer–Verlet, both employing $\Delta t = 0.05$ for 1200 steps. Comment on your results.

Question B.12. Discuss the conservation properties of Newton–Störmer–Verlet. Provide evidence for your assertions, and relate your results to your answer for the previous question.

B.5 *Concluding remarks*

Hamiltonian problems (like the Kepler problem) possess an extremely rich mathematical structure. The associated differential equations are symplectic, conserve the Hamiltonian, and possibly conserve other

invariants also. However, when discretising, one must in general make a choice of structure to preserve: an approximate integrator cannot generally preserve both symplecticity and the Hamiltonian[17]. For chaotic systems, preserving symplecticity is probably the right choice, as it is crucial for their statistical behaviour; the inevitable discretisation errors mean that any individual trajectory is not particularly meaningful, but symplecticity ensures their aggregation is.

However, for other systems, it may be preferable to choose approximate schemes that exactly conserve the invariants of the system. For example, for the Kepler problem, the trajectory of such an approximation would be confined to exactly the same ellipse as that of the true solution, which is very appealing. More generally, the design of such structure-preserving discretisations for physical systems such as the Navier–Stokes equations of fluid mechanics or the Einstein field equations of general relativity is a major focus of ongoing research in the numerical analysis of differential equations.

[17] G. Zhong and J. E. Marsden. Lie–Poisson Hamilton–Jacobi theory and Lie–Poisson integrators. *Physics Letters A*, 133(3):134–139, 1988

C Percolation

(*This project relates to material in Prelims and Part A courses on* Probability *and* Statistics*, and Part A* Simulation and Statistical Programming.)

Statistical mechanics is the branch of mathematics that applies statistical and probabilistic methods to large assemblies of microscopic entities. For example, one might consider a gas as composed of a very large number of molecules moving in all directions and colliding with each other. From this viewpoint we wish to derive macroscopic properties like its pressure or temperature. Of course, keeping track of the state of each individual molecule among the trillions of trillions in a typical cubic metre is simply impossible. One therefore instead develops a theory where one considers the *probability distribution* of the molecules of the gas; given knowledge of the probability distribution, we can at any time calculate the number of molecules of a certain velocity range in a certain volume of space. In 1860, Maxwell calculated the equilibrium distribution for a gas at a given temperature, now known as the Maxwellian[1]; in 1872 Boltzmann derived the equation governing the time evolution of the probability distribution function, now known as the Boltzmann equation[2].

The Boltzmann equation is a nonlinear integro–differential equation, where the unknown is a function in six dimensions (three of position, three of velocity). It is therefore not terribly surprising that it is rather hard to solve. Computational simulations are crucial to gaining mathematical and physical insight into most systems of statistical mechanics.

A major goal of statistical mechanics is to understand *phase transitions*. A phase transition is an abrupt, discontinuous change in the properties of a system. For example, if we take our gas and cool it, at a critical temperature it will (usually) turn into a liquid, with its density and volume changing discontinuously. Phase transitions are of enormous mathematical, physical, and economic importance. For example, some materials (known as superconductors) exhibit a phase transition at a critical temperature, below which they offer no resistance to electrical current. The discovery of a practical superconducting material where the critical temperature is above room temperature would

[1] J. C. Maxwell. V. Illustrations of the dynamical theory of gases. Part I. On the motions and collisions of perfectly elastic spheres. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 19(124):19–32, 1860

[2] L. Boltzmann. Weitere Studien über das Wärmegleichgewicht unter Gasmolekülen. *Sitzungsberichte der Akademie der Wissenschaften zu Wien*, 66:275–730, 1872

trigger a second industrial revolution[3].

One route to understanding phase transitions is to consider simple mathematical models that exhibit them. A prominent class of such mathematical models is studied in *percolation theory*[4]. Percolation theory describes the properties of a graph as nodes or edges are added. Hugo Duminil-Copin won the Fields Medal in 2022 for his work on percolation theory; a popular account of his work was published in Quanta magazine[5].

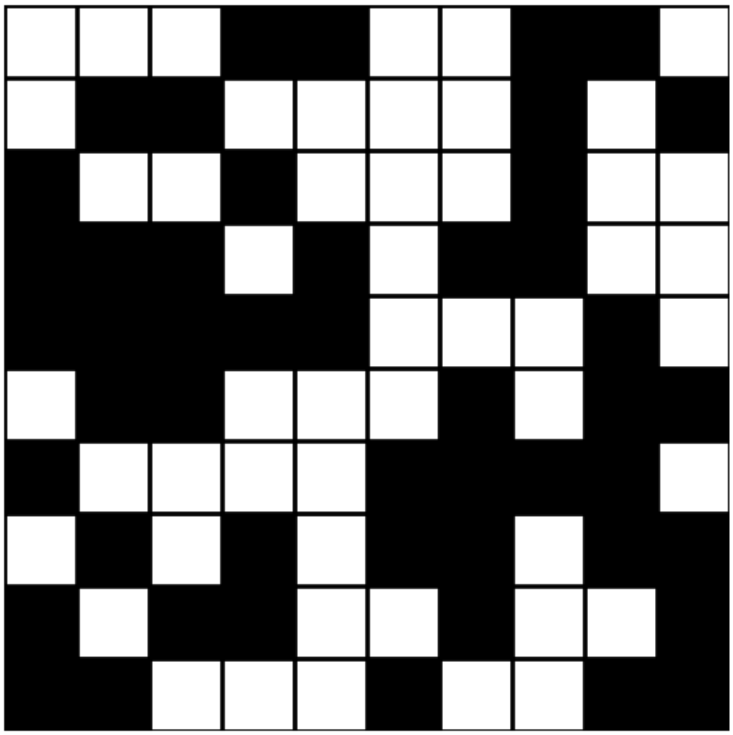


Figure C.1: A 10×10 grid sampled with vacancy probability $p = 0.5$. Open squares are white; closed squares are black.

The specific percolation model we consider in this project is the following. Consider an $n \times n$ grid of squares, where each site can be either *open* or *closed*, as in Figure C.1. We say that a site is *full* if it is open and can be connected to an open site in the top row via chain of neighbouring (left, right, up, down) open sites. If there is a full site on the bottom row, we say the system percolates (see Figures C.2 and C.3 for examples). In other words, is there a connected open component spanning from the top of the grid to the bottom? You might imagine this to model the question of whether water poured on the top will percolate to the bottom (hence the name), or whether a fire started at one end of a forest will propagate tree-by-tree to the other.

A natural question to ask about this system is: if each site is open with *vacancy probability* p, what is the probability that the system percolates, $C(p)$? As we will see, the percolation probability $C(p)$

[3] The current record at atmospheric pressure is held by the cuprate of mercury, barium, and calcium, which has a critical temperature around -140 °C.

[4] D. Stauffer and A. Aharony. *Introduction to Percolation Theory*. Taylor & Francis, second edition, 1994

[5] Hugo Duminil-Copin wins the Fields medal. *Quanta Magazine*, 2022

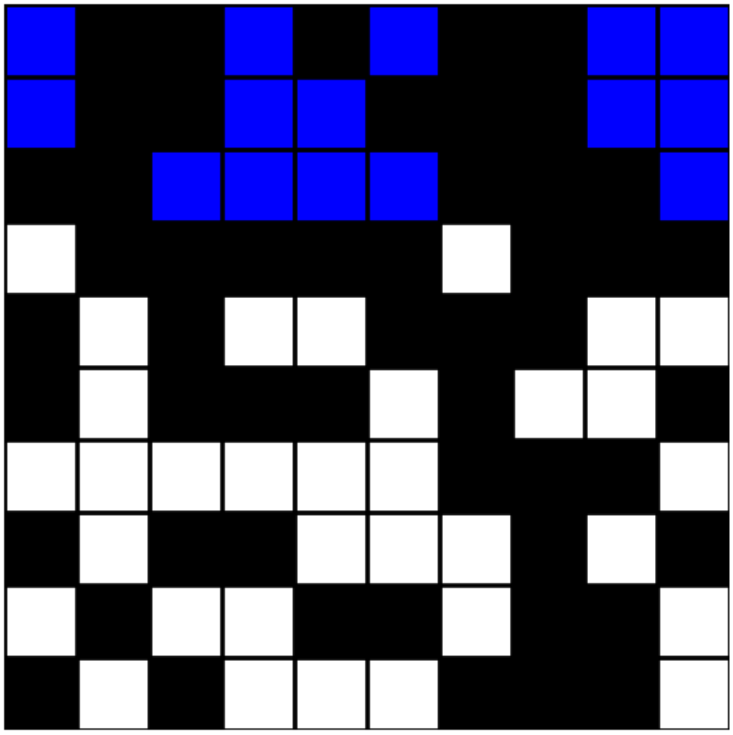


Figure C.2: A 10 × 10 grid sampled with vacancy probability $p = 0.5$. Full squares are blue. The system does not percolate.

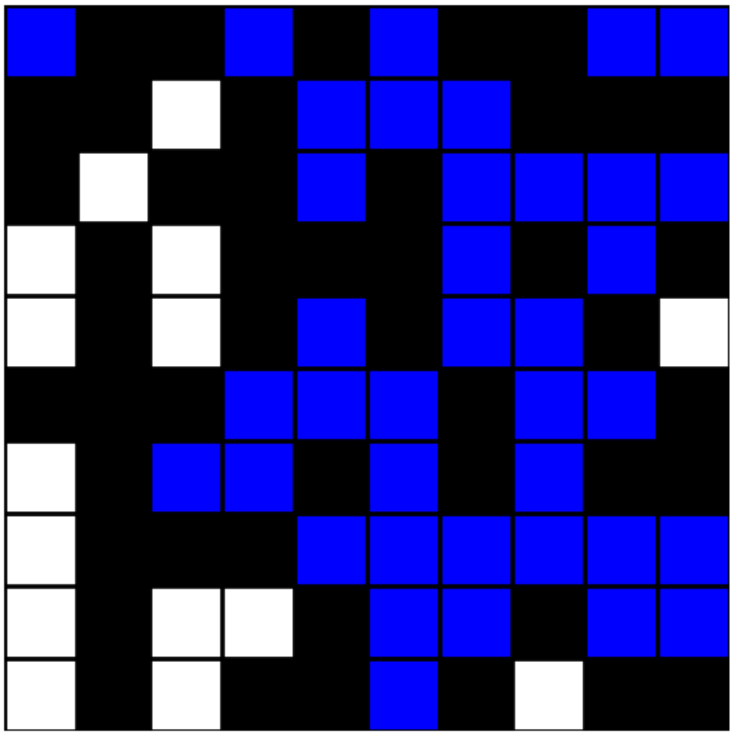


Figure C.3: A 10 × 10 grid sampled with vacancy probability $p = 0.5$. Full squares are blue. The system does percolate.

exhibits a phase transition in the vacancy probability. For low values of p, the system does not percolate; at a critical $p = p_c$ the system very rapidly switches to always percolating[6]. No analytical results are known characterising p_c. As described by Newman & Ziff[7],

> Percolation is one of the best-studied problems in statistical mechanics. ...It is one of the simplest and best understood examples of a phase transition in any system, and yet there are many things about it that are still not known. For example, despite decades of effort, no exact solution for the site percolation problem yet exists on the simplest two-dimensional lattice, the square lattice, and no exact results are known on any lattice in three dimensions or above. Because of these and many other gaps in our current understanding of percolation, numerical simulations have found wide use in the field.

By means of such numerical simulations, Newman & Ziff computed that the critical probability p_c for large n was approximately $p_c \approx 0.59274621$[8].

In this project you will investigate Monte Carlo algorithms for estimating the percolation probability $C(p)$. Monte Carlo methods are one of the most important and prominent tools for modern statistical inference. In a Monte Carlo simulation, we randomly draw inputs from a suitable probability distribution (in this case, the binomial distribution), perform deterministic computations (in this case, decide whether the grid percolates or not), and aggregate the results (to compute $C(p)$).

[6] In fact, in the limit of infinite grid size, the percolation probability for a given p is either zero or one, by Kolmogorov's zero-one law. For small n the transition becomes smoother.

[7] M. E. J. Newman and R. M. Ziff. Fast Monte Carlo algorithm for site or bond percolation. *Physical Review E*, 64:016706, 2001

[8] M. E. J. Newman and R. M. Ziff. Efficient Monte Carlo algorithm and high-precision results for percolation. *Physical Review Letters*, 85(19):4104–4107, 2000

C.1 *Representing the state*

Our first task is to generate suitable random samples of our grids.

Question C.1. Write a function `make_grid(n, p)` to make an $n \times n$ numpy array of Boolean values, with each site `True` with probability p and `False` otherwise.

[*Hint: this should take one line of numpy code.*]

Draw a few samples to ensure that the empirical probability of a site being open is approximately p.

Our next task is to visualise our grid status. This will be very useful in developing the code for the simulation.

Question C.2.

Write a function `visualise_grid` to visualise a grid produced by `make_grid` with matplotlib. The function should take in a Boolean array. The output should look similar to Figure C.1: plot closed sites in black; plot open sites in white; colour the borders of each square in black.

[*Hint: you will need to consult the matplotlib documentation and other online resources to do this; the relevant matplotlib methods were not discussed in Chapter 7.*]

Use your function to visualise a few sample grids.

C.2 *Calculating percolation*

Once we have represented our grid, we now proceed to calculate whether each site is full or not. As with our grid, we represent the full status of each site with a numpy array of Boolean variables.

Question C.3.
Write a function `visualise_fill` to visualise the fill status of a given grid. The function should take as input two Boolean arrays, the grid and the fill status. The output should look similar to Figures C.2 and C.3. Plot closed sites in black, open unfilled sites in white, and open filled sites in blue. Colour the borders of each square in black.

Apply your function to hand-crafted data (e.g. on a 3×3 grid) to verify it is working correctly. Ensure also that the visualisation code works correctly if the fill status is all `False`.

We now turn to the central task of computing the full status of each site. This is more subtle than it appears. An outline might be the following.

1. Visit each site in the top row.
2. For each visited site, do the following:
   (a) If appropriate, set the site to be full.
   (b) Visit each neighbouring site (left, right, up, down).

This general approach is known in the graph theory literature as *depth-first search*. It is most naturally written as a function that recursively calls itself, but non-recursive implementations are also possible.

Question C.4. Write a function `compute_fill` that takes in a grid produced by `make_grid` and calculates whether each site is full or not.

[*Hint: think carefully about what should happen when a site is visited. For example, if it is already full, it should terminate without further action.*]

[*Hint: it may be useful during your development to visualise the fill state of the grid as you visit each site in the top row.*]

Question C.5. Write a function `percolates` that returns `True` if the given grid percolates, and `False` otherwise.

[*Hint: the core logic can be written with one line of numpy.*]

Draw 10 samples of a 20×20 grid with vacancy probability $p = 0.6$. For each, visualise its fill status, titling each figure with whether that grid percolates or not.

C.3 *Monte Carlo simulation*

With the `percolate` function in hand, we can now conduct our statistical simulation. Our goal here is to calculate $C(p)$, the percolation probability as a function of the vacancy probability.

Question C.6. Take a suitable grid $P \subset [0, 1]$ of p values. (You may wish to increase the resolution for $p \in [0.4, 0.7]$.) For each $p \in P$, draw N samples of a 20×20 grid with vacancy probability p. For each sample, calculate whether the grid percolates or not; the fraction of grids that percolates is our estimate for $C(p)$. Plot $C(p)$ as a function of p.

[*Hint: you will need to choose suitable N and P so that the interpolation error and statistical error due to sampling are acceptable. The curve should appear smooth; if it is not, try increasing N and/or refining P.*]

A word on computational efficiency is in order for question C.6. This question is the most computationally intensive across the three projects[9]. When calling `publish()` as usual, your code actually gets executed twice; normally this is not a problem, but here it may be. Instead, with the latest version of `publish.py` it is possible to execute

[9] On an old laptop, the unoptimised reference solution for question C.6 takes approximately 9 minutes.

(terminal) `python publish.py percolation.py`

which publishes your script, only executing it once.

C.4 *Concluding remarks*

The algorithms investigated in this project can be substantially improved upon. For example, Newman & Ziff propose an entirely different approach to the simulation of site percolation that is several million times faster for 1000×1000[10]. Their approach relies on an alternative representation of the state of the system, explicitly keeping track of each connected cluster as a tree; with this alternative representation, entirely different statistical ensembles and search algorithms are used.

[10] M. E. J. Newman and R. M. Ziff. Fast Monte Carlo algorithm for site or bond percolation. *Physical Review E*, 64:016706, 2001

In computational mathematics, there is a constant iteration between programming, computation, and theory; computations motivate new mathematical questions, and mathematical insights make new computations possible.

Bibliography

[1] Hugo Duminil-Copin wins the Fields medal. *Quanta Magazine*, 2022.

[2] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2 edition, 1996.

[3] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[4] W. R. Alford, A. Granville, and C. Pomerance. There are infinitely many Carmichael numbers. *Annals of Mathematics*, 139(3):703, 1994.

[5] K. Appel and W. Haken. Every planar map is four colorable. *Bulletin of the American Mathematical Society*, 82(5), 1976.

[6] R. Baillie and S. S. Wagstaff. Lucas pseudoprimes. *Mathematics of Computation*, 35(152):1391–1417, 1980.

[7] L. Boltzmann. Weitere Studien über das Wärmegleichgewicht unter Gasmolekülen. *Sitzungsberichte der Akademie der Wissenschaften zu Wien*, 66:275–730, 1872.

[8] F. Bornemann. PRIMES is in P: a breakthrough for ''Everyman''. *Notices of the American Mathematical Society*, 50(5):545–553, 2003.

[9] R. P. Brent. *Algorithms for Minimisation without Derivatives*. Prentice Hall, 1973.

[10] P. L. Chebyshev. *Oeuvres de P. L. Tchébychef*, volume 1. Commissionaires de l'Académie Impériale des Sciences, 1899–1907.

[11] C. S. Covaci. The Unsolvability of the Quintic: an Insight into Galois Theory. Master's thesis, Universidad Politécnica de Madrid, 2022.

[12] R. Crandall and C. Pomerance. *Prime Numbers: a Computational Perspective*. Springer-Verlag New York, second edition, 2010.

[13] R. Dawkins. *The Blind Watchmaker*. WW Norton & Company, 1996.

[14] J. R. Dormand and P. J. Prince. A family of embedded Runge–Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, 1980.

[15] D. R. Emerson, A. J. Sunderland, M. Ashworth, and K. J. Badcock. High performance computing and computational aerodynamics in the UK. *Aeronautical Journal*, 111(1117):125–131, 2007.

[16] A. Ginsburg et al. `astroquery`: an astronomical web-querying package in Python. *The Astronomical Journal*, 157(3):98, 2019.

[17] A. Meurer et al. SymPy: symbolic computing in Python. *PeerJ Computer Science*, page e103, 2017.

[18] A. W. Senior et al. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020.

[19] C. R. Harris et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

[20] P. Virtanen et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[21] L. Euler. *Institutiones Calculi Integralis*. Academia Imperialis Scientiarum, 1768. Translation by I. Bruce. Independently published.

[22] L. Pisano (Fibonacci). *Liber Abaci*. Springer Science & Business Media, 2003. Translation by L. E. Sigler.

[23] J. G. F. Francis. The QR Transformation: A Unitary Analogue to the LR Transformation—Part 1. *The Computer Journal*, 4(3):265–271, 1961.

[24] C. F. Gauss. *Methodvs nova integralivm valores per approximationem inveniendi*. Dieterich, 1815.

[25] C. F. Gauss. *Disquisitiones Arithmeticae*. Yale University Press, 1965. Translation by A. A. Clarke.

[26] J. D. Giorgini. Status of the JPL Horizons Ephemeris System. In *IAU General Assembly*, volume 29, page 2256293, 2015.

[27] D. Gruntz. *On computing limits in a symbolic manipulation system*. PhD thesis, ETH Zürich, 1996.

[28] E. Hairer, C. Lubich, and G. Wanner. Geometric numerical integration illustrated by the Störmer–Verlet method. *Acta Numerica*, 12:399–450, 2003.

[29] E. Hairer, C. Lubich, and G. Wanner. *Geometric Numerical Integration*, volume 31 of *Springer Series in Computational Matheematics*. Springer-Verlag, 2006.

[30] E. Hairer, G. Wanner, and S. P. Nørsett. *Solving Ordinary Differential Equations I*, volume 8 of *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, 2nd edition, 1993.

[31] D. A. Ham. *Object-oriented Programming in Python for Mathematicians*. 2023. Independently published.

[32] W. R. Hamilton. XV. On a general method in dynamics; by which the study of the motions of all free systems of attracting or repelling points is reduced to the search and differentiation of one central relation, or characteristic function. *Philosophical Transactions of the Royal Society*, 124:247–308, 1834.

[33] S. K. Hanson, A. D. Pollington, C. R. Waidmann, W. S. Kinman, A. M. Wende, J. L. Miller, J. A. Berger, W. J. Oldham, and H. D. Selby. Measurements of extinct fission products in nuclear bomb debris: determination of the yield of the Trinity nuclear test 70 y later. *Proceedings of the National Academy of Sciences of the United States of America*, 113(29):8104–8108, 2016.

[34] C. Hill. *Learning scientific programming with Python*. Cambridge University Press, second edition, 2020.

[35] N. J. Hitchin, G. B. Segal, and R. S. Ward. *Integrable systems: Twistors, loop groups, and Riemann surfaces*, volume 4 of *Oxford Graduate Texts in Mathematics*. Clarendon Press, 1999.

[36] J. D. Hunter. Matplotlib: a 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[37] G. Jaeschke. On strong pseudoprimes to several bases. *Mathematics of Computation*, 61(204):915–926, 1993.

[38] A. Jameson. Time dependent calculations using multigrid, with applications to unsteady flows past airfoils and wings. In *10th Computational Fluid Dynamics Conference*. American Institute of Aeronautics and Astronautics, 1991.

[39] A. B. Kempe. On the geographical problem of the four colours. *American Journal of Mathematics*, 2(3):193–200, 1879.

[40] L. F. Menabrea of Turin, Officer of the Military Engineers. Sketch of the analytical engine invented by Charles Babbage, Esq. *Scientific Memoirs, Selected from the Transactions of Foreign Academies of Science and Learned Societies*, 3:666–731, 1843. Translated by A. King, Countess of Lovelace.

[41] J.-L. Lagrange. Applications de la méthode exposée dans le mémoire précédent à la solution de différents problèmes de dynamique. *Mélanges de Philosophie et de Mathématiques de la Société Royale de Turin*, 2:196–298.

[42] S. Kwan Lam, A. Pitrou, and S. Seibert. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. Association for Computing Machinery, 2015.

[43] L. J. Lander and T. R. Parkin. Counterexample to Euler's conjecture on sums of like powers. *Bulletin of the American Mathematical Society*, 72(6):1079, 1966.

[44] H. P. Langtangen. *A Primer on Scientific Programming with Python*. Springer Berlin Heidelberg, 2016.

[45] P. S. Laplace. *Traité de Mécanique Céleste*, volume IV. Chez Courcier, Paris, 1805.

[46] G. Lari, M. Saillenfest, and M. Fenucci. Long-term evolution of the Galilean satellites: the capture of Callisto into resonance. *Astronomy & Astrophysics*, 639:A40, 2020.

[47] J. Liouville. Premier mémoire sur la détermination des intégrales dont la valeur est algébrique. *Journal de l'École Polytechnique*, XIV:124–148, 1833.

[48] J. C. Maxwell. V. Illustrations of the dynamical theory of gases. Part I. On the motions and collisions of perfectly elastic spheres. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 19(124):19–32, 1860.

[49] G. L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[50] M. E. J. Newman and R. M. Ziff. Efficient Monte Carlo algorithm and high-precision results for percolation. *Physical Review Letters*, 85(19):4104–4107, 2000.

[51] M. E. J. Newman and R. M. Ziff. Fast Monte Carlo algorithm for site or bond percolation. *Physical Review E*, 64:016706, 2001.

[52] I. Newton. *The Method of Fluxions and Infinite Series*. Henry Woodfall; and sold by John Nourse, 1671. Translated from the Author's Latin Original Not Yet Made Publick. To which is Subjoin'd, a Perpetual Comment Upon the Whole Work, By John Colson. Published in 1736.

[53] I. Newton. An account of the book entituled Commercium Epistolicum Collinii et Aliorum, de Analysi Promota. *Philosophical Transactions of the Royal Society of London*, 342:173–224, 1715.

[54] E. Noether. Invariante Variationsprobleme. *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, pages 235–257, 1918.

[55] J. North. *God's clockmaker: Richard of Wallingford and the invention of time*. Hambledon Continuum, London, England, 2004.

[56] R. Penrose. A generalized inverse for matrices. *Mathematical Proceedings of the Cambridge Philosophical Society*, 51(3):406–413, 1955.

[57] R. Piessens, E. de Doncker-Kapenga, C. W. Überhuber, and D. K. Kahaner. *QUADPACK: a subroutine package for automatic integration*, volume 1 of *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, 1983.

[58] C. Pomerance, J. L. Selfridge, and S. S. Wagstaff. The pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35(151):1003–1026, 1980.

[59] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[60] D. Richardson. Some undecidable problems involving elementary functions of a real variable. *Journal of Symbolic Logic*, 33(4):514–520, 1969.

[61] R. H. Risch. The problem of integration in finite terms. *Transactions of the American Mathematical Society*, 139(0):167–189, 1969.

[62] F. Spufford. *Red Plenty*. Faber & Faber, 2010.

[63] D. Stauffer and A. Aharony. *Introduction to Percolation Theory*. Taylor & Francis, second edition, 1994.

[64] C. Störmer. Sur les trajectoires des corpuscules électrisés. *Archives des Sciences Physiques et Naturelles*, 24:5–18, 113–158, 221–247, 1907.

[65] G. S. Strang. The fundamental theorem of linear algebra. *The American Mathematical Monthly*, 100(9):848–855, 1993.

[66] J. J. Sylvester. A new proof that a general quadric may be reduced to its canonical form (that is, a linear function of squares) by means of a real orthogonal substitution. *Messenger of Mathematics*, 19:1–5, 1889.

[67] G. I. Taylor. The formation of a blast wave by a very intense explosion. II. The atomic explosion of 1945. *Proceedings of the Royal Society A*, 201(1065):175–186, 1950.

[68] L. N. Trefethen. Floating point numbers and physics. *Newsletter of the London Mathematical Society*, November, 2021.

[69] L. N. Trefethen, Á. Birkisson, and T. A. Driscoll. *Exploring ODEs*. Society for Industrial & Applied Mathematics, 2018.

[70] A. M. Turing. Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1(1):287–308, 1948.

[71] J. VanderPlas. *Python Data Science Handbook*. O'Reilly Media, Inc., 2016.

[72] J. VanderPlas. *A Whirlwind Tour of Python*. O'Reilly Media, Inc., 2016.

[73] L. Verlet. Computer ''experiments'' on classical fluids. I. Thermodynamical properties of Lennard–Jones molecules. *Physical Review*, 159(1):98, 1967.

[74] R. Wilson. *Four Colors Suffice: How the Map Problem Was Solved*. Princeton University Press, 2013.

[75] G. Zhong and J. E. Marsden. Lie–Poisson Hamilton–Jacobi theory and Lie–Poisson integrators. *Physics Letters A*, 133(3):134–139, 1988.

[76] G. Zolotareff. Sur la méthode d'intégration de M. Tchébychef. *Mathematische Annalen*, 5(4):560–580, 1872.

[77] Εκλεδης. Στοιχεα. 300 B.C. Translation by R. Fitzpatrick. Independently published.